University of Augsburg

Department of Computer Science

Learning to Explore: Reinforcement Learning with Scene Graph Integration for Full Environment Coverage

Master's Thesis
for the Degree of
M. Sc. in Computer Science

Roman Küble

Matriculation Number:	2138785
Date of Submission:	29.09.2025
Supervisor:	Marco Hüller
Examiner:	Prof. Dr. Jörg Hähner

Abstract

The field of *Embodied AI* has gained considerable traction in recent years, focusing on agents that not only navigate but also semantically understand their environment. Such capabilities are foundational for a new generation of applications, including autonomous service robots in domestic settings and intelligent systems for search-and-rescue missions. This thesis addresses this challenge through the task of *Embodied Semantic Scene Graph Generation (ESSGG)*, where an agent's primary objective is to build a complete and accurate semantic graph of its surroundings. The main baseline for this task relies on a Recurrent Neural Network (RNN) and the foundational reinforcement learning (RL) algorithm REINFORCE, which exhibit limitations in processing long-term dependencies and can suffer from high-variance gradients, respectively.

To overcome these challenges, this work designs, implements, and evaluates a novel navigation module modernized on two key levels. First, the RNN-based architecture is replaced by a *Transformer-based model* to more effectively process the long sequence of observations and actions inherent in exploration tasks. Second, the high-variance REINFORCE algorithm is substituted with a more stable and data-efficient *Advantage Actor-Critic (A2C) agent* to foster robust policy convergence. The proposed agent is implemented and systematically evaluated within the Ai2-THOR simulation environment.

The central hypothesis is that this synergistic combination of a modern architecture and an advanced learning algorithm will lead to significant improvements in both the quality of the generated scene graph and the efficiency of the exploration process, ultimately enabling the agent to achieve full environment coverage.

The experimental evaluation reveals that the algorithmic modernization to A2C is the dominant factor, leading to significantly more stable policies and superior performance across all metrics, including scene graph quality, exploration efficiency, and generalization. While the Transformer-based architecture's performance was similar to the RNN network's, it did not provide a decisive advantage in the conducted experiments. This leads to the central conclusion that, in the investigated setup, the choice of a stable learning algorithm is substantially more critical to the agent's success than the choice of the sequential model.

Zusammenfassung

Das Forschungsfeld der *verkörperten KI (Embodied AI)* hat in den letzten Jahren erheblich an Bedeutung gewonnen und konzentriert sich auf Agenten, die in ihrer Umgebung nicht nur navigieren, sondern sie auch semantisch verstehen. Solche Fähigkeiten sind grundlegend für eine neue Generation von Anwendungen, darunter autonome Serviceroboter im häuslichen Umfeld und intelligente Systeme für Such- und Rettungseinsätze. Diese Arbeit befasst sich mit dieser Herausforderung im Rahmen der Aufgabe der *Embodied Semantic Scene Graph Generation (ESSGG)*, bei der das Hauptziel eines Agenten darin besteht, einen vollständigen und genauen semantischen Graphen seiner Umgebung zu erstellen. Die primäre Baseline für diese Aufgabe stützt sich auf ein rekurrentes neuronales Netzwerk (RNN) und den grundlegenden Reinforcement Learning (RL) Algorithmus REINFORCE, welche jedoch Limitationen bei der Verarbeitung langfristiger Abhängigkeiten aufweisen und unter Gradienten mit hoher Varianz leiden können.

Um diese Herausforderungen zu überwinden, entwirft, implementiert und evaluiert diese Arbeit ein neuartiges Navigationsmodul, das auf zwei zentralen Ebenen modernisiert wird. Erstens wird die RNN-basierte Architektur durch ein *Transformer-basiertes Modell* ersetzt, um die lange Sequenz von Beobachtungen und Aktionen, die für Explorationsaufgaben typisch ist, effektiver zu verarbeiten. Zweitens wird der varianzanfällige REINFORCE-Algorithmus durch einen stabileren und dateneffizienteren *Advantage Actor-Critic (A2C) Agenten* ersetzt, um eine robuste Konvergenz der Handlungsstrategie zu fördern. Der vorgeschlagene Agent wird implementiert und systematisch in der Ai2-THOR-Simulationsumgebung evaluiert.

Die zentrale Hypothese lautet, dass diese synergetische Kombination aus moderner Architektur und fortschrittlichem Lernalgorithmus zu signifikanten Verbesserungen sowohl bei der Qualität des generierten Szenengraphen als auch bei der Effizienz des Explorationsprozesses führt und dem Agenten schlussendlich eine vollständige Abdeckung der Umgebung ermöglicht.

Die experimentelle Evaluation zeigt, dass die algorithmische Modernisierung zu A2C der dominante Faktor ist, der zu signifikant stabileren Policies und einer überlegenen Leistung in allen Metriken führt, einschließlich der Qualität des Szenengraphen, der Explorationseffizienz und der Generalisierungsfähigkeit. Obwohl die Transformer-basierte Architektur eine ebenbürtige Leistung zum RNN-Netzwerk zeigte, lieferte sie in den durchgeführten Experimenten keinen entscheidenden Vorteil, was zur zentralen Schlussfolgerung führt, dass im untersuchten Setup die Wahl eines stabilen Lernalgorithmus wesentlich entscheidender für den Erfolg des Agenten ist als die Wahl des sequentiellen Modells.

Contents

List of Figures

List of Tables

Abbreviations

A2C	Advantage Actor-Critic
ACO	Ant Colony Optimization
AI	Artificial Intelligence
BC	Behavioral Cloning
CNN	Convolutional Neural Network
DQN	Deep Q-Network
ESSGG	Embodied Semantic Scene Graph Generation
GAT	Graph Attention Network
GNN	Graph Neural Network
GSSG	Global Semantic Scene Graph
HGT	Heterogeneous Graph Transformer
HRON	Hierarchical Relational Object Navigation
IL	Imitation Learning
IQR	Interquartile Range
LayerNorm	Layer Normalization
LSSG	Local Semantic Scene Graph
LSTM	Long Short-Term Memory
MDP	Markov Decision Process
MLP	Multi-Layer Perceptron
PPO	Proximal Policy Optimization
ReLU	Rectified Linear Unit
ResNet	Residual Network
RL	Reinforcement Learning
RNN	Recurrent Neural Network
SE	Standard Error
SLAM	Simultaneous Localization and Mapping
TD	Temporal-Difference
TSP	Traveling Salesman Problem
VLM	Vision-Language Model

1 Introduction

1.1 Motivation and Problem Context

The research in the field of artificial intelligence (AI) has achieved transformative progress over the past decades, particularly in passive perception such as image classification or natural language processing. However, while these advances have been substantial, a major challenge remains in the development of active, *embodied AI* — intelligent agents that not only perceive passively, but also autonomously navigate and interact with complex, dynamic environments in order to achieve higher-level goals [18]. This development marks a fundamental shift from algorithms that process data to agents that gather experiences and build an understanding of their environment.

The growing importance of embodied AI is reflected not only in theoretical models, but is also evident in the exponentially increasing number of publications in this field. Figure 1 shows the annual distribution of documents indexed in Scopus with the search term "Embodied AI" (title/abstract/keywords) from 1974 to 2024 [49]. Since around 2018, a steep increase can be observed, reaching a preliminary peak in 2024 with nearly 380 articles. This trend quantitatively underscores the growing interest and relevance of the research field of embodied AI.

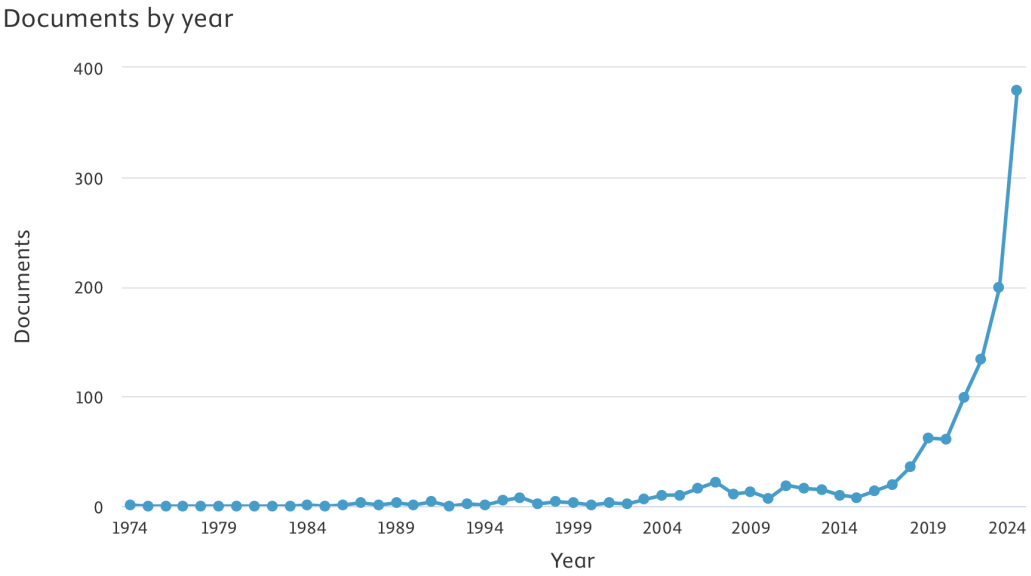

Figure 1: Annual distribution of publications indexed in Scopus with the keyword "Embodied AI" (1974-2024) [49]

At a conceptual level, the core of this research field lies in the ability to *navigate and explore*. Efficient movement through an unknown environment is a fundamental prerequisite for nearly any subsequent task, whether it be object manipulation, answering questions about the environment, or interacting with humans. The focus is not only on the purely geometric traversal of a space, but increasingly on achieving a semantic

understanding of the environment — that is, recognizing objects, their properties, and their relationships to one another, and using this knowledge for intelligent action.

A pioneering approach that formalizes this idea of semantic exploration is the task of *Embodied Semantic Scene Graph Generation (ESSGG)*, introduced by Li et al. [34]. Instead of viewing navigation merely as a means to an end, the explicit goal of exploration here is the generation of a comprehensive and accurate semantic scene graph. This creates a symbiotic loop: the graph already constructed informs the navigation policy about knowledge gaps, and the navigation policy selects actions that promise the greatest gain of high-quality semantic information. While the problem definition of ESSGG is conceptually advanced, the technical solution implemented by Li et al. reflects the state of the art of its time. The navigation module is based on an *Long Short-Term Memory (LSTM)* architecture, which is known to have limitations in handling long-term dependencies, and relies on the fundamental *REINFORCE* algorithm, which is recognized for its high variance and inefficient data usage [57, 52]. It is precisely in these two aspects — the architecture for long-term memory and the learning algorithm for behavior optimization — that the research gaps addressed in this work become apparent.

1.2 Aim of This Thesis

The central goal of this work is the design, implementation, and evaluation of a novel navigation module for ESSGG that overcomes the limitations of the baseline by Li et al. [34] on both the architectural and algorithmic levels. Specifically, two key improvements are introduced:

1. **Architectural modernization**
   The *LSTM*-based architecture for processing the exploration history is replaced with a *Transformer-based model*. This model is better suited to capturing long-term dependencies between observations, actions, and the evolving scene graph.

2. **Algorithmic modernization**
   The *REINFORCE* algorithm, which is prone to high variance, is replaced by a modern and more stable *Advantage Actor-Critic (A2C)* agent. This is intended to ensure more efficient and robust convergence of the navigation policy.

The **core hypothesis** of this work is:

> *The combination of a Transformer architecture for handling long-term dependencies and a stable A2C learning procedure improves performance on the ESSGG task, as measured by the quality of the final scene graph (recall) and the efficiency of the exploration path (number of steps).*

To rigorously isolate the contributions of both components, they are evaluated in a 2×2 ablation design (*LSTM* $\leftrightarrow$ *Transformer* $\times$ *REINFORCE* $\leftrightarrow$ *A2C*). This allows their causal effects to be demonstrated separately.

1.3 Structure

This thesis is structured as follows: Chapter 2, *Background*, introduces the theoretical foundations of machine learning and the methods relevant to this work. Building on this, Chapter 3, *Related Work*, analyzes the state of the art in detail and highlights recent advances that motivate the proposed contributions. Chapter 4, *Ai2-THOR Environment*, presents the simulation environment that serves as the basis for the experiments. The central reinforcement learning algorithms applied in this work are explained in Chapter 5, *Learning Algorithms*. Chapter 6, *Methodology*, describes the conceptual design and implementation of the novel navigation module. The conducted evaluations and the resulting findings are presented in Chapter 7, *Evaluation*. Finally, Chapter 8, *Conclusion and Future Work*, summarizes the key insights, discusses limitations, and provides an outlook on future research directions.

To ensure the reproducibility of the experiments, the complete source code of this work has been released under an open license on GitHub. The repository is available at the following link:

https://github.com/kueblero/Learning-to-Explore---Reinforcement-Learning-w-Scene-Graph-Integration-for-Full-Environment-Coverage

2 Background

This section introduces the central concepts, methods, and structures necessary for understanding the present work. Its aim is to establish the methodological foundation for the subsequent chapters and to situate the most important theoretical concepts.

2.1 Embodied AI and Visual Exploration

Embodied Artificial Intelligence [17] refers to AI systems that can interact with their environment through a virtual or physical body. Such systems are capable of capturing perceptions through sensors and carrying out actions via actuators.

A central challenge in the field of embodied AI is *embodied visual exploration* [44]. Here, an agent must learn to move autonomously in an unknown environment in order to construct an internal model of it. Such models can range from simple metric maps that merely indicate navigable space to complex semantic representations that also capture objects and their relationships to one another.

2.2 Environment Representation via Scene Graphs

For an embodied agent to perform complex, goal-directed tasks beyond simple reactive behaviors, it requires an internal model of its environment. Since the agent perceives the world only through local observations, this internal representation serves as memory to understand spatial and semantic relationships and to plan actions efficiently. For tasks that demand a deep semantic understanding, *scene graphs* have emerged as a particularly powerful structured representation [32].

Scene graphs [9] model an environment as a graph in which the nodes represent objects (e.g. “table”, “chair”) and the edges denote the semantic relationships between these objects (e.g. “stands on”, “is next to”). Rather than viewing the world as a collection of pixels or metric points, they capture an abstract, human-like understanding of the scene. This enables the agent to reason about objects and their relationships, rather than relying solely on visual similarity, which is especially valuable for complex interactions.

2.3 Neural Networks

Artificial Neural Networks [19] are machine learning models whose structure is loosely inspired by the biological neural networks of the human brain. They consist of a large number of interconnected processing units, known as *neurons*, which are typically organized in layers. A typical neural network comprises an *input layer*, one or more *hidden layers*, and an *output layer*. A network is considered *deep* once it contains more than one hidden layer. Figure 2 illustrates such a multilayer architecture.

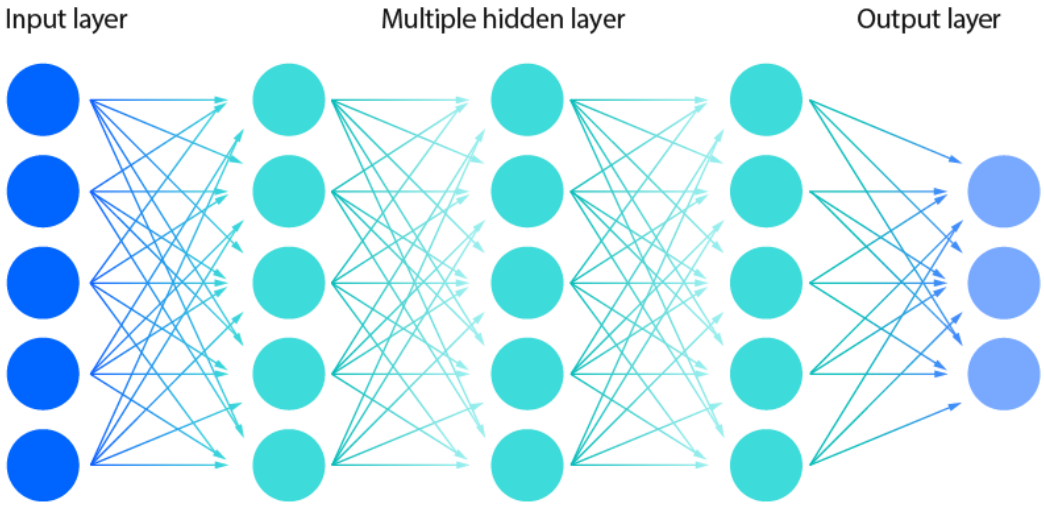

Figure 2: Architecture of a deep neural network [25]

Each neuron receives weighted signals from the neurons of the preceding layer, sums them, and adds a so-called bias term. The result is then passed through a nonlinear *activation function* (e.g. sigmoid, tanh, or ReLU), which determines the final output of the neuron. This nonlinearity is crucial, as it enables the network to learn complex, nonlinear relationships in the data. An abstract depiction of the functionality of a single neuron, which receives multiple input signals X and produces a single output, is shown in Figure 3.

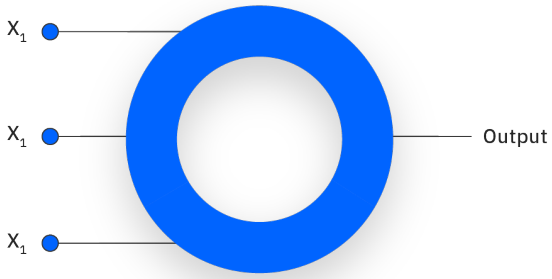

Figure 3: Schematic representation of a neuron [25]

One of the most widely used activation functions today is the *Rectified Linear Unit (ReLU)* [39] (see Figure 4). Defined as $f(x) = \max(0, x)$, it is computationally efficient and plays a key role in mitigating the vanishing gradient problem[1]. Unlike the sigmoid or tanh functions, whose derivatives approach zero for large input values, ReLU has a constant derivative of 1 for all positive inputs. This allows gradients to flow more effectively through deep networks.

[1]The vanishing gradient problem refers to the tendency of gradients to shrink exponentially as they are propagated back through many layers in deep networks, which hampers effective training of earlier layers [8, 21].

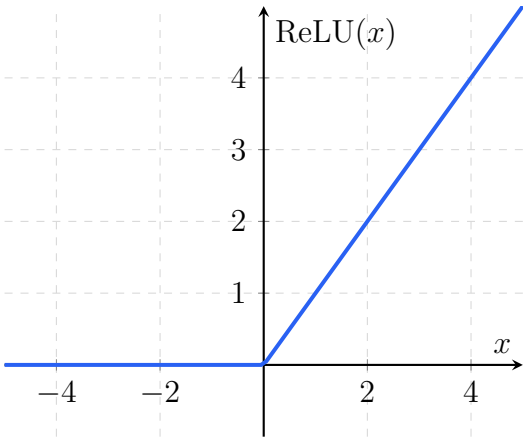

Figure 4: ReLU activation function $f(x) = \max(0, x)$

The learning process of a neural network, commonly referred to as *training*, is an optimization problem. The objective is to adjust the weights and bias terms of the network so that a predefined *loss function* is minimized. This function quantifies the discrepancy between the network's predictions and the actual target values from the training data. To minimize the loss, the algorithm of *backpropagation* is employed in conjunction with an optimization method such as *gradient descent* or its variants (e.g. Adam [30]). Backpropagation efficiently computes the gradient of the loss function with respect to all weights in the network, after which the weights are updated in the direction of steepest descent of the loss.

To stabilize and accelerate the training of deep networks, normalization techniques are employed. One such technique is *Layer Normalization (LayerNorm)* [5]. Its goal is to mitigate the problem of *internal covariate shift*, in which the distribution of the inputs to each layer changes continuously during training. In contrast to batch normalization, which normalizes across the entire data batch, LayerNorm computes the normalization statistics (mean and variance) for each individual data point across all neurons within the same layer. The activations a of a layer are then normalized, scaled, and shifted for that data point as follows:

$$y = \gamma \frac{a - \mu}{\sqrt{\sigma^2 + \epsilon}} + \beta,$$

where μ and σ^2 denote the mean and variance across features of a, ϵ is a small constant for numerical stability, and γ (scale) and β (shift) are learnable parameters.

This approach is particularly effective in architectures such as Transformers, since it is independent of batch size and well suited for processing sequences of variable length.

Neural networks process continuous, numerical data. Many real-world problems, however, involve discrete, categorical inputs such as words, object categories, or, as in this work, actions. *Embeddings* are a fundamental technique to transform such categorical

data into dense, low-dimensional, and continuous vector representations that a network can process. Instead of representing categories through simple, information-poor one-hot vectors, an embedding layer learns a dedicated feature vector for each category. The key advantage is that these vectors are optimized during training to encode semantic relationships: categories with similar meaning or function obtain similar representations in the vector space. This enables the network to identify generalizable patterns and to transfer knowledge across related categories.

2.3.1 Convolutional Neural Networks

Convolutional Neural Networks (CNNs) [33, 19] are a specialized class of neural networks that have proven highly effective for processing data with a grid-like topology, such as images. In contrast to traditional fully connected networks, where every input is connected to every neuron in the next layer, CNNs employ specialized architectures to exploit the spatial hierarchies in the data while drastically reducing the number of learnable parameters.

The fundamental building blocks of a CNN are specialized layers:

- **Convolutional Layer**
  This layer forms the core of a CNN. Its purpose is to detect local features such as edges, corners, or textures in the input data (e.g. an image). Instead of full connectivity, this is achieved through the process of *convolution*: a small, learnable filter, called a *kernel*, is systematically slid across the input. Figure 5 illustrates this procedure. At each position, the kernel is applied to the corresponding patch of the input, the element-wise products are computed, and the results are summed into a single output value.

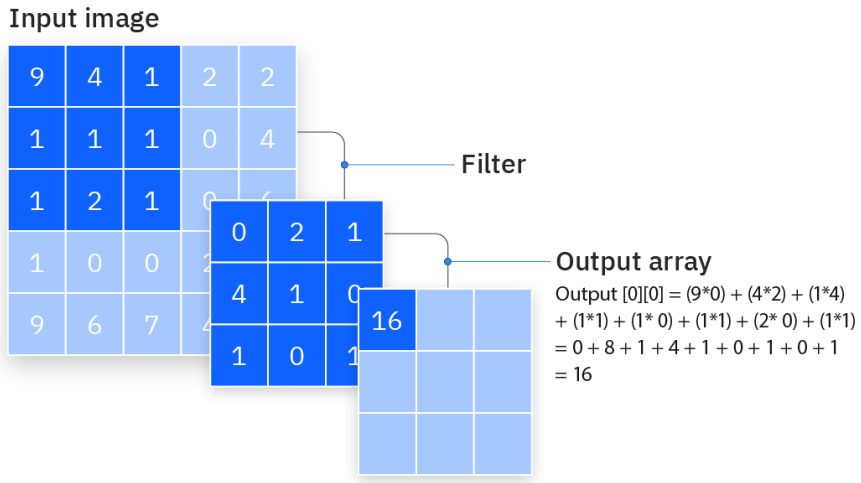

Figure 5: Visualization of the convolution process [27]

The step size of this movement is called the *stride*; a stride of 1 shifts the filter by a single pixel, while a larger stride reduces the output dimensions. A key

advantage of this architecture is *parameter sharing*: since the same kernel is reused across the entire image, the network can recognize a learned feature at any spatial position. This makes the representation translation-invariant and drastically more parameter-efficient. The result of convolving a kernel with the entire input is a *feature map*, which indicates where the specific feature is detected in the image.

- **Pooling Layer**
  A convolutional layer is often followed by a pooling layer (e.g. max pooling). Its primary function is to reduce the spatial dimensions of the feature maps (down-sampling). This makes the representation more compact, reduces computational cost, and provides the model with a degree of robustness to small shifts of features in the input image.

- **Fully Connected Layer**
  At the end of a CNN architecture, one or more fully connected layers are typically employed. These layers take the high-level features extracted by the preceding convolutional and pooling layers, flatten them into a vector, and perform the final classification or regression task.

By stacking convolutional and pooling layers, CNNs can learn a hierarchy of features — starting from low-level patterns such as edges and corners in the early layers, up to high-level concepts in the deeper layers [19].

Building on these fundamental layers, increasingly deeper architectures were developed to enhance performance. A decisive breakthrough for training very deep networks was the introduction of the *Residual Network (ResNet)* [20]. ResNets were designed to address the *vanishing gradient problem*, which becomes more pronounced as network depth increases. The core idea is *residual learning*, where individual blocks in the network do not attempt to learn a direct mapping, but instead focus on learning the residual (difference) relative to their input. This is achieved through so-called *skip connections*, which additively link the input of a block directly to its output (see Figure 6). These connections enable more effective gradient flow during training and make it possible to construct networks with hundreds or even thousands of layers, leading to groundbreaking advances in image recognition.

The enormous computational power and data required to train deep architectures such as ResNet from scratch has made the reuse of already trained models a common and highly effective practice. This approach is known as *transfer learning* [64]. The core idea is to leverage a model that has already been trained on a very large and general dataset and use it as a starting point for a new, more specific task.

A prominent example, also applied in this work, is the use of a ResNet model pre-trained on the *ImageNet* dataset. ImageNet contains millions of images across thou-

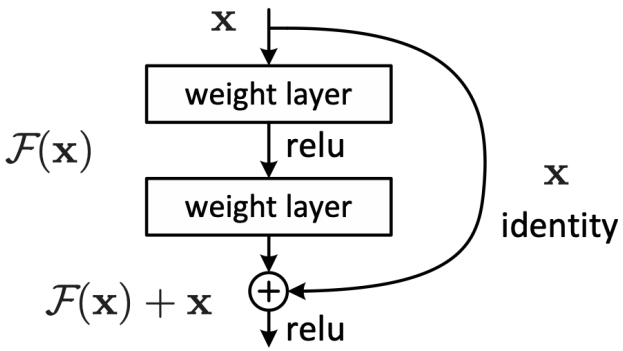

Figure 6: A single residual block [20]

sands of categories [13]. A network trained on this dataset has already learned to extract a rich hierarchy of visual features — from simple edges and textures in the lower layers to complex object parts in the higher layers. For a new task, these learned weights can be used as initialization. Typically, the lower layers of the network (the *feature extractor*) are frozen and only the top layers are retrained for the specific task, or the entire network is fine-tuned with a very small learning rate. This not only saves significant training time and resources, but also leads to substantially better results on smaller, task-specific datasets.

2.3.2 Recurrent Neural Networks

While CNNs are specialized for processing spatial data, *Recurrent Neural Networks (RNNs)* [19] were developed for handling sequential data such as time series or text. In contrast to pure feedforward networks, where information flows only in one direction through the layers, RNNs incorporate recurrent connections in their architecture. As illustrated in Figure 7, this loop in the hidden layer allows the output of a neuron at time step t to depend not only on the current input x_t, but also on the activation from the previous time step $t-1$. This structure, which is often shown in a compact "rolled" form, can be "unrolled" through time to highlight the sequence of dependencies.

This internal state, also called the *hidden state*, enables the network to persist information across longer sequences and to model contextual dependencies. In practice, however, simple RNN architectures suffer from the *vanishing or exploding gradient problem* [21]. During *backpropagation through time*, gradients are repeatedly multiplied by the same weights. As a result, they either shrink exponentially — preventing the learning of long-range dependencies (vanishing gradient) — or grow exponentially, making training unstable (exploding gradient).

To address the vanishing gradient problem, specialized RNN architectures were developed, among which the *Long Short-Term Memory (LSTM)* network [22] is one of the most influential. LSTMs extend the simple RNN neuron with an explicit memory

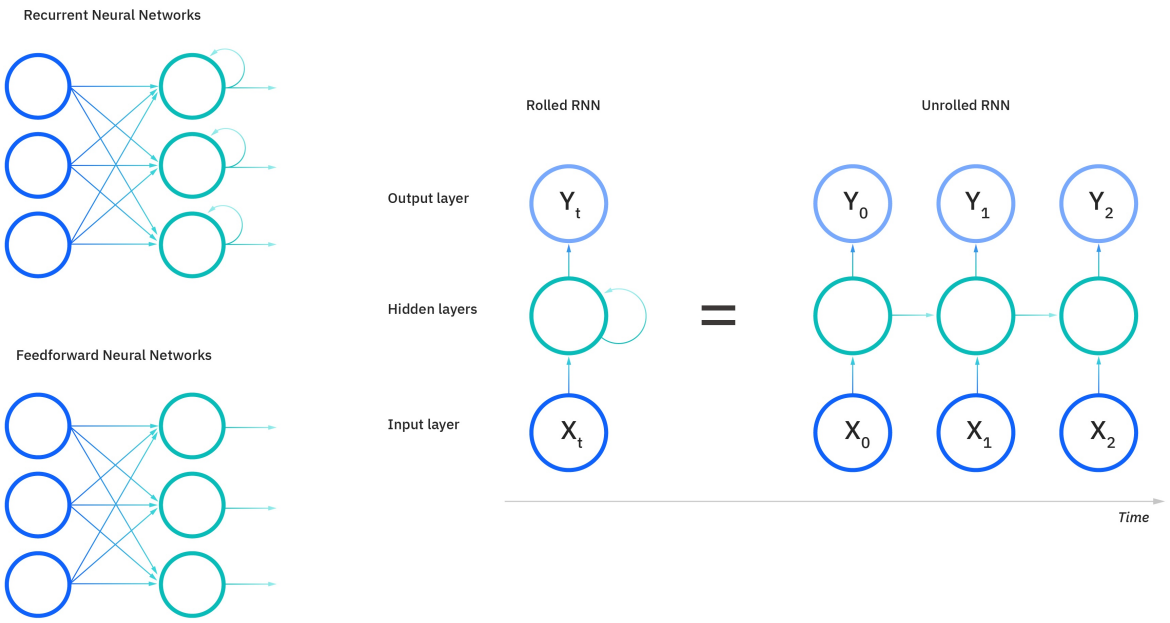

Figure 7: Feedforward vs. Recurrent Neural Networks [26]

structure, known as the *cell state*. This state can carry information across long time spans with little modification. Control over the flow of information into and out of the cell state is regulated by three specialized gating units, called *gates* (see Figure 8): the *forget gate* decides which old information to discard, the *input gate* determines which new information to store, and the *output gate* controls which part of the memory is used for the output. Through this mechanism, LSTMs can selectively retain or forget information over very long time horizons and are thus capable of effectively learning long-range dependencies in sequential data.

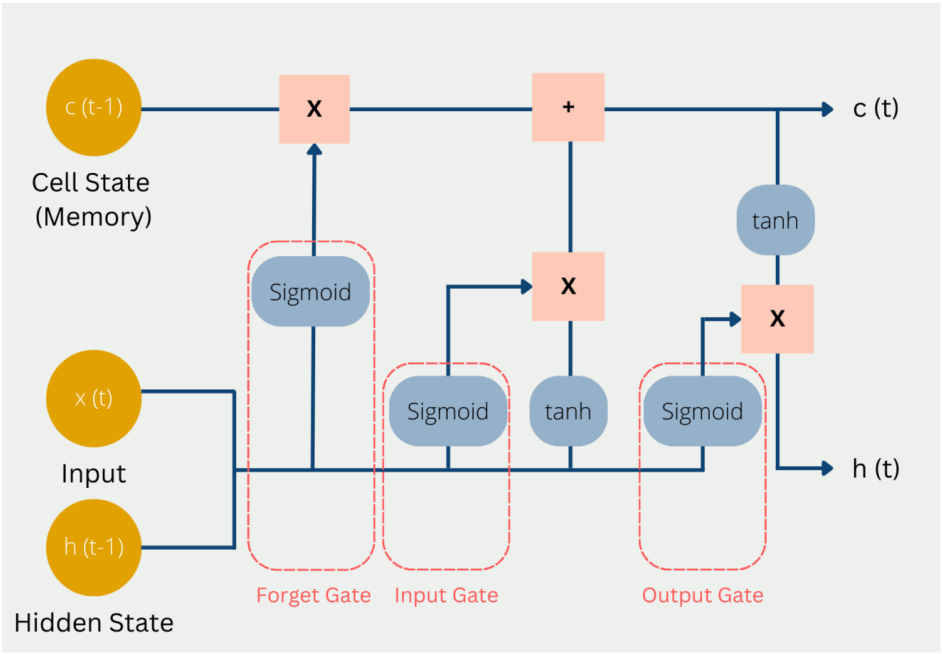

Figure 8: LSTM architecture [12]

2.3.3 Attention Mechanism and Transformer Networks

Although LSTMs mitigate the vanishing gradient problem, their inherently sequential processing remains a bottleneck that hinders parallelization and makes modeling very long dependencies challenging. To overcome this limitation, the *attention mechanism* [6] was introduced. The core idea is to avoid compressing an entire sequence into a single static context vector, and instead allow the model to dynamically focus on the parts of the input that are most relevant for a given task.

This is achieved by computing a learnable importance weight (attention weight) for each part of the input. The result is a context-sensitive weighted sum of the input features that highlights the most important information. This principle is applied in so-called *attention pooling*, where a set of feature vectors is aggregated into a single, informative representation. Applications range from aggregating node features into a single expressive graph representation [56] to weighting object features in a visual scene in order to direct an agent's attention to the information most relevant for the current task [11].

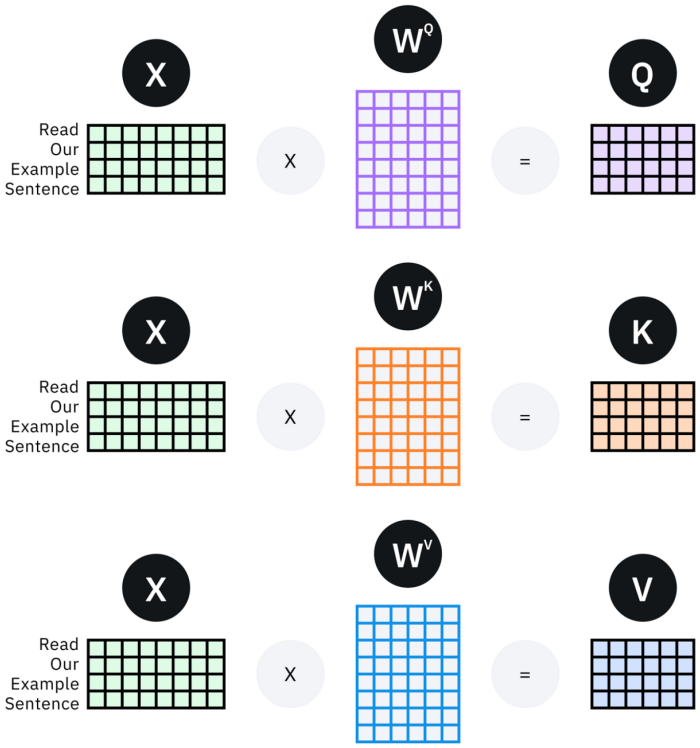

Figure 9: Generation of query, key, and value vectors from an input sequence X through multiplication with learned weight matrices [28]

The *Transformer* architecture [55] refines this idea by completely eliminating recurrence and relying instead on *self-attention*. Here, attention is not computed between an input and an output sequence, but within the input sequence itself. The self-attention mechanism is based on the *query-key-value* concept. As illustrated in Figure 9, for each

token vector in the input sequence, three new vectors are generated by multiplying it with three separate learned weight matrices: a query (Q), a key (K), and a value (V). The query represents the information being requested by the token, the key encodes the attributes of all other tokens, and the value contains the actual information to be aggregated.

The similarity between a token's query and the keys of all other tokens in the sequence determines the attention weights. The final output for the querying token is then computed as the weighted sum of the value vectors of all tokens. In this way, each token can contextually aggregate information from the entire sequence.

Multi-Head Attention and Positional Encoding To capture different aspects of relationships (e.g. syntactic and semantic) simultaneously, the self-attention process is executed in parallel with different learned projections of Q, K, and V in *multi-head attention* [55] (see Figure 10). Since this parallel processing does not inherently encode information about the order of tokens, positional information is artificially introduced through an explicit *positional encoding*. This encoding assigns each position in the sequence a unique vector, which is added to the corresponding token embedding so that the model can distinguish the order of elements.

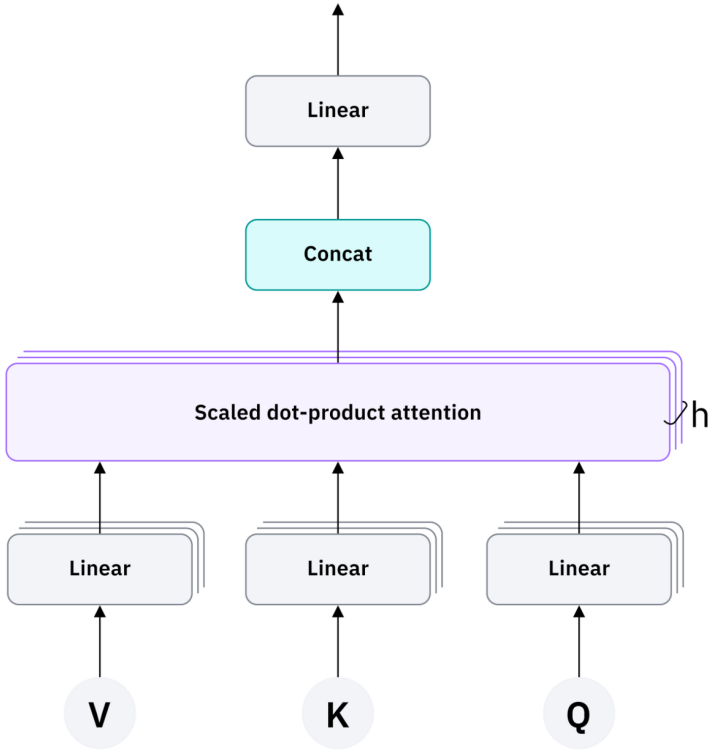

Figure 10: Architecture of multi-head attention [28] — Q, K, and V are split across multiple parallel scaled dot-product attention heads

Transformer Encoder The architecture of a *Transformer encoder block*, the central building unit of a Transformer encoder, is shown in Figure 11. The process begins with the input sequence ("Inputs"), which is first converted into numerical vectors called "input embeddings." A "positional encoding" is then added to these embeddings to reintroduce the information about token order that would otherwise be lost.

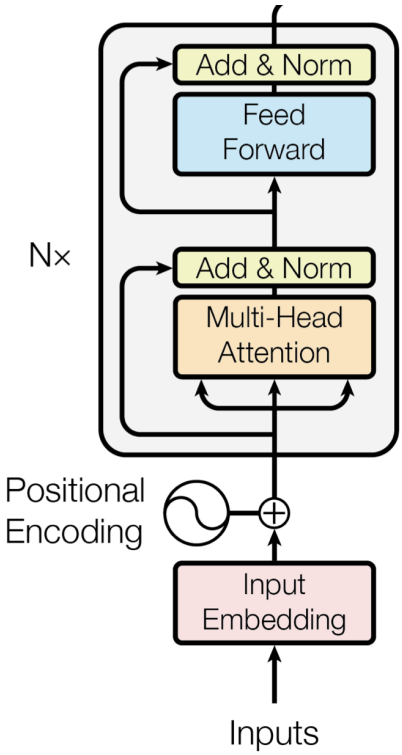

Figure 11: Architecture of a Transformer encoder block, based on [55]

These prepared vectors then pass through the two main sublayers of the block. The first is a multi-head attention layer that models the contextual relationships between all tokens in the sequence. This is followed by an *"Add & Norm"* operation: here, the original sublayer input is added to the output of the attention layer via a residual connection, and the result is normalized using layer normalization. The second sublayer, a position-wise *Feed-Forward Network*, further processes the output and is likewise followed by an "Add & Norm" operation. A complete Transformer encoder is constructed by stacking multiple identical blocks, as indicated by the "Nx" notation in the figure. Its function is to generate a rich, contextualized representation for each token in the input sequence, which can then be used for downstream tasks.

2.3.4 Graph Neural Networks

While the architectures discussed so far — such as CNNs, RNNs, and Transformers — are specialized for data with a fixed, regular structure (grids, sequences), *Graph Neural Networks (GNNs)* [58] were developed to operate directly on irregularly structured data that can be represented as graphs. The fundamental goal of a GNN is to learn an

expressive feature representation for each node in the graph (a *node embedding*) that incorporates both the node's own attributes and information from its structural neighborhood. This is typically achieved through an iterative process of *message passing*, in which each node collects information from its neighbors (aggregation) and updates its own state accordingly (update).

Graph Attention Networks *Graph Attention Networks (GATs)* [56] are an influential class of GNNs that apply the attention mechanism to neighborhood aggregation. Instead of weighting all neighboring nodes equally (e.g. by averaging), GATs dynamically learn attention coefficients. These coefficients determine how much influence each neighbor should have on the update of the central node. This enables a more flexible and expressive aggregation, as the model can learn to assign greater importance to the most relevant neighbors.

Heterogeneous Graph Transformer While GATs are primarily designed for homogeneous graphs (a single node and edge type), many real-world graphs — in particular semantic scene graphs — consist of multiple types of nodes and edges. Such graphs are referred to as *heterogeneous graphs* [51]. The *Heterogeneous Graph Transformer (HGT)* [23] is an architecture specifically developed for this kind of data. HGT extends the attention mechanism by making its computations type-dependent. For each possible relation (source node type, edge type, target node type), distinct learnable projection matrices are used. This allows the model to capture the different semantics of various relationships. Accordingly, the attention between two nodes is determined not only by their states, but also by their types and the type of their connection. This enables the model to process complex heterogeneous structures in a differentiated manner, such as "an [object: chair] *stands on* a [object: floor] in a [room: kitchen]".

2.4 Paradigms of Machine Learning

Machine learning can broadly be divided into three paradigms: supervised learning, unsupervised learning, and reinforcement learning.

Supervised learning [40] is characterized by the use of labeled data. The algorithm is provided with pairs of inputs and their corresponding correct outputs. The objective is to learn a generalizable function that can correctly map new, unseen inputs to the appropriate outputs. Typical applications include classification and regression problems.

In contrast, *unsupervised learning* [29] operates on unlabeled data. The goal is not to predict a specific output, but to autonomously identify inherent patterns, structures, or hidden relationships in the data. The most prominent methods include clustering, which groups similar data points, and dimensionality reduction, which compresses

information into a lower-dimensional space.

Reinforcement learning [52] combines elements of both approaches and learns through trial and error by interacting with an environment, where an agent adapts its strategy based on received rewards. Since reinforcement learning is the primary focus of this work, it will be discussed in detail in Section 2.4.1.

Another paradigm relevant to this thesis is *imitation learning* [24], in which an agent learns by mimicking the behavior of an expert rather than relying on an explicit reward function. Although imitation learning is formally a variant of supervised learning, it plays a distinct role in this work and will therefore be discussed in more detail in Section 2.4.2.

2.4.1 Reinforcement Learning (RL)

RL [52] describes a process in which an agent learns to develop an optimal behavior strategy (*policy*) through interaction with an environment. At the core of this paradigm lies an iterative feedback cycle (see Figure 12): the agent observes a state (S_t), selects an action (A_t), and receives from the environment a scalar reward (R_t) that evaluates this action. The fundamental objective is to maximize the cumulative reward over time. Mathematically, this process is modeled as a *Markov Decision Process (MDP)*.

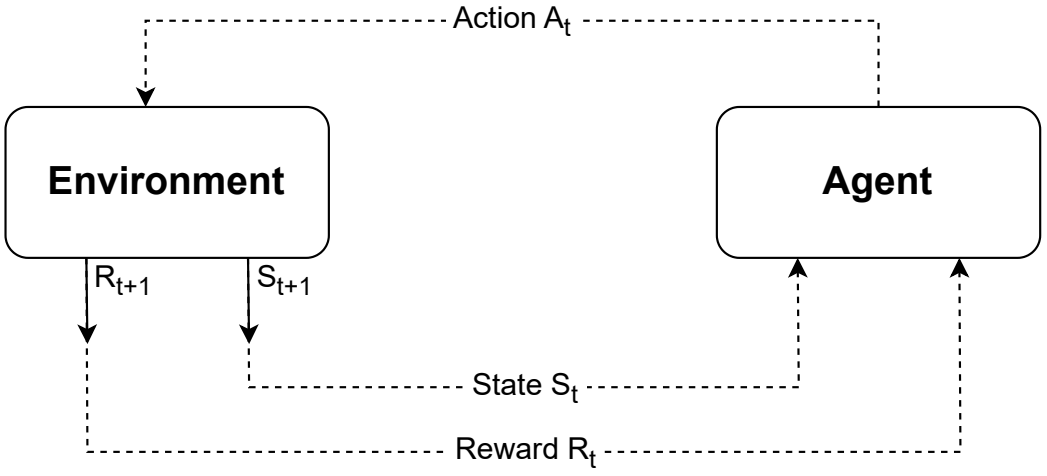

Figure 12: RL cycle (based on [52])

The design of the reward function is of central importance in RL. In general, the reward serves as the driving incentive that guides the agent's learning process. In the context of this thesis, which focuses on exploration, the reward does not primarily indicate the achievement of a predefined goal, but instead incentivizes the agent to gradually improve its strategy and to expand its understanding of the environment.

Markov Decision Process A *Markov Decision Process (MDP)* [52, 59] formalizes the interaction of an agent with an environment as a sequential decision-making prob-

lem under uncertainty. An MDP is defined as a tuple $(\mathcal{S}, \mathcal{A}, T, R, \gamma)$, consisting of a state space $\mathcal{S}$, an action space $\mathcal{A}$, a transition function $T(s' \mid s, a)$, a reward function $R(s, a, s')$, and a discount factor $\gamma \in [0, 1)$. The defining property is the *Markov property*, which states that the future depends only on the present state and action, not on the past. By convention, a distinction is made between concrete values (lowercase letters such as s, a) and random variables (uppercase letters such as S_t, A_t), which represent the state and action at time step t, respectively.

A stochastic policy $\pi(a \mid s)$ selects actions based on the current state. The overarching objective within this framework is to learn a policy that maximizes the expected discounted return $J(\pi)$:

$$J(\pi) = \mathbb{E}_{\pi,T}\left[\sum_{t=0}^{\infty} \gamma^t \, R(S_t, A_t, S_{t+1})\right] \tag{1}$$

Exploration vs. Exploitation Every learning agent in RL faces the fundamental trade-off between exploration and exploitation [52]. This dilemma arises from the need, on the one hand, to engage in *exploration* in order to discover new, potentially better strategies by visiting unknown states and actions. On the other hand, *exploitation* aims to leverage the knowledge already acquired by applying the strategy currently believed to be optimal in order to maximize reward. Striking the right balance between these opposing objectives is crucial for the efficiency of the learning process. An agent that exploits exclusively risks getting stuck in a local optimum, whereas an agent that only explores will never benefit from the knowledge it has accumulated.

Core Distinctions in RL RL [52] can be characterized along two central dimensions. One concerns the optimization target: some methods approximate a value function (value-based), while others directly learn a policy distribution (policy-based). The other relates to the way experience is collected and reused, contrasting on-policy with off-policy learning.

Value-Based vs. Policy-Based Methods The first dimension focuses on the optimization target. *Value-based* methods focus on learning the value of states or actions. They approximate a value function (e.g. a Q-function), from which the policy is derived implicitly by selecting the action with the highest value. In contrast, *policy-based* methods learn the policy directly, i.e., a mapping from states to actions or action probabilities, without the need to explicitly estimate a value function.

On-Policy vs. Off-Policy Learning The second dimension addresses the use of experience. *On-policy* methods improve their policy solely from data generated by the most recent version of that same policy — essentially, they "learn while doing." *Off-policy* methods, by contrast, can also learn from earlier experiences produced under

different policies. Such experiences are often stored in a *replay buffer*, which substantially improves sample efficiency.

2.4.2 Imitation Learning (IL)

Imitation learning [24] is a paradigm of machine learning in which an agent acquires a task not through trial and error as in RL, but by imitating expert behavior. The core idea is to derive a policy from a dataset of demonstrations provided by an expert. These demonstrations typically consist of state-action pairs (s, a) that record which action the expert performed in a given state.

The main advantage of IL is that it eliminates the need to design an explicit reward function, which is often one of the greatest challenges in RL. Instead, it is assumed that the expert's behavior implicitly represents an optimal, or at least highly effective, policy. IL is therefore particularly advantageous when the desired task is easy to demonstrate but difficult to formalize through a reward function alone, as in autonomous driving or robotic manipulation.

Behavioral cloning (BC) [41] is the most direct and straightforward form of IL. The approach reduces the learning problem to standard *supervised learning*. The objective is to learn a policy $\pi_\theta(s)$ with parameters θ that directly maps states to actions [43]:

$$\pi_\theta : S \to A \tag{2}$$

A model, typically a neural network, is trained to predict the expert's action a for a given state s from the demonstration dataset. The training process minimizes a loss function that measures the deviation between the predicted action and the expert action. For continuous actions, the mean squared error is commonly used, while for discrete actions cross-entropy loss is applied.

Despite its simplicity, BC suffers from a critical conceptual weakness known as *covariate shift* or *distribution mismatch* [46]. The model is trained only on states visited by the expert. If the agent makes a small mistake during execution, it may enter a state outside the training distribution. In such unfamiliar situations, the model may output unpredictable actions, leading to further errors. These errors can cascade, gradually driving the agent away from the expert's trajectory and often resulting in complete task failure.

To mitigate the limitations of pure BC, in particular covariate shift, while still benefiting from the high sample efficiency of expert-guided learning, a *hybrid approach* has become common in practice [38].

This approach begins with IL — typically BC — applied during a *pre-training* phase. The goal here is not to obtain a perfect final policy, but to initialize the network weights

in a meaningful way. By imitating expert demonstrations, the network learns from the outset to construct useful state representations and to propose actions that already exhibit basic competence.

From this "warm start", a second phase of *fine-tuning* is performed using a RL algorithm. In this phase, the agent can improve its behavior by explicitly exploring the environment, learning from its own mistakes, and potentially discovering strategies that surpass those of the expert. The hybrid approach thus combines the fast initial convergence of IL with the robustness and self-improvement capabilities of RL.

2.5 Heuristic Planning

For many complex planning and optimization problems, particularly those classified as NP-hard, finding a guaranteed optimal solution is computationally infeasible. This is where *heuristic planning* comes into play [47]. A heuristic is a problem-solving strategy that aims to find a "good enough" solution within reasonable time, rather than systematically searching the entire solution space. Heuristic planners exploit problem-specific knowledge in the form of a heuristic function to evaluate promising actions or paths and thereby guide the search process. An overarching class of such methods is *metaheuristics*, which provide general algorithmic frameworks that can be adapted to a wide range of optimization problems.

2.5.1 Ant Colony Optimization

Ant Colony Optimization (ACO) [15] is a nature-inspired, swarm-based metaheuristic that is modeled on the foraging behavior of ant colonies. It has been observed that ants are able to efficiently discover the shortest path between their nest and a food source, a process governed by a form of indirect communication known as stigmergy.

The mechanism operates as follows: as ants move, they deposit a chemical trail called pheromone along their path. Subsequent ants tend to follow routes with higher pheromone concentration. Since ants traveling shorter paths return more quickly between nest and food source, pheromone accumulates more rapidly on those paths. At the same time, pheromone gradually evaporates, reducing the attractiveness of longer, less frequently used routes. This positive feedback loop enables the colony as a whole to converge on the shortest path, which emerges as the most strongly reinforced route. In ACO, this principle is simulated with artificial "ants" (agents) to solve optimization problems.

2.5.2 Application to the Traveling Salesman Problem

One of the most prominent applications of ACO is the *Traveling Salesman Problem (TSP)* [15], a classical combinatorial optimization problem. The task is to find the

shortest possible tour through a given set of locations (cities) such that each city is visited exactly once and the tour returns to the starting point.

In applying ACO to the TSP, cities are represented as nodes of a graph and the direct connections between them as edges [14]. Pheromone trails are deposited on these edges. An artificial ant starts from a randomly chosen city and selects the next city to visit according to a probabilistic rule that accounts for two factors:

1. the *pheromone concentration* on the edge leading to the next city (stronger trails are more attractive), and

2. a *heuristic value*, typically the inverse of the distance to the next city (shorter edges are more attractive).

Once an ant has completed a full tour, the pheromone levels on the edges it traversed are reinforced, with the update being proportional to the quality (i.e., length) of the tour. At the same time, pheromone trails undergo evaporation. Over many iterations, this process strengthens the edges that belong to the best global solutions, leading the colony to converge toward an optimal or near-optimal solution of the TSP.

3 Related Work

Research in the field of embodied artificial intelligence (Embodied AI) has progressed from simple reactive systems to agents capable of semantically understanding their environment and acting purposefully within it. This section reviews the state of the art relevant to the task of semantics-driven exploration addressed in this thesis. The presentation follows a thematic progression: from early navigation paradigms, through structured semantic representations, to architectural and methodological developments that collectively define the research gap targeted by this work.

3.1 Early Navigation Strategies

Early approaches to autonomous navigation laid the foundation for more complex, semantically enriched methods. These can be roughly divided into *geometric exploration*, which ignores semantic content, and *semantic exploration*, which leverages contextual cues. Alongside this distinction, another line of research developed around end-to-end learning approaches.

Geometric vs. Semantic Exploration The first exploration strategies focused solely on the geometric mapping of the environment. The most prominent example is *frontier-based exploration*, where an agent repeatedly targets the boundary between known and unknown space in order to expand its map [61]. The key limitation of these methods lies in their semantic blindness: the agent merely maximizes the explored area without considering the content or relevance of the discovered regions. In response, later approaches began to exploit semantic information to improve efficiency, particularly in *active object search* [16, 62]. Here, the agent leverages contextual knowledge (e.g. "a television is typically found in a living room") to guide and accelerate the search for a specific object. However, such methods usually have the goal of locating a single target rather than building a comprehensive, holistic understanding of the entire scene.

End-to-End Learning with Recurrent Architectures In parallel, *end-to-end learning approaches* [11, 10] emerged that learn a navigation policy directly from sensory input. The dominant architecture of this era combined a *Convolutional Neural Network (CNN)* for visual feature extraction with a *Recurrent Neural Network (RNN)*, most often a *Long Short-Term Memory (LSTM) network*, to process the temporal sequence of observations. Although these models can acquire complex behaviors, they are known to struggle with accurately maintaining information over long trajectories [7].

Despite these advances, key challenges remain — in particular, the efficient storage and processing of long-term contextual information and the explicit modeling of semantic structures. This has led to a paradigm shift away from purely sequential

architectures toward explicit memory structures and semantic representations that can more directly guide navigation behavior.

3.2 Structured Representations for Semantic Exploration

This paradigm shift manifests itself in the use of explicit, structured representations of the environment that encode both geometric and semantic information [63, 34, 45, 42].

Geometric Memory: Neural SLAM A key approach to overcoming the memory limitations of purely recurrent architectures is *Neural SLAM*, introduced by Zhang et al. [63]. Rather than relying on an unstructured hidden state, this method equips the agent with an explicit external 2D memory map that serves as an internal representation of the environment. The core innovation lies in integrating traditional SLAM (Simultaneous Localization and Mapping) procedures — such as localization and motion prediction on the map — as differentiable operations directly into the neural architecture. This enables the network to build a stable internal representation of the environment, allowing for more targeted and efficient exploration compared to models that must implicitly infer spatial structure and localization from raw observations.

Semantic Memory: Scene Graph Representations An even richer representation is provided by *semantic scene graphs*, which capture not only objects but also their attributes and semantic relationships in a graph structure. Such graphs serve as a structured knowledge base that can be leveraged by navigation agents to reason about both spatial arrangements and semantic relations. The methodology for constructing these graphs varies across approaches. For example, Oskolkov et al. [42] employ CLIP, a model developed by OpenAI, to extract semantic representations from images. These embeddings are then organized into a graph encoding both spatial layout and semantic relations between objects in the environment.

A more classical approach to generating scene graphs was presented by Li et al. [34]. Their method employs a CNN to extract visual features, constructs a 3D semantic point cloud, and transforms it into a semantic graph. Training is performed using a hybrid approach that combines imitation learning with reinforcement learning, the latter implemented via the REINFORCE algorithm [57]. Most importantly, this work formally introduced the task of *Embodied Semantic Scene Graph Generation (ESSGG)*, which defines the agent's objective as building a comprehensive and accurate semantic scene graph of its environment, rather than merely reaching a specific target. The approach by Li et al. thus provides the baseline for the present thesis and serves as the starting point for investigating how more modern architectures and learning paradigms can be integrated.

3.3 Transformer Architectures and Modern Learning Algorithms

Since the publication of Li et al. [34], the field has advanced considerably, especially in the development of neural architectures. Moreover, variance-reduced policy gradient methods have been developed as alternatives to *REINFORCE*, such as *Advantage Actor-Critic (A2C)*, which leverages an advantage estimator and is often combined with entropy regularization for more stable and efficient learning [36].

The limitations of LSTMs in capturing long-term dependencies have led to the increasing adoption of Transformer architectures in navigation tasks. Models such as *NaviFormer* [60] employ specialized encoder-decoder Transformers to contextualize spatial, temporal, and exploratory cues far more effectively than generic LSTMs. Similarly, multimodal Transformers such as *EmBERT* [50] demonstrate advantages in fusing diverse input streams (vision, language, actions) within a unified model. In the context of navigation, Transformers have been applied not only to visual and linguistic modalities, but also to structured representations such as graphs. For instance, HRON (Hierarchical Relational Object Navigation) [35] employs a Heterogeneous Graph Transformer to natively integrate the topological structure of scene graphs into the navigation policy. This enables more expressive relational reasoning about the environment by propagating information along the graph edges.

3.4 Synthesis and Research Gap

The current state of research highlights (i) a clear shift toward Transformer-based architectures for handling long-term contextual dependencies and (ii) the adoption of more robust policy-gradient methods such as A2C to reduce learning variance.

At the same time, the ESSGG task defined by Li et al. [34] has not yet been systematically examined under these two paradigms. Specific shortcomings of the Li baseline include:

- **Sequence modeling**
  LSTM-based policies struggle to reliably preserve context over long trajectories.
- **Learning variance**
  REINFORCE is highly susceptible to variance and requires a large number of samples to produce stable updates.

This thesis addresses the gap through a targeted ablation study in which the architectural (LSTM → Transformer) and algorithmic (REINFORCE → A2C) modifications are evaluated both in isolation and in combination. The aim is to clearly attribute performance gains to their respective sources.

Concretely, a novel navigation module is designed that (i) employs a Transformer to process the exploration history and (ii) adopts A2C as the learning algorithm.

4 The Ai2-THOR Environment

The following section provides an overview of the Ai2-THOR [4, 3] simulation environment used throughout this thesis. The structure and main functionalities of the iTHOR module are introduced, with a focus on simulation principles, the agent-simulator interaction paradigm, available actions, observation modalities, and the organization of event data and metadata. Features of particular relevance for this thesis are highlighted.

4.1 Environment Overview

iTHOR is a module within the Ai2-THOR framework, developed by the Allen Institute for AI, which provides a realistic, interactive 3D environment for research in visual artificial intelligence. With 120 modeled scenes — including kitchens, living rooms, bedrooms, and bathrooms — iTHOR enables agents to navigate complex spaces and interact with objects within them. The scenes in iTHOR comprise over 2000 unique elements, many of which are interactive and can assume various states, such as opened/closed or toggled on/off. Agents can perform actions such as `PickupObject`, `OpenObject`, or `ToggleObjectOn` to interact with these objects.

Because iTHOR is built on the Unity 3D engine [54] — which allows for physically realistic simulations — objects in iTHOR possess physical properties such as mass, friction, and elasticity. The Python API for Ai2-THOR enables programmatic control of agents and their interactions with the environment.

Setting up iTHOR is straightforward due to the comprehensive Python API and detailed documentation. Installation is performed via the Python package manager `pip`:

```
pip install ai2thor
```

Ai2-THOR is currently supported on macOS (version 10.9 and above) and Ubuntu Linux (version 14.04 and above). For Linux users, an X server with GLX support is required [3].

A minimal Python example for initializing the environment and executing a simple action is shown below:

```
from ai2thor.controller import Controller
controller = Controller(scene="FloorPlan1")
event = controller.step(action="MoveAhead")
```

4.2 Functionality

Interaction with the Ai2-THOR environment follows an agent-simulator paradigm (see Figure 13): A Python agent sends actions to the Unity-based simulator. After each action, the agent receives an `Event` object, which contains all observation data and scene metadata [3].

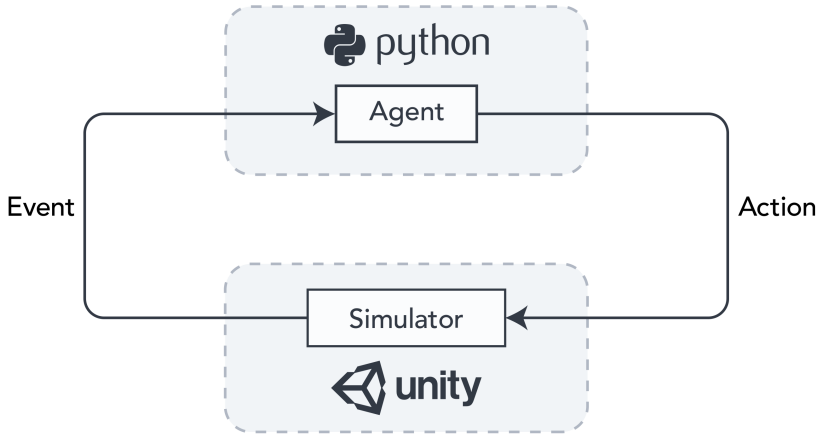

Figure 13: Communication between Python agent and Unity simulator in Ai2-THOR [3]

4.2.1 Available Actions

The agent in Ai2-THOR can execute a wide range of actions. A selection of possible actions that are used in this thesis is summarized in Table 1, grouped into movement, rotation, and special actions.

Movement	Rotation	Special Actions
MoveAhead MoveBack MoveLeft MoveRight	RotateLeft RotateRight	GetReachablePositions GetShortestPathToPoint Teleport Pass

Table 1: Ai2-THOR actions used in this thesis

The actions `MoveAhead`, `MoveBack`, `MoveLeft`, and `MoveRight` move the agent by 0.25 m in the specified direction. The rotation actions `RotateLeft` and `RotateRight` rotate the agent by 90 degrees. The `Teleport` action moves the agent to a specified position within the scene. `GetReachablePositions` returns a list of all positions reachable by the agent in the current scene while `GetShortestPathToPoint` computes the shortest path to a specified point, returning the corner points of the path. The `Pass` action does not change the agent's state, but forces the scene to be rendered; this can be used to update observations without movement.

In addition to these, Ai2-THOR provides a broad range of further interaction and manipulation actions.

4.2.2 Event Object and Metadata

After executing any action, the simulator returns an `Event` object with data describing the current scene state. The main components of an `Event` object include:

- `metadata`: Detailed information on the current scene state, including all objects, agent position, scene name, and more.
- `frame`: The current RGB image from the agent's point of view (2D color image of the scene).
- `cv2img`: An OpenCV-compatible representation of the RGB image.
- `depth_frame`: Optional depth image representing the distance from the agent to each pixel in the scene.
- `instance_segmentation_frame`: Optional segmentation image, where each object is represented by a unique color.
- `instance_masks`: A dictionary containing a binary mask (2D) for each recognized object.
- `instance_detections2D`: A dictionary with 2D bounding boxes for all visible objects in the scene.
- `color_to_object_id`: Mapping between segmentation colors and object IDs.
- `object_id_to_color`: Inverse mapping from object ID to the corresponding segmentation color.

4.2.3 Relevant Metadata

The metadata field in the event object contains several key items:

- `errorMessage`, `lastActionSuccess`, `lastAction`: Status of the last action executed.
- `sceneName`, `sceneBounds`: Scene identifier and scene dimensions.
- `agent`: Status of the agent (position, rotation, camera horizon).
- `objects`: List of all known objects in the scene (with attributes).

For each object, the following attributes are particularly relevant:

- `objectId`: Unique object identifier.
- `objectType`: Object type (e.g. “Mug”, “Microwave”).
- `position`: Object position within the scene.
- `visible`: Boolean indicating if the object is currently visible from the agent’s perspective.
- Additional state attributes (e.g. “open”, “pickupable”, “toggleable”, ...).

An example structure of a metadata block is shown in Figure 14.

```
{
    "errorMessage": null,
    "lastActionSuccess": true,
    "actionReturn": {...},
    "lastAction": "MoveAhead",

    "sceneName": "FloorPlan1",
    "sceneBounds": {...},

    "agent": {
        "position": {"x": ..., "y": ..., "z":
            ...},
        "rotation": {"x": ..., "y": ..., "z":
            ...},
        "cameraHorizon": ...
    },
    "objects": [
        {
            "objectId": "Mug
                |+01.00|+01.22|+02.15",
            "objectType": "Mug",
            "position": {"x": 1.0, "y": 1.2, "z":
                2.15},
            "visible": true,
            ...
        },
        ...
    ],

    "fov": 60,
    "screenWidth": 300,
    "screenHeight": 300,
    {...}
}
```

Figure 14: Exemplary excerpt from the `metadata` block of an Ai2-THOR event

4.3 Environment Wrapper and Observation Space

A custom wrapper (`thor_env`) was implemented around Ai2-THOR [31]. The design follows the Gymnasium interface to provide a familiar `reset` and `step` API [53]. Each

call to `reset` or `step` returns an observation object containing the current state, the received reward, termination flags, and additional diagnostic info. The state comprises an RGB image from the agent's perspective, which is provided by the `frame` of the returned `Event` object, a local scene graph (LSSG) of visible objects and relations and a global scene graph (GSSG) aggregated over the episode, which are constructed from the `Event` object's metadata, and the last action executed.

Each environment action is a tuple of two basic commands executed in sequence: a movement (`MoveAhead`, `MoveLeft`, `MoveRight`, `MoveBack`) and a rotation (`RotateLeft`, `RotateRight`). For each basic command, the special action `Pass` can be chosen to leave that dimension unchanged. If both commands are `Pass`, this is interpreted as a `Stop` signal that terminates the episode.

4.4 Pre-computed Environment

Within the scope of this thesis, a data-driven implementation of the custom `thor_env` wrapper was developed to reduce the computational cost during training and to decouple the experiments from the need for a running Unity simulation.

The foundation of this implementation is a pre-computed database, implemented as a lookup table. For this purpose, in a one-time initialization step, all points reachable by an agent within a scene are determined using the `GetReachablePositions` action. For each of these discrete states — defined by the agent's position and rotation — the complete event data, as described in Section 4.2.2, is retrieved and stored with the state serving as a unique key.

The agent's interaction with the environment thereby becomes a series of queries to this database. When the agent executes an action such as `MoveAhead`, the theoretical target state is calculated. It is then checked whether this new state key exists in the lookup table. If it does, the wrapper returns the pre-stored data for that state. If the key does not exist, the action is considered invalid, which corresponds to an invalid move, such as a collision with a wall. In this scenario, the agent maintains its previous state, and the data from the last valid state is returned again.

5 Learning Algorithms

In this section, the learning algorithms used to control the agent are described. As a reference method, we employ the classical Monte Carlo[2] algorithm *REINFORCE*, while the more advanced *Advantage Actor-Critic (A2C)* algorithm is adopted as a modern extension. Both approaches are based on the policy-gradient method, but they differ substantially in the nature of their learning signals and in their mechanisms for variance reduction.

5.1 REINFORCE

In contrast to value-based methods, which learn a value function and derive a policy indirectly, *policy-gradient methods* [52] operate directly in the space of policies. They parameterize the policy π itself and optimize its parameters θ via gradient ascent. The *REINFORCE algorithm* [57], also known as the Monte Carlo policy gradient, is a fundamental representative of this class.

Formally, the policy is represented as a function $\pi_\theta(a \mid s)$, which denotes the probability of selecting action a in state s given parameters θ:

$$\pi_\theta(a \mid s) = P(A_t = a \mid S_t = s; \theta) \tag{3}$$

The objective is to optimize θ in order to maximize a performance measure $J(\theta)$, typically defined as the expected return starting from an initial state s_0 [52]:

$$J(\theta) = \mathbb{E}_{\pi_\theta}[G_t] = \mathbb{E}_{\pi_\theta}\left[\sum_{k=0}^{T} \gamma^k R_{t+k+1}\right] \tag{4}$$

where G_t denotes the total discounted return from time step t onward.

The core of REINFORCE relies on the *policy gradient theorem* [52], which provides an analytical expression for the gradient of the performance objective:

$$\nabla_\theta J(\theta) = \mathbb{E}_{\pi_\theta}\Big[G_t \, \nabla_\theta \ln \pi_\theta(A_t \mid S_t)\Big] \tag{5}$$

Here, $\nabla_\theta \ln \pi_\theta(a \mid s)$ is referred to as the *score function*. The intuition is that the parameters θ are updated in a direction that increases the probability of actions A_t proportional to the returns they produced. Actions that yielded high return are reinforced, whereas those leading to poor outcomes are suppressed.

Since REINFORCE is a Monte Carlo method, the expectation in Equation 5 is approximated using samples from complete episodes. After each episode, the parameter

[2] Monte Carlo methods estimate expectations (e.g. discounted episode returns) by averaging over complete, independent episodes. They avoid intermediate bootstrapping estimates and instead use the final returns of each episode directly to update value functions or policy gradients. [52]

update for every time step t is performed according to the original formulation in [57]:

$$\theta \leftarrow \theta + \alpha\, G_t\, \nabla_\theta \ln \pi_\theta(A_t \mid S_t) \tag{6}$$

where α denotes the learning rate.

A well-known issue of the plain REINFORCE algorithm is the high variance of its gradient estimates [52]. Variance here refers to the spread of the estimates around their expectation, which can lead to unstable learning. It arises because the return G_t is accumulated over an entire episode and is strongly influenced by stochastic actions and state transitions. To reduce this variance without altering the expected value of the gradient — that is, without introducing bias — it is common to subtract a state-dependent *baseline* $b(S_t)$ from the return. A baseline acts as a reference value that reduces variance while preserving the unbiasedness of the gradient estimate:

$$\nabla_\theta J(\theta) = \mathbb{E}_{\pi_\theta}\Big[\big(G_t - b(S_t)\big)\, \nabla_\theta \ln \pi_\theta(A_t \mid S_t)\Big] \tag{7}$$

A typical choice for the baseline is an estimate of the state-value function, $b(S_t) \approx v_{\pi_\theta}(S_t)$, which naturally leads to more advanced methods such as *actor-critic algorithms* [52], discussed in the following section.

As a pragmatic alternative that avoids explicit value-function learning, many implementations use *return centering and scaling*. Here, the empirical mean of the episodic returns within a batch serves as a constant baseline $b = \hat{\mu}_G$, and the centered returns are subsequently scaled by their standard deviation. This reduces variance while maintaining the expectation of the policy gradient (baseline property) [52]. The update rule becomes:

$$\theta \leftarrow \theta + \alpha \sum_{i=1}^{N} \sum_{t=0}^{T_i} \left(\frac{G_{i,t} - \hat{\mu}_G}{\hat{\sigma}_G + \epsilon} \right) \nabla_\theta \ln \pi_\theta(a_{i,t} \mid s_{i,t}) \tag{8}$$

where $\hat{\mu}_G$ and $\hat{\sigma}_G$ are the mean and standard deviation of returns within the batch, N is the number of episodes in the batch, T_i the length of episode i and ϵ is a small constant to ensure numerical stability.

Finally, to guarantee sufficient exploration of the environment, actions are sampled stochastically from the probability distribution output by the policy $\pi_\theta(a \mid s)$, rather than deterministically selecting the most likely action.

5.2 Actor-Critic Methods

Actor-Critic methods [52] represent a hybrid architecture that combines the advantages of policy-gradient and value-based approaches. They were developed to address the high variance of gradient estimates in REINFORCE by replacing the Monte Carlo return G_t with a learnable, step-wise critic. The architecture consists of two interacting components, typically implemented as separate neural networks:

- The *actor* is responsible for action selection. It represents the policy to be learned and is optimized using a policy-gradient approach. Its role is to adjust the parameters θ of the policy $\pi_\theta(a \mid s)$ in order to maximize the expected return.
- The *critic* evaluates the actions chosen by the actor. It learns a value function, typically the state-value function $V_w(s)$ parameterized by w. Instead of proposing actions itself, the critic "critiques" the actor's decisions by providing a better estimate of the return than a full Monte Carlo sample could.

Learning proceeds in an online, step-by-step fashion. After each action A_t taken in state S_t, with received reward R_{t+1} and next state S_{t+1}, the critic computes the *temporal-difference (TD) error* δ_t:

$$\delta_t = R_{t+1} + \gamma V_w(S_{t+1}) - V_w(S_t). \tag{9}$$

This TD error serves as a lower-variance learning signal for both components. It quantifies how much the critic's estimate for state S_t deviates from the improved estimate $R_{t+1} + \gamma V_w(S_{t+1})$, which is known as the *bootstrap estimate* because it updates one value estimate (for S_t) using a subsequent value estimate (for S_{t+1}).

The parameters of the two components are updated using this signal:

1. **Critic update**
   The critic minimizes the TD error, for example via gradient descent on the mean squared error, in order to improve its value estimates:

   $$w \leftarrow w + \beta\, \delta_t \nabla_w V_w(S_t) \tag{10}$$

   where β is the critic's learning rate.

2. **Actor update**
   The actor uses the TD error δ_t as an advantage estimate to update its policy parameters. The Monte Carlo return G_t from the REINFORCE algorithm is thus replaced by the TD error:

   $$\theta \leftarrow \theta + \alpha\, \delta_t \nabla_\theta \ln \pi_\theta(A_t \mid S_t) \tag{11}$$

   where α is the actor's learning rate.

Through this mechanism, the algorithm can learn online while benefiting from the substantially reduced variance of the learning signal, which typically results in more stable and often faster convergence compared to REINFORCE.

A direct extension of this architecture is the *Advantage Actor-Critic (A2C)* algorithm [36], which is used in this thesis. The central idea is that the actor should

not only be informed whether an action was "good" (positive TD error), but also "how much better" it was compared to the expected baseline. This is formalized via the *advantage function* $A(s,a) = Q(s,a) - V(s)$, where $Q(s,a)$ is the *action-value function* that estimates the expected return of taking a specific action a in a state s, while $V(s)$ represents the average value of being in that state. In practice, the TD error δ_t already provides a useful approximation of the advantage. A2C further stabilizes training by often aggregating over multiple time steps (n-step returns) to obtain more robust advantage estimates.

As in REINFORCE, exploration is maintained by sampling actions from the probability distribution $\pi_\theta(a \mid s)$ learned by the actor. To further encourage exploration and prevent premature convergence to a suboptimal deterministic strategy, an entropy bonus is added to the actor's loss [36]. The *entropy* of the policy, $\mathcal{H}(\pi_\theta(\cdot \mid S_t))$, measures its randomness, and maximizing it rewards higher action diversity.

The final loss function minimized in A2C is therefore a weighted sum of three components: the actor's policy loss, the critic's value loss, and the entropy bonus:

$$\mathcal{L}(\theta) = \mathcal{L}_{\text{policy}}(\theta) + \beta_{\text{val}}\,\mathcal{L}_{\text{value}}(\theta) - \beta_{\text{ent}}\,\mathcal{H}(\pi_\theta) \tag{12}$$

where β_{val} and β_{ent} are weighting coefficients for the value loss and entropy regularization, respectively.

6 Methodology

This chapter presents the methodological foundation of this thesis, focusing on the development and evaluation of an autonomous agent to navigate in unknown 3D environments. The overall approach is first outlined in Section 6.1, situating it within the context of existing research. Building on this, Section 6.2 introduces the modular agent architecture, which consists of a multimodal feature encoder and a flexible navigation policy. The internal environment representation through semantic scene graphs is then described in Section 6.3, enabling a rich spatial understanding. The multi-stage training and optimization strategy, combining imitation learning (IL) with reinforcement learning (RL), is detailed in Section 6.4. Finally, Section 6.5 explains the design of the ablation study, which systematically investigates the contribution of individual components.

6.1 Overall Approach and Context

This thesis methodologically builds on the baseline established by Li et al. [34], but differs in key aspects with respect to scene representation, data foundation, action space, and the architecture for graph processing. Instead of being estimated from image data, the scene graphs are extracted directly from the ground-truth metadata of the Ai2-THOR environment, since their construction is not the focus of this work. In this way, (i) the node classes and positions as well as (ii) the relations (edges) are, by definition, correct and consistent. Consequently, the evaluation does not differentiate on edge quality; the decisive factor is how efficiently the agent brings previously unobserved objects into view, thereby increasing the Node Recall. The main evaluation metrics are *Node Recall* and *Mean Steps*; a separate edge evaluation is omitted by design.

In contrast to Li et al. [34], training in this work is restricted exclusively to ten kitchen scenarios and employs a reduced action space with only four instead of eight movement directions. Furthermore, the original mechanism for updating the global scene graph is replaced by a *Heterogeneous Graph Transformer (HGT)*. Unlike in Li et al., where graph Transformers were applied only to local scene graphs, this thesis extends their use to both local and global representations. Potential limitations that may arise from these design choices, along with further ones, are discussed in Section 7.5.

The primary objective of this methodology is the systematic improvement of the navigation module. Specifically, the study investigates whether replacing *Long Short-Term Memory (LSTM)* based modules with *Transformer*-based architectures and adopting the *Advantage Actor-Critic (A2C)* algorithm instead of the previously used *REINFORCE* method can lead to enhanced navigation performance. These investigations define the core innovations of this work: the integration of modern Transformer models and advanced RL algorithms into an established multimodal navigation framework.

The methodological workflow is structured into five main steps:

1. Generation of a ground-truth scene graph from environment metadata.
2. Construction of an expert dataset for IL.
3. Pretraining of the feature encoder via IL, using both LSTM- and Transformer-based modules.
4. Separate hyperparameter optimization for the REINFORCE and A2C algorithms.
5. Final training and fine-tuning using RL, conducted in the form of an ablation study to systematically evaluate the individual components.

To ensure fair comparability with existing methods, a dedicated baseline was implemented that closely follows the original approach (LSTM + REINFORCE) of Li et al. [34]. This baseline serves as a reference point against which the impact of the architectural and algorithmic modifications can be systematically assessed.

6.2 Agent Architecture

The agent architecture follows a modular design and, as illustrated in Figure 15, comprises two main components: a *feature encoder* and a *navigation policy*.

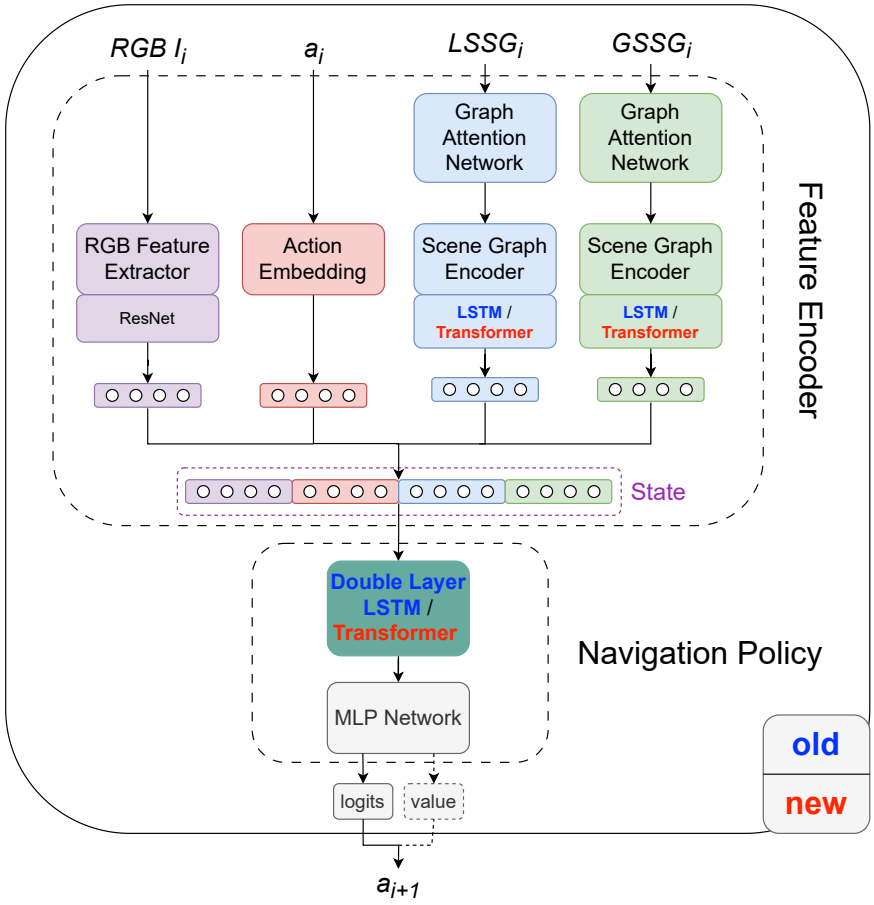

Figure 15: Modular architecture of the agent, consisting of the feature encoder and the navigation policy

The *feature encoder* is implemented as a multimodal module that processes the diverse information available from the environment. It integrates visual input from the RGB camera image (I_i), historical information in the form of the last executed action (a_i), and semantic input from both the local ($LSSG_i$) and the global scene graph ($GSSG_i$). These inputs are processed in parallel and combined into a comprehensive state vector.

The *navigation policy* constitutes the decision-making module of the agent. It receives the state vector from the feature encoder and determines the next action to be executed (a_{i+1}). At its core lies a sequential network, implemented either as a *double-layer LSTM* or a *Transformer*, whose output is passed to a *multi-layer perceptron (MLP)*. The detailed design of the agent model is described in the following sections.

6.2.1 Multi-Modal Feature Encoder

The feature encoder serves as the perception module of the agent. Its purpose is to transform the raw multimodal sensory inputs together with the internal state (scene graphs) into a fixed-size feature vector that represents the complete agent state for decision-making. As shown in Figure 16, the encoder processes the inputs through four parallel streams that can be grouped into three categories: visual perception (purple), action history (red), and semantic graph perception (blue & green).

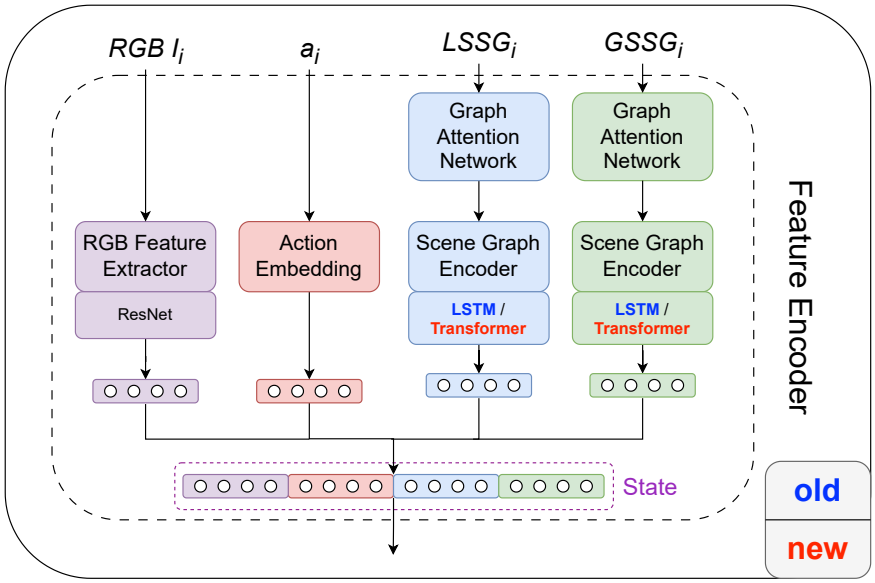

Figure 16: Modular design of the feature encoder

Visual Perception The agent's visual perception is based on a *ResNet18* model pre-trained on the ImageNet dataset. In the proposed setup, the original classification head of the network is removed and replaced by a linear projection layer. The resulting model extracts from each RGB camera image (I_i) a feature vector of dimension **512**. The RGB inputs are preprocessed by resizing them to 224×224 and normalizing them using the ImageNet statistics[3]. The ResNet18 backbone is fine-tuned together with the newly added prediction head, rather than being kept frozen.

Action History To provide the agent with a short-term memory of its most recent action, the last executed action a_{i-1} is represented as a discrete index. This index is passed through a learnable *embedding layer* and mapped to a vector of dimension **32**.

Semantic Graph Perception Semantic graph perception is handled through two independent processing streams: one for the local scene graph (LSSG) and one for the

[3]The term *ImageNet statistics* refers to the channel-wise normalization values, i.e., the per-channel means and standard deviations computed on the ImageNet-1K training set.

global scene graph (GSSG). Although both streams share the same architecture, they are parameterized by distinct, independently trained weights. Feature extraction for each graph proceeds in four stages:

1. **Initialization of node features** Graph nodes, representing objects in the scene, are initialized with features composed of their 3D position, a visibility flag, and a unique identifier for the object type (e.g. "Mug" or "Table").

2. **Graph representation learning** A *Heterogeneous Graph Transformer (HGT)* is applied to model the relationships between nodes. The HGT outputs contextualized vectors of dimension **256** for each node and edge, i.e., semantic information is projected into a 256-dimensional vector space.

3. **Graph pooling and concatenation** A final graph representation is obtained in two steps. First, an *attention pooling mechanism* aggregates all node vectors into a single 256-dimensional "node summary" vector. Similarly, a second mechanism aggregates all edge vectors into a 256-dimensional "edge summary" vector. These two vectors are then concatenated to yield the complete graph representation of dimension **512**.

4. **Sequential processing** The graph vectors produced at each timestep are assembled into a sequence. This sequence is passed to a sequential model (*LSTM* or *Transformer*) that captures temporal dependencies across graph states.

State Representation The final state vector, which serves as input to the navigation policy, is formed by concatenating the outputs of the four processing streams. It consists of the visual feature vector (512-D), the action embedding (32-D), and the two graph representations for LSSG (512-D) and GSSG (512-D), resulting in a total state dimensionality of **1568**.

6.2.2 Navigation Policy

The *navigation policy* is the agent's decision-making module. It processes the state vector provided by the feature encoder to produce a probability distribution over the action space, from which the next action a_{t+1} is sampled. At its core lies a sequential network that models temporal dependencies across the sequence of state vectors. Within the ablation study, two alternative architectures are evaluated:

- A *double-layer LSTM*, a well-established approach for modeling sequential data.

- A *Transformer encoder* based on the self-attention mechanism. To ensure a fair comparison with the LSTM variant, a linear projection is applied to match the hidden dimensionality of both architectures. This guarantees identical input

dimensions for the sequential module, thereby isolating architectural effects and enabling unbiased performance comparisons.

The output of the sequential module is passed to a shared *multi-layer perceptron (MLP)* (see Figure 17). The MLP comprises two blocks, each consisting of a linear layer, layer normalization, and a ReLU activation. Its output is then split into two dedicated heads:

- The *policy head* computes the logits for the action distribution. It consists of a sequence of `Linear`, `LayerNorm`, `ReLU`, and a final `Linear` layer.
- The *value head* estimates the state value $V(s)$. Its structure mirrors that of the policy head and it is used exclusively in the A2C training algorithm to compute advantage values.

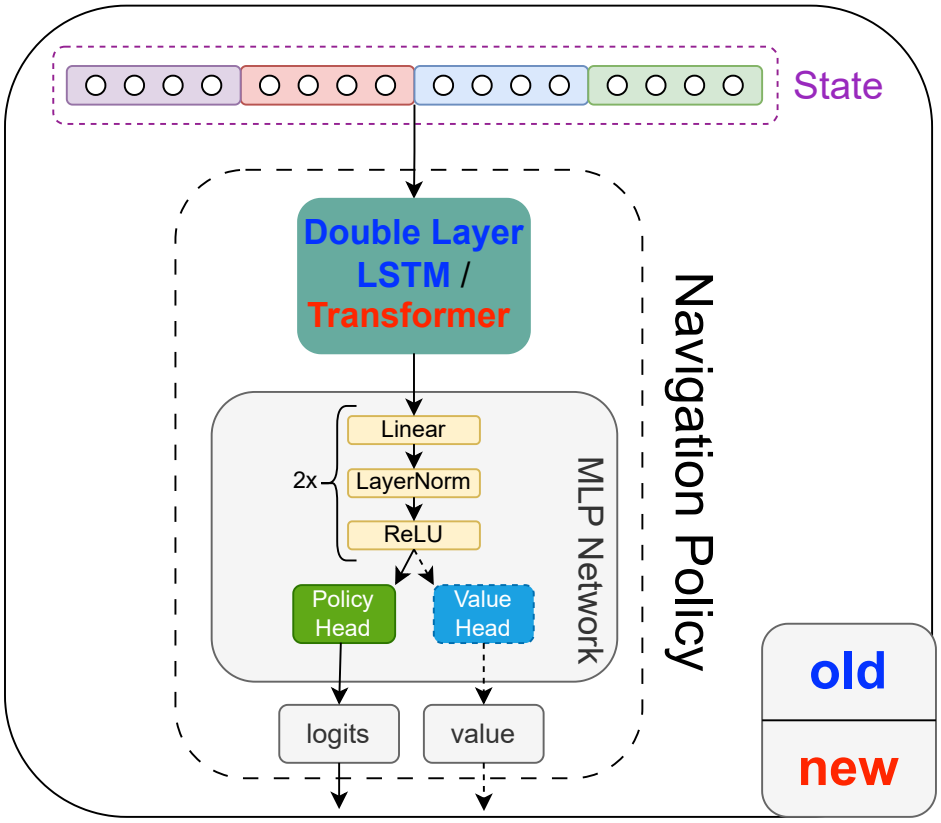

Figure 17: Architecture of the navigation policy

If the agent selects the action `Pass` for both movement and rotation, this is interpreted as a *Stop* signal that terminates the episode. In addition, each episode is forcibly terminated after at most 40 steps, consistent with the setup of Li et al. [34]. The detailed hyperparameters of the policy architecture are provided in Table 8 in the appendix.

6.3 Semantic Scene Representation

To provide a rich and structured representation of the environment that goes beyond mere object recognition, the scene is modeled as a semantic graph. This representation captures not only the objects themselves but also their spatial relations to one another. The agent's perception and memory are organized on two levels: a LSSG that encodes the immediately observable surroundings, and a GSSG that accumulates knowledge about the environment over the entire episode.

6.3.1 Local Semantic Scene Graph (LSSG)

The *LSSG* represents the agent's egocentric perception at a given time step t. It is extracted at every step directly from the ground-truth metadata of the Ai2-THOR environment, avoiding the need for potentially error-prone perception models such as object detectors, since their development is beyond the scope of this work. The graph's nodes correspond to the objects visible to the agent, with their positions likewise taken from the metadata. Edges represent spatial relations (e.g. "next to", "above") between objects, extracted analogously to the approach of Li et al. [34].

In the proposed approach, *soft visibility* is employed instead of a binary visibility indicator. Rather than assigning objects a hard visible/invisible flag, a continuous score is computed that simulates the confidence with which a hypothetical segmentation network would detect the object. This choice reflects the fact that scene graphs are derived directly from environment metadata rather than from a learned perception model, and it provides a means to evaluate the robustness of the system under potential perceptual uncertainty.

The soft visibility V_{soft} is defined using a sigmoid function whose center dynamically depends on the object's size, thereby modeling the fact that larger objects remain visible from greater distances:

$$V_{\text{soft}}(d, s_{\max}) = \frac{1}{1 + e^{\sigma(d - c(s_{\max}))}} \tag{13}$$

Here, d denotes the agent-object distance, σ controls the steepness of the curve, and $c(s_{\max})$ is the sigmoid's center, parameterized by the maximum object dimension $s_{\max}$:

$$c(s_{\max}) = c_{\text{base}} + k_{\text{size}}(s_{\max} - s_{\text{ref}}) \tag{14}$$

In this formulation, c_{base} is a reference distance, k_{size} a scaling factor for object size, and s_{ref} a reference size. For this work, the hyperparameters were set to $\sigma = 1.0$, $c_{\text{base}} = 3.5$, $k_{\text{size}} = 1.5$, and $s_{\text{ref}} = 0.5$, with distances and sizes expressed in meters. These values were selected through exploratory tuning and proved consistent with an intuitive notion of visibility for small versus large objects.

6.3.2 Global Semantic Scene Graph (GSSG)

The *GSSG* serves as the agent's persistent, allocentric memory. It aggregates information from all previously visited states into a coherent overall semantic model of the scene. At each time step, the information contained in the current LSSG is integrated into the GSSG.

In contrast to the baseline of Li et al. [34], which constructs the GSSG from a point cloud and updates features via an Exponential Moving Average, the proposed approach implements a mechanism of exact integration. Because Ai2-THOR provides unique and persistent `objectId`s for each object, items in the LSSG can be matched unambiguously with those in the GSSG. Newly observed objects are added, while the states of already known objects are updated precisely, without information loss due to averaging.

A particular focus lies on updating the *soft visibility*. When an object already present in the GSSG is observed again in the LSSG, its global soft visibility V_{GSSG} is updated according to the following probabilistic rule:

$$V_{GSSG,t+1} = 1 - (1 - V_{GSSG,t}) \cdot (1 - \alpha \cdot V_{LSSG,t+1}) \tag{15}$$

where $V_{GSSG,t}$ denotes the current global visibility, $V_{LSSG,t+1}$ the newly observed local visibility, and α a weighting factor (set to 0.9 in this work) that controls the influence of the new observation. A direct consequence of this rule is that the global visibility of an object is monotonically increasing; with each additional observation, the agent becomes more confident about the object's existence.

For higher-level control and goal definition, it is necessary to derive a discrete state from this continuous visibility score. In the proposed approach, a node in the GSSG is considered "seen" or "discovered" once its aggregated soft visibility V_{GSSG} exceeds a threshold of 0.8. This threshold was chosen empirically as a practical heuristic to distinguish between uncertain and reliable observations, without aiming for a theoretical justification.

6.4 Training and Optimization Strategy

The training and optimization of the agent follows a multi-stage methodology that synergistically combines Imitation Learning (IL) for robust feature extraction and Reinforcement Learning (RL) for policy optimization. This section outlines the complete pipeline. It begins with the offline generation of the expert dataset and the subsequent pretraining of the encoder. Afterward, it details the core components of the RL setup — specifically the reward function and its hyperparameter optimization — before concluding with the final training phase. The process encompasses the following key stages, each detailed in a corresponding subsection:

1. *Generation of the Expert Dataset* (Section 6.4.1), which provides the foundation for pretraining.

2. *Pretraining of the Feature Encoder via IL* (Section 6.4.2) to establish a robust feature representation.

3. *Definition of the Reward Function* (Section 6.4.3) and its *Hyperparameter Optimization* (Section 6.4.4) to guide the agent's learning.

4. *Final Training and Fine-Tuning via RL* (Section 6.4.5), where the agent improves through direct interaction.

The agent's performance is evaluated using the metrics detailed in Section 7.1.

6.4.1 Generation of the Expert Dataset

The foundation for IL is a dataset of expert demonstrations. This dataset is generated in a preparatory offline process across ten different kitchen scenarios. Therefore, an algorithmic expert is employed that can plan and execute meaningful exploration routes through the environment. Since such planning requires full knowledge of the scene, a ground-truth scene graph (GT-graph) is initially constructed. This is achieved by teleporting to every reachable point in the environment in order to record all objects, their positions, and their relations. For each visited viewpoint, the set of visible objects is stored along with their associated soft visibility scores.

The generation of an expert trajectory follows a three-stage planning process:

1. **Selection of target viewpoints (Viewpoint Cover)**
   First, the expert identifies a minimal set of viewpoints from which all objects in the scene can be observed with an aggregated soft visibility above 0.8. This is formulated as a set-cover problem and approximated with a greedy algorithm, which iteratively selects the viewpoint that provides the largest additional increase in soft visibility.

2. **Route optimization**
   Next, an efficient visiting order for these viewpoints is determined. This is modeled as a Traveling Salesperson Problem and solved using Ant Colony Optimization, yielding a near-optimal shortest path.

3. **Action generation**
   Finally, the expert traverses the optimized path by selecting the optimal action at each timestep. This selection relies on a lookahead strategy[4] over two future

[4]Lookahead strategy refers to simulating not only the immediate consequence of an action, but short sequences of actions (in this case of length two). Only the first action of the sequence that leads to the most promising future state is executed, after which the process is repeated at the next step.

steps to identify the action with the greatest long-term benefit. The procedure unfolds as follows:

- **Movement direction** — As a first step, the movement direction (e.g. `MoveAhead`) that brings the agent closer to the next target viewpoint is determined.
- **Sequence simulation** — Next, all possible two-step action sequences beginning with this movement are simulated. Each action is represented as a tuple of movement and rotation, encoded by a single action index.
- **Heuristic evaluation** — Each sequence is scored by a heuristic function H_{score}, which evaluates the resulting state:

$$H_{\text{score}} = S_{\text{vis}} + S_{\text{exp}} + S_{\text{rot}} \tag{16}$$

  In detail, the three components are defined as follows:

  - **Visibility Score (S_{vis})** The visibility score component quantifies the increase in soft visibility across all objects. In addition, it assigns a substantial bonus to any object that surpasses the "seen" threshold of 0.8 for the first time.
  - **Exploration Score (S_{exp})** The exploration score is defined as a fixed bonus awarded for entering a previously unvisited region of the environment.
  - **Rotation Score (S_{rot})** The rotation score constitutes a strongly weighted bonus (or penalty) that rewards correct orientation toward the target viewpoint. This component is applied only when the agent is sufficiently close to the respective viewpoint, ensuring that it not only reaches the correct location but also faces the appropriate direction.
- **Execution** — Finally, only the first primitive action of the sequence with the highest heuristic score is executed.

The result of this process is a collection of expert trajectories, each consisting of a sequence of state observations and the corresponding optimal actions. For each scene, ten trajectories are generated, resulting in a total of 100 trajectories across the dataset. This dataset provides the foundation for the subsequent IL training phase.

6.4.2 Training-Phase 1: Pretraining via IL

Building on the previously generated expert dataset, Phase 1 involves pretraining the agent using *IL*. The primary objective at this stage is not to learn a final navigation strategy, but to train the complex feature encoder to extract a rich and task-relevant

representation of the environment. This process, commonly referred to as *behavioral cloning* (see Section 2.4.2), provides a *warm start* for the subsequent fine-tuning with RL.

For this supervised learning task, the feature encoder is combined with the navigation policy into a complete `ImitationAgent`. The agent is trained to predict the expert's next action given a sequence of observed states.

The loss function used is *class-weighted cross-entropy*, where each class is assigned a weight inversely proportional to its frequency in the training dataset. In this way, actions that occur less frequently in the expert demonstrations receive a higher penalty when misclassified, ensuring that the model does not become biased toward the more common actions.

To process trajectories of varying length efficiently in batches, they are padded to a uniform length via a custom `collate` function. To ensure that the loss is computed only over valid (non-padded) time steps of each sequence, the `pack_padded_sequence` utility from `torch.nn.utils.rnn` is employed.

Training is performed over multiple epochs using the `Adam` optimizer and stabilized through gradient clipping,[5] with the dataset split into 85% for training and 15% for validation to select the best model.

As final weights for the feature encoder, the checkpoint of the `ImitationAgent` achieving the highest validation accuracy is stored. The weights of the navigation policy trained during this phase are discarded.

IL in this work serves exclusively as pretraining for the feature encoder; the policy is re-initialized at the beginning of RL training. Thus, no final behavioral strategy is transferred. Instead, the encoder acts as a stable warm start that typically accelerates convergence, though it may initially reduce exploration diversity.

6.4.3 Reward Function

A central element of the RL setup is the definition of the *reward function*, which provides the agent with numerical feedback to guide its behavior. The function is adapted from Li et al. [34] and is designed to encourage the agent to maximize recall of both nodes and edges in the scene graph, while also promoting diversity in the set of visited viewpoints. At the same time, a negative reward in the form of a *step penalty* is applied to incentivize efficiency, i.e., shorter trajectories.

Formally, the reward r_t at time step t is defined as the difference in a potential score S between the current and the previous state:

$$r_t = S_t - S_{t-1} \tag{17}$$

[5] Gradient clipping constrains the gradients to a predefined threshold (0.5 in this case) in order to prevent excessively large updates that could destabilize training.

The score S_t consists of a similarity component and additional terms for viewpoint diversity and step penalty. Specifically, the similarity at time step t is defined as:

$$\text{sim}_t = \lambda_{\text{node}}\Big(R_{\text{node},t} + \lambda_p\, P_{\text{node},t}\Big) + R_{\text{edge},t} + \lambda_p\, P_{\text{edge},t} \tag{18}$$

Here, $R_{\text{node},t}$ and $R_{\text{edge},t}$ denote node and edge recall, while $P_{\text{node},t}$ represents the mean visibility of nodes. Since relation extraction is perfect in this setup, edge precision $P_{\text{edge},t}$ is constant and equals 1; it does not affect the score difference and is included only for completeness. The overall score is then given by:

$$S_t = \text{sim}_t + \lambda_d \cdot \text{diversity}_t - \rho \cdot t \tag{19}$$

In this equation, diversity_t quantifies the spatial diversity of visited viewpoints, which will be defined in more detail below. The hyperparameters λ_{node}, λ_p, and λ_d weight the respective components and were set to the same values as in Li et al. [34]: 0.1, 0.5, and 0.001. The step penalty ρ is subject to optimization. Further details on the optimization process are provided in Section 6.4.4.

To encourage the agent to discover objects from as many different perspectives as possible, a *diversity* term rewards the variety of explored viewpoints. For this purpose, the agent's spatial domain is discretized into a grid of positions and orientations, referred to as *view-cells*. A new, reward-eligible perspective is registered whenever an object is observed from a view-cell from which it has not been seen before.

The diversity score (diversity_t) at time step t is thus defined as the total number of unique object-view-cell pairs accumulated up to that point. Whenever the agent observes an object from its current position, the system checks whether this specific combination of object ID and view-cell is already known. If not, the pair is added to the set of discovered perspectives, incrementally increasing the diversity_t score and triggering a positive reward.

This mechanism not only promotes exploration of previously unvisited areas but also incentivizes the discovery of advantageous viewpoints that provide clear visibility of multiple objects. The discretization of view-cells follows the step sizes defined in the action space (translation step for `MoveAhead`, rotation step for `Rotate`), ensuring that diversity is measured consistently with the agent's actual movement grid.

6.4.4 Hyperparameter Optimization

An essential step in the training process is the calibration of the hyperparameter ρ, which controls the step penalty within the reward function. This penalty critically shapes the agent's behavior in terms of exploration efficiency and episode length.

Because the architecture and experimental setup differ from Li et al. [34], the original

weighting of ρ cannot be adopted directly. Without an appropriate penalty, agents in this setup collapse into degenerate strategies: (i) terminating immediately with the `Stop` action after a single step (episode length ≈ 1), or (ii) exploiting the reward structure by extending the episode to its hard cap of 40 steps in order to maximize cumulative score. Since r_t is defined as the potential difference $S_t - S_{t-1}$, and S_t contains a linear time penalty term $-\rho \cdot t$, the parameter ρ directly governs temporal preference and prevents both collapse modes when properly calibrated. For this reason, ρ was systematically tuned using the hyperparameter optimization framework *Optuna* [2].

The optimization was performed in two stages: initially, coarse intervals were explored to identify promising ranges, which were then provided to *Optuna* [2] to refine the search with a precision of 10^{-3} and obtain a precise optimum. The target was to calibrate agents to a mean episode length of about 25 steps, ensuring that scores across models are fairly comparable. Since the optimal value of ρ differs substantially between REINFORCE and A2C, separate optimizations were conducted for each algorithm. Preliminary experiments also indicated minor differences between LSTM- and Transformer-based policies within the same algorithm; therefore, independent optimizations were performed for each architecture to account for potential deviations in the optimal values of ρ.

To ensure a stable and generalizable estimate of the optimal step penalty, multiple agents (three per algorithm variant) were trained in parallel, and their mean performance was evaluated. Episodes were aggregated into blocks of 25 to reduce variance in the results. The objective function was defined to steer the agents toward an average episode length of approximately 25 steps. This target length was chosen to align with the results of Li et al. [34], thus ensuring comparability. It also provides a sufficient buffer to the hard termination limit of 40 steps, preventing agents from merely exhausting the maximum episode duration. Additionally, this choice contributes to a more efficient training process due to shorter sequences. A direct consequence of this choice, however, is that the agent can only explore a fraction of the environment within a single episode. This inherent limitation and its impact on performance are further addressed in the evaluation section. The objective function was structured as follows:

- For an average episode length of ≤ 25 steps, no penalty was applied and the original score was used.
- For lengths between 25 and 30 steps, the score was multiplied by a penalty factor that decreases linearly from 1.0 at 25 steps to 0.9 at 30 steps.
- For more than 30 steps, the score was set to 0, strictly penalizing longer episodes.

Hyperparameter optimization yielded a clear optimum for ρ, which was subsequently used for the training and fine-tuning of the models. This calibration step is essential to avoid degenerate policies and to ensure fair comparisons across architectural and algorithmic variants.

In contrast to the REINFORCE algorithm — where only the step penalty parameter ρ was optimized — the optimization for the A2C variant additionally included the value and entropy coefficients. This difference arises because REINFORCE has conceptually fewer hyperparameters, whereas A2C introduces extra coefficients that control the balance between policy and value loss, as well as the strength of entropy regularization. To ensure fair comparability, these two parameters were therefore incorporated into the optimization process for A2C. The evaluation metric serving as the objective function remained identical for both approaches. The final optimized values of all tuned hyperparameters are reported in Table 2 in Section 7.2.

Other parameters, such as the discount factor γ and the learning rate α, were set analogously to Li et al. [34] and kept identical across both algorithms. No additional optimization of these parameters was conducted, in order to preserve comparability. A complete overview of all hyperparameters used, apart from those optimized, can be found in the appendix (Table 8).

6.4.5 Training-Phase 2: Final Training and Fine-Tuning via RL

Following the pretraining of the feature encoder via IL, the second phase of the training process consists of training a new navigation policy from scratch while simultaneously fine-tuning the pretrained encoder using *RL*. During this phase, the agent interacts directly with the environment and optimizes its behavior based on the defined reward function. The pretrained feature encoder continues to be updated, providing a stable and informative representation from the outset. However, the main focus of this phase lies on learning the navigation policy anew.

For the training and fine-tuning, two RL algorithms were employed: REINFORCE and Advantage Actor-Critic (A2C). REINFORCE was chosen because it had already been used in the baseline by Li et al. [34]. To assess whether a more advanced RL method could yield improved performance, A2C was additionally implemented and evaluated. The choice of A2C was not motivated by an inherent superiority over alternative approaches such as PPO [48] or DQN [37], but rather by its straightforward implementation as an extension of the existing REINFORCE framework. The goal was thus primarily to demonstrate the potential benefits of employing a more sophisticated algorithm, rather than to identify the best possible algorithmic choice.

Both algorithms were evaluated under identical conditions to clearly isolate their relative contribution to improving the navigation policy. Training consisted of 1000 episodes for each algorithm, during which performance metrics such as mean episode length, achieved scores, and policy entropy were recorded at regular intervals. The metrics were logged using TensorBoard [1] for subsequent analysis.

6.5 Ablation Study

To systematically assess the impact of different architectural and algorithmic design choices, an ablation study is conducted. The aim is to disentangle the respective contributions of the RL algorithm (REINFORCE vs. A2C) and the sequential model within the policy network and the feature encoder (LSTM vs. Transformer). For this purpose, the following four configurations are trained and evaluated:

- **Baseline (REINFORCE + LSTM)**
  This configuration most closely mirrors the approach of Li et al. [34] and serves as the primary baseline. Both components of the agent, namely the feature encoder and the navigation policy, use LSTM modules trained with REINFORCE.

- **REINFORCE + Transformer**
  In this variant, the LSTM modules in both the feature encoder and the navigation policy are replaced with Transformer counterparts, to assess the effect of the architectural change within the classical REINFORCE framework.

- **A2C + LSTM**
  Here, the more advanced A2C algorithm is employed while retaining the LSTM modules in both components of the agent. This setup isolates the benefit of the algorithmic improvement independently of the architectural change.

- **A2C + Transformer**
  In this configuration, A2C is combined with Transformer modules in both the feature encoder and the navigation policy, enabling the evaluation of their potential synergy in improving navigation performance.

7 Evaluation

This chapter presents and analyzes the experimental results of the conducted investigations. Due to significant differences in the experimental setup — particularly the use of ground-truth scene graphs, a reduced action space, and a different data basis — a direct quantitative comparison of the achieved metrics with the results from Li et al. [34] is not feasible. Instead, this evaluation focuses on the results of the ablation study described in Section 6.5. The aim is to assess the effects of the proposed architectural and algorithmic changes within a consistent framework.

First, Section 7.1 defines the metrics used for performance assessment, and Section 7.2 details the experimental setup. The main results section, Section 7.3, analyzes the outcomes of the ablation study from multiple perspectives: it begins with a comparison of the overall performance, then examines the learning stability and exploration efficiency, and concludes with an evaluation of the agents' generalization capabilities on unseen scenes. To substantiate these quantitative findings, illustrative examples of agent behavior are qualitatively analyzed in Section 7.4. The chapter concludes in Section 7.5 with a summary discussion of the results and a critical examination of the limitations of this work.

7.1 Metrics

To evaluate the agent's performance, several metrics are used that capture different aspects of exploration and perception.

The primary metric is the *Node Recall*, which indicates how many of the objects present in the scene were detected by the agent. An object is considered detected once its aggregated soft visibility V_{GSSG} (see Section 6.3.2) exceeds the threshold of 0.8. This metric directly reflects the agent's ability to discover relevant objects in the scene.

Additionally, the *mean episode length* (*Mean Steps*) is measured, which is the average number of steps the agent requires per episode.

Furthermore, a derived metric, *Steps for Score 1*, is calculated from these two values, indicating the hypothetical number of steps the agent would need to achieve a Node Recall of 1.0 (meaning every object in the scene has been seen). However, this metric must be interpreted with great caution, as its validity is severely limited. It is based on the problematic assumption of a linear relationship between the number of steps and the achieved score. In practice, however, score gains typically diminish with increasing episode length, which can lead to misleading comparisons. For example, an agent that achieves a score of 0.7 in 20 steps receives a value of approx. 28, whereas a clearly superior agent that achieves a score of 0.95 in 28 steps ends up with a nearly identical value of approx. 29. The metric therefore has no reliable statistical validity and serves

merely as a rough heuristic. However, in the specific case where agents have very similar mean episode lengths, it can help to clarify their relative efficiency more intuitively.

A separate evaluation of *Edge Recall* is omitted, as edge information is extracted directly and error-free from the environment's ground-truth metadata and thus does not represent a differentiating performance factor.

Formal Definitions Let V be the set of all objects in the scene and $V_{GSSG}(v)$ be the aggregated soft visibility of an object $v \in V$ in the global graph. The *Node Recall* is then defined as:

$$\text{NodeRecall} \;=\; \frac{1}{|V|} \sum_{v \in V} \mathbf{1}[V_{GSSG}(v) \geq 0.8] \tag{20}$$

Let T be the number of evaluated episodes and s_t the number of steps required in episode t. The *Mean Steps* are:

$$\text{MeanSteps} \;=\; \frac{1}{T} \sum_{t=1}^{T} s_t \tag{21}$$

For illustrative purposes only, *Steps for Score 1* is defined as a linear extrapolation based on the current average performance. With a small stability parameter $\varepsilon = 10^{-6}$:

$$\text{StepsForScore1} \;=\; \frac{\text{MeanSteps}}{\max\bigl(\varepsilon,\ \text{NodeRecall}\bigr)} \tag{22}$$

This measure is a heuristic and makes no claim to statistical validity.

7.2 Experimental Setup

All experiments are conducted in the Ai2-THOR simulation environment, version 5.0.0. The experiments employ a strict separation into a training set and a test set. The training set comprises ten kitchen scenes (`FloorPlan1-12`), with `FloorPlan6` being excluded due to the need for diagonal movements not provided in the agent's action space, and `FloorPlan8` due to persistent simulation errors. The test set consists of three additional kitchen scenes (`FloorPlan13-15`), unseen during training, which are used exclusively for the final evaluation of generalization capabilities.

The training process for each of the four models spans 1000 macro-episodes, each initialized with ten different random seeds to ensure statistical reliability. One macro-episode corresponds to the completion of one episode in each of the ten training environments. The training is conducted in a single-worker setup with a maximum episode length of 40 steps. Throughout the process, performance is continuously evaluated to generate the learning curves, with stochastic sampling from the action distribution and random starting positions ensuring unique episodes.

For the final evaluation on the test set, the trained agents are evaluated for 100

macro-episodes, where one macro-episode corresponds to the completion of one episode in each of the three test scenes.

All performance metrics are calculated as the mean of the ten independent runs. For the visualization of the training phase, the plots are smoothed over 100 episodes, and the table values are averaged over the last 200 episodes. In contrast, for the evaluation on the test set, the smoothing is performed over ten episodes, and the averaging for the tables is done over the entire 100 episodes.

A critical preparatory step was the separate hyperparameter optimization for each of the four model variants, as detailed in Section 6.4.4. The goal was to establish a fair comparative framework for each configuration, specifically by calibrating the agents to a mean episode length of approximately 25 steps. Table 2 summarizes the final, optimized values for the step penalty ρ and the *A2C*-specific coefficients, which were used for all subsequently presented experiments.

	LSTM		**Transformer**	
Hyperparameter	**REINFORCE**	**A2C**	**REINFORCE**	**A2C**
Step penalty ρ	**0.071**	**0.014**	**0.037**	**0.013**
Value coefficient β_{val}	-	1	-	1
Entropy coefficient β_{ent}	-	0.03	-	0.03

Table 2: Optimized hyperparameters for the agents

7.3 Experimental Results

This section presents the results of the ablation study. The analysis is structured in multiple stages to provide a comprehensive picture of the agent's performance: It begins with a comparison of the overall performance on the training data to highlight the fundamental performance differences. Subsequently, the learning stability is examined based on the reward progression to investigate the causes of these differences. This is followed by an assessment of the agents' exploration efficiency. Finally, the generalization capability of the final policies is evaluated on a separate test set to verify their robustness.

7.3.1 Overall Performance Comparison

The key performance indicators are the *Node Recall* achieved over time and the number of steps required per episode, as shown in Figure 18. The final numerical values, averaged over the last 200 episodes of training, are summarized in Table 3 and Table 4.

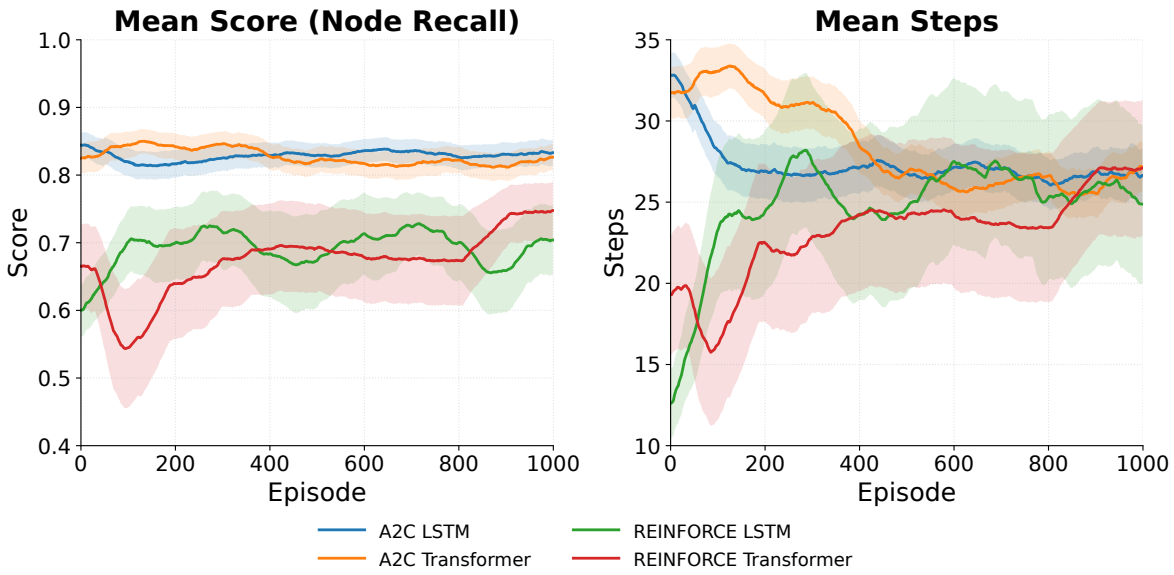

Figure 18: **Mean Score (Node Recall) and Mean Steps on train scenes** (smoothed, averaged across seeds); shaded area shows standard error (mean ± SE across seeds with SE = $\hat{\sigma}/\sqrt{n}$; not a confidence interval)

- *A2C* agents converge quickly to a high and stable performance level
- *REINFORCE* agents exhibit a more volatile and ultimately lower performance
- *Mean Steps* for all agents converge to a similar range, indicating successful calibration

The results show a clear performance advantage for the *A2C*-based agents over the *REINFORCE* variants. The *A2C* agents achieve a stable final *Node Recall* of approximately 0.83 *(LSTM)* and 0.82 *(Transformer)*, whereas the *REINFORCE* agents lag significantly behind with scores of approximately 0.68 *(LSTM)* and 0.72 *(Transformer)* (see Table 3). Furthermore, the learning curves in Figure 18 reveal differences in stability: the wider standard error bands of the *REINFORCE* variants suggest a higher inter-seed variance, indicating a less stable training process compared to the *A2C* agents.

Agent	**Mean Score (last 200)**	**Std. Dev.**
A2C LSTM	0.8300	0.0187
A2C Transformer	0.8173	0.0215
REINFORCE LSTM	0.6795	0.0456
REINFORCE Transformer	0.7236	0.0397

Table 3: **Final Score (Node Recall) on train scenes** (averaged over the last 200 episodes, higher is better)

Within the algorithm families, it is apparent that for the *A2C* algorithm, both architectures — *LSTM* and *Transformer* — achieve a very similar, high level of performance. In contrast, the *REINFORCE* algorithm visibly benefits from the *Transformer* architecture, which outperforms the *LSTM* variant.

As shown in Table 4, all four agent variants operate within a very similar range of mean episode length — around 26 steps — thanks to the hyperparameter optimization. This confirms that the observed performance differences in *Node Recall* are not attributable to varying exploration durations, but rather to the effectiveness and stability of the respective algorithm and architecture.

Agent	Mean Steps (last 200)	Std. Dev.
A2C LSTM	26.60	1.43
A2C Transformer	26.04	1.89
REINFORCE LSTM	25.53	2.33
REINFORCE Transformer	26.08	1.93

Table 4: **Mean episode length on train scenes** (averaged over the last 200 episodes)

7.3.2 Analysis of Learning Stability and Reward Optimization

To investigate the causes of the observed performance differences, the progression of the mean reward per episode is analyzed (see Figure 19). This analysis reveals fundamental differences in learning stability between *A2C* and *REINFORCE*. The *A2C* agents are

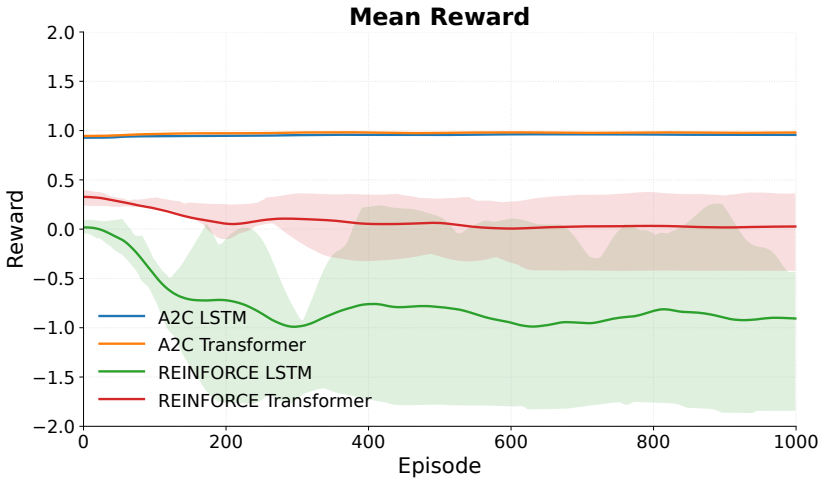

Figure 19: **Mean reward per training episode** (smoothed, averaged across seeds); shaded area shows the interquartile range (IQR) across seeds

- *REINFORCE* fails to sustainably optimize its policy, as the reward for both of its variants deteriorates over time
- *A2C* maintains a stable, positive level
- Note: Absolute rewards are not directly comparable due to different calibrations of the step penalty ρ (see Table 2)

able to successfully maximize the reward and maintain it at a stable, positive level throughout the entire training process. This indicates a robust and convergent learning process. In contrast, the *REINFORCE* agents fail to sustainably optimize the reward. Instead of improving, their reward deteriorates from the very beginning of training, a decay that is particularly pronounced for the *REINFORCE-LSTM* variant, which falls deep into the negative range. This behavior can be interpreted as an indicator of unstable learning, which can result from the high variance of the gradient estimates of the *REINFORCE* algorithm. This instability is a plausible explanation for the ultimately poorer performance in *Node Recall*.

7.3.3 Evaluation of Exploration Efficiency

Finally, the efficiency of the agents is assessed using the heuristic metric *Steps for Score 1*, which is presented in Table 5. This metric estimates how many steps an agent would hypothetically need to find all objects in the scene.

Agent	**Steps for Score 1**	**Std. Dev.**
A2C LSTM	31.94	0.38
A2C Transformer	31.88	0.70
REINFORCE LSTM	34.81	1.91
REINFORCE Transformer	33.42	1.66

Table 5: **Steps for Score 1 on train scenes** (averaged over the last 200 episodes, lower is better)

This metric also highlights the superiority of the *A2C* agents. With approximately 32 steps, they hypothetically require fewer actions to achieve a score of 1.0 than their *REINFORCE* counterparts (approx. 33-35 steps). The *A2C*-based methods are thus not only more effective in terms of the final score, but they also achieve it in a more efficient manner. The progression of this metric over time is shown in Figure 23 in the appendix.

7.3.4 Generalization Performance on Unseen Scenes

To test whether the learning stability of the *A2C* agents observed during the training phase also leads to more robust generalization capabilities, the final agents were evaluated on three previously unseen kitchen scenes. Figure 20 shows the performance progression over 100 evaluation episodes, while Table 6 and Table 7 summarize the final, averaged metrics.

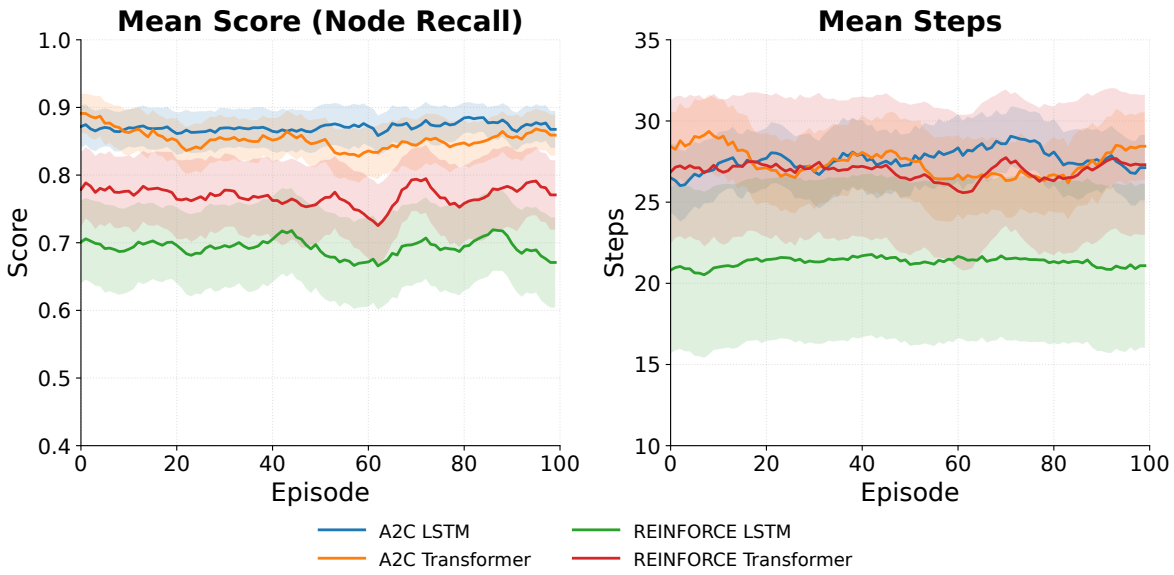

Figure 20: **Mean Score (Node Recall) and Mean Steps on unseen test scenes** (smoothed, averaged across seeds); shaded area shows standard error (mean ± SE across seeds with SE = $\hat{\sigma}/\sqrt{n}$; not a confidence interval)

- Clear and stable performance gap visible between the superior *A2C* agents and the underperforming *REINFORCE* variants
- *REINFORCE-LSTM* agent exhibits anomalous behavior, consistently taking fewer steps than all other agents

The results on the test set confirm the primary finding from the training phase: the *A2C*-based agents significantly outperform the *REINFORCE* variants. The ranking of the agents remains consistent, which underscores the robustness of the previous analysis. Interestingly, all agents achieve even slightly higher absolute *Node Recall* scores on the unseen scenes than in training, which could indicate potentially simpler layouts of the test scenes.

Agent	**Mean Score**	**Std. Dev.**
A2C LSTM	0.8709	0.0271
A2C Transformer	0.8536	0.0338
REINFORCE Transformer	0.7691	0.0405
REINFORCE LSTM	0.6936	0.0383

Table 6: **Final Score (Node Recall) on the three unseen test scenes** (averaged over all 100 eval episodes, higher is better)

A notable behavioral change is evident in the *REINFORCE-LSTM* agent on the test set (Table 7). Its mean episode length drops significantly to approx. 21 steps, while the other agents maintain a consistent step count. This shortened episode duration, coupled with the persistently lowest score, suggests that this agent struggles to follow

an effective exploration strategy in unknown environments and frequently terminates episodes prematurely. In contrast, the *REINFORCE-Transformer* agent generalizes more effectively, maintaining a consistent episode length and achieving a significantly higher score. This divergence underscores the poor generalization capability of the policy trained with *REINFORCE* and an *LSTM*, while suggesting that the *Transformer* architecture may have a regularizing effect that leads to more stable behavior.

Agent	**Mean Steps**	**Std. Dev.**
A2C LSTM	27.62	1.74
A2C Transformer	27.42	1.94
REINFORCE Transformer	26.92	1.36
REINFORCE LSTM	21.32	0.94

Table 7: **Mean episode length on the test scenes** (averaged over all 100 eval episodes)

7.4 Qualitative Analysis

To visualize the learning instability of the *REINFORCE* agents identified in the reward analysis, this section qualitatively examines the concrete behavior of the agents based on their trajectories. The examination of visual examples helps to make the causes for the performance differences — particularly between the *A2C* and *REINFORCE* algorithms — tangible.

Figure 21 provides an exemplary comparison of the paths taken by all four agent configurations within the same scene, starting from an identical position. The comparison reveals a drastic difference in exploration behavior. The two *A2C*-based agents

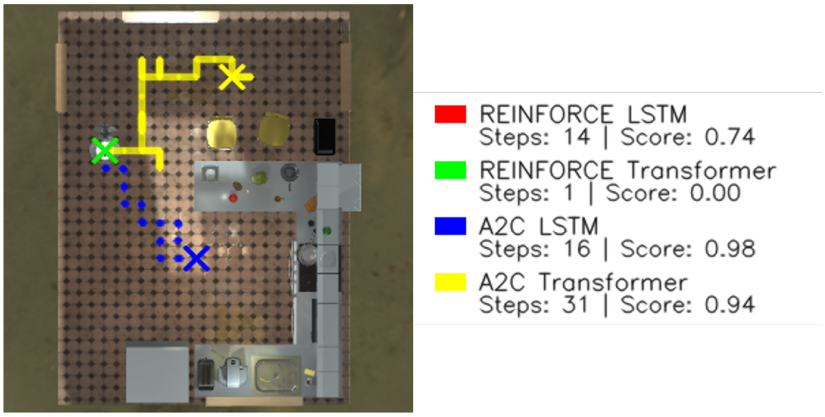

Figure 21: Direct comparison of the trajectories of all four agents in scene `FloorPlan14`
- the *A2C* agents (yellow, blue) explore the scene
- the *REINFORCE* agents (green, red) do not move at all

(yellow and blue) generate long, meaningful trajectories that cover large parts of the environment, resulting in high *Node Recall* scores. In stark contrast, the *REINFORCE* agents fail completely in this scenario. The *REINFORCE-Transformer* (green) remains stationary and terminates the episode immediately, achieving a score of zero. Meanwhile, the *REINFORCE-LSTM* (red, obscured by the green marker) also fails to move, presumably only rotating on the spot for 14 steps to achieve a meager score of 0.74. This example vividly visualizes the learning instability of the *REINFORCE* algorithm, as noted in Section 7.3.2, which can lead to a catastrophic failure of the exploration policy in practice.

This pattern of effective exploration by *A2C* and flawed behavior by *REINFORCE* is also evident in pairwise comparisons across the unseen test scenes (see Figure 22). Both

(a) `FloorPlan13` (LSTM) (b) `FloorPlan14` (LSTM) (c) `FloorPlan15` (LSTM)

(d) `FloorPlan13` (Transf.) (e) `FloorPlan14` (Transf.) (f) `FloorPlan15` (Transf.)

Figure 22: Comparison of trajectories on the three test scenes

- Top row: *A2C-LSTM* (green) vs. *REINFORCE-LSTM* (red)
- Bottom row: *A2C-Transformer* (green) vs. *REINFORCE-Transf.* (red)
- *A2C* agents consistently find efficient paths, while the *REINFORCE* agents exhibit flawed and inconsistent behaviors

the *A2C-LSTM* (top row) and the *A2C-Transformer* (bottom row) agents consistently execute efficient, purposeful paths that lead to high scores across all three scenes, demonstrating the robustness of policies trained with A2C.

In stark contrast, the two *REINFORCE* variants exhibit different, yet equally ineffective, failure modes. As previously noted, the *REINFORCE-LSTM* agent (top row, red) displays inconsistent behavior, often terminating prematurely. The *REINFORCE-Transformer* agent (bottom row, red), however, appears to have converged to a consistent but degenerate policy of moving in small, unproductive loops. This suggests that while the *Transformer* architecture may prevent the immediate termination seen in the *LSTM* variant, the underlying instability of *REINFORCE* still prevents it from learning a genuinely effective exploration strategy.

In summary, the qualitative observations confirm the quantitative results. The agents trained with *A2C* learn robust and efficient exploration strategies, whereas the *REINFORCE* agents are prone to unstable policies that manifest in inefficient or flawed behavior.

7.5 Discussion and Limitations

The conducted ablation study, whose results were validated by an evaluation on unseen scenes, clearly shows that the choice of the reinforcement learning algorithm had a substantially greater impact on navigation performance than the choice of the sequential network architecture. Due to its more stable learning behavior, the *A2C* algorithm led to significantly better and more efficient results than the *REINFORCE* algorithm. The comparison between the *Transformer* and the *LSTM* did not yield a clear superiority of either architecture. It is conceivable, however, that the advantages of the *Transformer* in processing long-term dependencies did not fully materialize in the currently short episodes with a maximum of 40 steps. In more complex environments requiring longer exploration paths, the *Transformer*-based agent could potentially leverage its strengths more effectively.

To properly contextualize the presented results, it is important to reflect on the potential limitations of the chosen methodology and their influence on the validity of the findings.

Simulated Perception and Perfect Information A central limitation is that the perception is simulated rather than learned. The scene graphs are derived directly from the environment's metadata, relying on two simulator-specific advantages: perfect object matching via persistent `objectId`s and a heuristic soft visibility model. The parameters for this visibility model were chosen pragmatically to *simulate* a learned component; a sensitivity analysis was deliberately not performed, but since all variants use the same parameters, no impact on the relative comparisons is expected. While

this was a deliberate choice to isolate the navigation module, this simplification — in conjunction with the monotonically increasing visibility update rule (Equation 15) — creates a potential loophole in the reward system. Specifically, an agent can achieve a high score by adopting a simplistic policy, such as rotating on the spot. In the small kitchen environments, most objects are visible from such a vantage point, and with each repeated sighting, their cumulative visibility score increases, generating a positive reward for non-exploratory behavior. This is a plausible explanation for the degenerate policies learned by the unstable *REINFORCE* agents, which achieved high scores despite moving in unproductive loops. Crucially, the fact that the *A2C* agents did not exploit this loophole demonstrates that the setup is not fundamentally flawed, but rather that *REINFORCE* is susceptible to converging to such simplistic local optima. A dedicated, learned perception module would likely mitigate this issue, as it would require the agent to get closer to objects for confident detection.

Restricted Action Space The action space was restricted to four movement directions. While this simplifies navigation, it may limit the transferability of the results to more realistic action spaces with more degrees of freedom.

Expert Trajectory Planning The expert routes are based on *Ant Colony Optimization* and thus provide heuristic, non-guaranteed optimal solutions. This means that the expert trajectories, while effective, are not guaranteed to be optimal, and deviations from the theoretical optimum are possible.

Limited Episode Length The hyperparameter optimization was deliberately designed to achieve an average episode length of approximately 25 steps. This constitutes a significant limitation, as it means that the agent can only explore a fraction of the environment. Although episodes are only forcibly terminated after a hard cap of 40 steps, the reward function effectively discourages longer trajectories. To enable a more complete exploration, this limit could be raised in future work. However, such an adjustment would require a renewed optimization of the step-penalty parameter ρ to find a balance between exploration depth and efficiency.

8 Conclusion and Future Work

This thesis presented the design, implementation, and evaluation of a modernized navigation module for the task of *Embodied Semantic Scene Graph Generation (ESSGG)*. This chapter summarizes the key contributions and findings of the work, answers the research questions formulated in the introduction, and provides an outlook on future research directions.

8.1 Conclusion

The core objective of this work was to investigate the extent to which modern architectures and algorithms can improve the performance of an embodied AI agent in semantic exploration. The starting point was the analysis of the existing ESSGG framework by Li et al. [34], whose navigation module is based on an *LSTM* architecture and the *REINFORCE* algorithm — both technologies with known limitations regarding the processing of long-term dependencies and learning stability.

To address these limitations, a comprehensive ablation study was conducted in which both the sequential architecture (*LSTM* vs. *Transformer*) and the reinforcement learning algorithm (*REINFORCE* vs. *A2C*) were systematically replaced and evaluated. The experimental results from Chapter 7 provided a clear and unambiguous picture: The switch from the *REINFORCE* to the *A2C* learning algorithm led to a significant and robust increase in performance across all metrics. The *A2C*-based agents demonstrably learned more stable policies, which manifested as a higher *Node Recall*, better exploration efficiency, and superior generalization capability on unseen scenes.

In contrast, replacing the *LSTM* architecture with a *Transformer* yielded no decisive advantage in the conducted experiments, with both models performing comparably. It is conceivable, however, that the *Transformer's* advantages in processing long-term dependencies would become more apparent in scenarios requiring longer trajectories. The central conclusion of this work therefore remains that for the navigation task investigated here, the choice of a stable and sample-efficient learning algorithm like *A2C* is of far greater importance than the choice of the sequential network architecture.

8.2 Answering the Research Questions

Based on these findings, the implicit research questions derived from the aim of this thesis in Section 1.2 can be answered as follows:

1. Does replacing the *REINFORCE* algorithm with an *A2C* agent lead to a performance improvement?

Yes, unequivocally. The evaluation has shown that the *A2C* algorithm is superior to the *REINFORCE* algorithm in every respect in this scenario. It leads to signifi-

cantly higher scores, learns more stable policies (as evidenced by the reward analysis), and shows more robust generalization performance. The *REINFORCE* agents suffered from high variance, which manifested as unstable and often suboptimal behavior.

2. Does replacing the *LSTM* architecture with a *Transformer* model lead to a performance improvement?
No, not in the setup investigated here. In combination with the stable *A2C* algorithm, the *Transformer* showed no significantly better performance than the *LSTM* architecture. An improvement was only observed in combination with the unstable *REINFORCE* algorithm, which suggests that the *Transformer* may have had a slight stabilizing effect here. However, the hypothesis that the *Transformer* is superior per se could not be confirmed.

3. How does the combination of both modernizations affect the overall performance?
The algorithmic improvement is the dominant factor. The performance of the top-performing *A2C-LSTM* and *A2C-Transformer* agents was nearly identical, with the former holding a marginal advantage. The largest performance leap was achieved by switching from *REINFORCE* to *A2C*, regardless of the architecture used. The combination of both modernizations thus leads to a high-performing agent, whose strength, however, is primarily based on the choice of the learning algorithm.

8.3 Future Work

Several promising directions for future research emerge from the findings and limitations of this work.

Learning Scene Graph Generation from Sensor Data The most fundamental extension for future work is to replace the ground-truth scene graphs with a learned perception module. In this work, metadata was deliberately used to evaluate the navigation policy in isolation. However, for real-world deployment and a fair comparison with end-to-end systems, the agent must learn to generate the local scene graph directly from its sensor inputs, such as RGB images and optional depth information. This step would transform the system into a fully autonomous agent that independently perceives and interprets its environment. This could be implemented using specialized models for scene graph generation or through multimodal foundation models that combine visual and linguistic capabilities.

Testing in More Complex Scenarios A central limitation of this work is the restricted episode length. As noted in the discussion, the advantages of the *Transformer* in processing long-term dependencies might only become apparent in larger environ-

ments that require longer exploration paths. Future experiments should therefore be conducted in more complex scenes with an increased step limit to re-evaluate the choice of architecture.

Expansion of the Action Space The present work is limited to a highly discretized action space with four movement directions and 90-degree rotations. An important step towards more realistic navigation is the expansion of this action space to provide the agent with finer control over its movements. This could include the introduction of diagonal movements, rotations in smaller angular increments, or even continuous actions where the agent directly predicts the distance to travel and the angle to turn. Such a more expressive action space would potentially increase navigation efficiency in complex environments, but would also pose a greater challenge to the learning algorithm.

Use of Vision-Language Models for Navigation and Perception A promising future research direction is the replacement of specialized modules with a unified approach based on *Vision-Language Models (VLMs)*. For instance, a VLM could replace the navigation policy, generate the scene graph directly from visual inputs, or unify both functions within a single end-to-end model. Furthermore, the inherent reasoning capabilities of VLMs could be leveraged to create a more robust system. This would involve prompting the model to critically interrogate its own perceptual judgments and action proposals before they are finalized, leading to more deliberate and reliable behavior.

Combination of Semantic and Geometric Maps The current agent relies on a purely topological scene graph (GSSG). An extension would be to fuse the GSSG with a metric, geometric map of the environment (e.g. a 2D occupancy map). Such a "semantic map" would not only store which objects exist and how they are related, but also where they are located in space. This could further increase exploration efficiency, as the agent could identify and target "knowledge gaps" not only semantically but also spatially.

Critical Reflection on Baseline Reproducibility During this project, it was found that reproducing the results from the original work by Li et al. [34] is challenging. In particular, the use of the *REINFORCE* algorithm, which is known for its high variance, raises the question of the specific conditions under which the original results were achieved. Inquiries to the authors for clarification remained unanswered. Future work should therefore critically question the reproducibility of RL-based studies and ideally use more robust algorithms such as *A2C* or *PPO* as baselines for comparisons.

Appendix

Hyperparameter	Value
General parameters	
Gradient clipping threshold	0.5
Max steps per episode	40
Policy network parameters	
LSTM hidden size, MLP hidden dimensions	512
LSTM number of layers	2
Transformer encoder layers L	2
Transformer model dimension d_{model} & feedforward dim	512
Transformer attention heads	4
Transformer dropout rate	0.1
Imitation learning	
Epochs	100
Learning rate α	1×10^{-4}
Batch size	8
Validation split	0.15
Reinforcement learning	
Shared (Both Agents)	
Epochs / Episodes	1000
Learning rate α	1×10^{-4}
Discount factor γ	0.99
n-step return length	40

Table 8: Overview of used hyperparameters

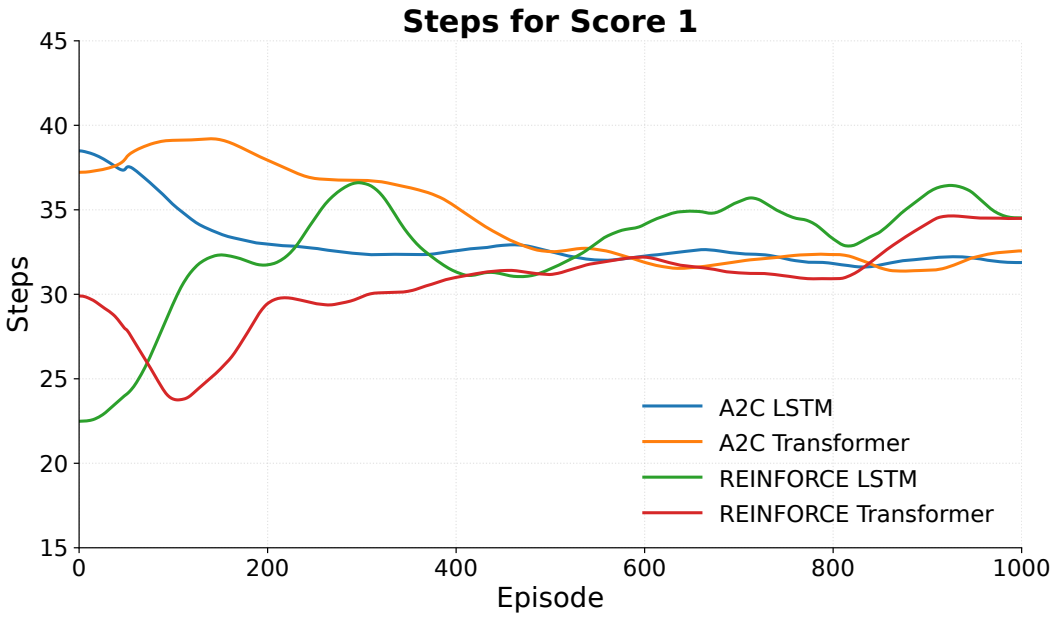

Figure 23: **Steps for Score 1 on train scenes** (smoothed, averaged across seeds, lower is better)

Bibliography

[1] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dan Mane, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viegas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. TensorFlow: Large-Scale Machine Learning on Heterogeneous Distributed Systems, 2016. Version Number: 2.

[2] Takuya Akiba, Shotaro Sano, Toshihiko Yanase, Takeru Ohta, and Masanori Koyama. Optuna: A Next-generation Hyperparameter Optimization Framework, 2019. Version Number: 1.

[3] Allen Institute for AI (AI2). AI2-THOR documentation. Online resource, 2025. Available at: https://ai2thor.allenai.org/ithor/documentation. Accessed on 2025-05-20.

[4] Allen Institute for AI (AI2). AI2-THOR ithor environment overview. Online resource, 2025. Available at: https://ai2thor.allenai.org/ithor. Accessed on 2025-05-20.

[5] Jimmy Lei Ba, Jamie Ryan Kiros, and Geoffrey E. Hinton. Layer Normalization, July 2016. arXiv:1607.06450 [stat].

[6] Dzmitry Bahdanau, Kyunghyun Cho, and Yoshua Bengio. Neural machine translation by jointly learning to align and translate. In *International Conference on Learning Representations (ICLR)*, 2015.

[7] Alejandro Barredo Arrieta, Natalia Díaz-Rodríguez, Javier Del Ser, Adrien Bennetot, Siham Tabik, Alberto Barbado, Salvador Garcia, Sergio Gil-Lopez, Daniel Molina, Richard Benjamins, Raja Chatila, and Francisco Herrera. Explainable Artificial Intelligence (XAI): Concepts, taxonomies, opportunities and challenges toward responsible AI. *Information Fusion*, 58:82–115, June 2020.

[8] Y. Bengio, P. Simard, and P. Frasconi. Learning long-term dependencies with gradient descent is difficult. *IEEE Transactions on Neural Networks*, 5(2):157–166, 1994.

[9] Xiaojun Chang, Pengzhen Ren, Pengfei Xu, Zhihui Li, Xiaojiang Chen, and Alex Hauptmann. A comprehensive survey of scene graphs: Generation and application. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 45(1):1–26, January 2023.

[10] Tao Chen, Saurabh Gupta, and Abhinav Gupta. Learning Exploration Policies for Navigation, March 2019. arXiv:1903.01959 [cs].

[11] Abhishek Das, Samyak Datta, Georgia Gkioxari, Stefan Lee, Devi Parikh, and Dhruv Batra. Embodied question answering. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 1–10, 2018.

[12] DatabaseCamp. Long short-term memory networks (lstm) - simply explained! Online resource, 2024. Available at: https://databasecamp.de/en/ml/lstms. Accessed on 2025-07-31.

[13] Jia Deng, Wei Dong, Richard Socher, Li-Jia Li, Kai Li, and Li Fei-Fei. ImageNet: A large-scale hierarchical image database. In *2009 IEEE Conference on Computer Vision and Pattern Recognition*, pages 248–255, June 2009. ISSN: 1063-6919.

[14] M. Dorigo and L.M. Gambardella. Ant colony system: a cooperative learning approach to the traveling salesman problem. *IEEE Transactions on Evolutionary Computation*, 1(1):53–66, April 1997.

[15] Marco Dorigo, Mauro Birattari, and Thomas Stutzle. Ant colony optimization. *IEEE Computational Intelligence Magazine*, 1(4):28–39, November 2006.

[16] Raphael Druon, Yusuke Yoshiyasu, Asako Kanezaki, and Alassane Watt. Visual object search by learning spatial context. *IEEE Robotics and Automation Letters*, 5(2):1279–1286, 2020.

[17] Jiafei Duan, Samson Yu, Hui Li Tan, Hongyuan Zhu, and Cheston Tan. A Survey of Embodied AI: From Simulators to Research Tasks. *IEEE Transactions on Emerging Topics in Computational Intelligence*, 6(2):230–244, April 2022.

[18] Sourav Garg, Niko Sünderhauf, Feras Dayoub, Douglas Morrison, Akansel Cosgun, Gustavo Carneiro, Qi Wu, Tat-Jun Chin, Ian Reid, Stephen Gould, Peter Corke, and Michael Milford. Semantics for robotic mapping, perception and interaction: A survey. *Foundations and Trends® in Robotics*, 8(1–2):1–224, 2020.

[19] Ian Goodfellow, Aaron Courville, and Yoshua Bengio. *Deep learning*. Adaptive computation and machine learning. The MIT Press, Cambridge, Massachusetts, 2016.

[20] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep Residual Learning for Image Recognition, December 2015. arXiv:1512.03385 [cs].

[21] Sepp Hochreiter. *Untersuchungen zu dynamischen neuronalen Netzen*. PhD thesis, Technische Universität München, Munich, Germany, 1991.

[22] Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural Computation*, 9:1735–1780, November 1997.

[23] Ziniu Hu, Yuxiao Dong, Kuansan Wang, and Yizhou Sun. Heterogeneous Graph Transformer, 2020. Version Number: 1.

[24] Ahmed Hussein, Mohamed Medhat Gaber, Eyad Elyan, and Chrisina Jayne. Imitation Learning: A Survey of Learning Methods. *ACM Computing Surveys*, 50(2):1–35, March 2018.

[25] IBM Cloud Education. What is a neural network? Online resource, 2021. Available at: https://www.ibm.com/think/topics/neural-networks. Accessed on 2025-06-18.

[26] IBM Cloud Education. What is a recurrent neural network (rnn)? Online resource, 2024. Available at: https://www.ibm.com/think/topics/recurrent-neural-networks. Accessed on 2025-06-18.

[27] IBM Cloud Education. What are convolutional neural networks? Online resource, 2025. Available at: https://www.ibm.com/think/topics/convolutional-neural-networks. Accessed on 2025-06-18.

[28] IBM Cloud Education. What is a transformer model? Online resource, 2025. Available at: https://www.ibm.com/think/topics/transformer-model. Accessed on 2025-07-01.

[29] Gareth James, Daniela Witten, Trevor Hastie, Robert Tibshirani, and Jonathan Taylor. Unsupervised Learning. In Gareth James, Daniela Witten, Trevor Hastie, Robert Tibshirani, and Jonathan Taylor, editors, *An Introduction to Statistical Learning: with Applications in Python*, pages 503–556. Springer International Publishing, Cham, 2023.

[30] Diederik P. Kingma and Jimmy Ba. Adam: A Method for Stochastic Optimization, January 2017. arXiv:1412.6980 [cs].

[31] Eric Kolve, Roozbeh Mottaghi, Winson Han, Eli VanderBilt, Luca Weihs, Alvaro Herrasti, Daniel Gordon, Yuke Zhu, Abhinav Gupta, and Ali Farhadi. Ai2-THOR: An Interactive 3D Environment for Visual AI. *arXiv*, 2017.

[32] Ranjay Krishna, Yuke Zhu, Oliver Groth, Justin Johnson, Kenji Hata, Joshua Kravitz, Stephanie Chen, Yannis Kalantidis, Li-Jia Li, David A. Shamma, Michael S. Bernstein, and Fei-Fei Li. Visual Genome: Connecting Language and Vision Using Crowdsourced Dense Image Annotations, February 2016. arXiv:1602.07332 [cs].

[33] Y. Lecun, L. Bottou, Y. Bengio, and P. Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324, November 1998.

[34] Xinghang Li, Di Guo, Huaping Liu, and Fuchun Sun. Embodied Semantic Scene Graph Generation. In *Proceedings of the 5th Conference on Robot Learning*, pages 1585–1594. PMLR, January 2022. ISSN: 2640-3498.

[35] Michael Lingelbach, Chengshu Li, Minjune Hwang, Andrey Kurenkov, Alan Lou, Roberto Martín-Martín, Ruohan Zhang, Li Fei-Fei, and Jiajun Wu. Task-Driven Graph Attention for Hierarchical Relational Object Navigation, 2023. Version Number: 1.

[36] Volodymyr Mnih, Adrià Puigdomènech Badia, Mehdi Mirza, Alex Graves, Timothy P. Lillicrap, Tim Harley, David Silver, and Koray Kavukcuoglu. Asynchronous Methods for Deep Reinforcement Learning, 2016. Publisher: arXiv Version Number: 2.

[37] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A. Rusu, Joel Veness, Marc G. Bellemare, Alex Graves, Martin Riedmiller, Andreas K. Fidjeland, Georg Ostrovski, Stig Petersen, Charles Beattie, Amir Sadik, Ioannis Antonoglou, Helen King, Dharshan Kumaran, Daan Wierstra, Shane Legg, and Demis Hassabis. Human-level control through deep reinforcement learning. *Nature*, 518(7540):529–533, February 2015.

[38] Ashvin Nair, Bob McGrew, Marcin Andrychowicz, Wojciech Zaremba, and Pieter Abbeel. Overcoming Exploration in Reinforcement Learning with Demonstrations, February 2018. arXiv:1709.10089 [cs].

[39] Vinod Nair and Geoffrey Hinton. Rectified Linear Units Improve Restricted Boltzmann Machines. In *Proceedings of ICML*, volume 27, pages 807–814, June 2010.

[40] Vladimir Nasteski. An overview of the supervised machine learning methods. *HORIZONS.B*, 4:51–62, December 2017.

[41] Takayuki Osa, Joni Pajarinen, Gerhard Neumann, J. Andrew Bagnell, Pieter Abbeel, and Jan Peters. An Algorithmic Perspective on Imitation Learning. *Foundations and Trends® in Robotics*, 7(1-2):1–179, 2018.

[42] Nikita Oskolkov, Huzhenyu Zhang, Dmitry Makarov, Dmitry Yudin, and Aleksandr Panov. SGN-CIRL: Scene Graph-based Navigation with Curriculum, Imitation, and Reinforcement Learning, 2025. Version Number: 1.

[43] Dean A. Pomerleau. ALVINN: an autonomous land vehicle in a neural network. In *Proceedings of the 2nd International Conference on Neural Information Processing Systems*, NIPS'88, pages 305–313, Cambridge, MA, USA, January 1988. MIT Press.

[44] Santhosh K. Ramakrishnan, Dinesh Jayaraman, and Kristen Grauman. An Exploration of Embodied Visual Exploration. *International Journal of Computer Vision*, 129(5):1616–1649, May 2021.

[45] Zachary Ravichandran, Lisa Peng, Nathan Hughes, J. Daniel Griffith, and Luca Carlone. Hierarchical Representations and Explicit Memory: Learning Effective Navigation Policies on 3D Scene Graphs using Graph Neural Networks. In *2022 International Conference on Robotics and Automation (ICRA)*, pages 9272–9279, May 2022.

[46] Stephane Ross, Geoffrey J. Gordon, and J. Andrew Bagnell. A Reduction of Imitation Learning and Structured Prediction to No-Regret Online Learning, March 2011. arXiv:1011.0686 [cs].

[47] Stuart J. Russell and Peter Norvig. *Artificial intelligence: a modern approach*. Prentice Hall series in artificial intelligence. Pearson, Boston, fourth, global edition, 2022.

[48] John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. Proximal Policy Optimization Algorithms, 2017. Version Number: 2.

[49] Scopus. Search results for "embodied ai" (1974–2024). Database query, 2025. Retrieved on 2025-08-01 from Scopus; query: TITLE-ABS-KEY("embodied AI").

[50] Alessandro Suglia, Qiaozi Gao, Jesse Thomason, Govind Thattai, and Gaurav Sukhatme. Embodied bert: A transformer model for embodied, language-guided visual task completion, 2021.

[51] Yizhou Sun and Jiawei Han. *Mining heterogeneous information networks: principles and methodologies*. Number #5 in Synthesis lectures on data mining and knowledge discovery 2151-0067. Morgan & Claypool Publishers, San Rafael, Calif., online-ausg edition, 2012.

[52] Richard S. Sutton and Andrew Barto. *Reinforcement learning: an introduction*. Adaptive computation and machine learning. The MIT Press, Cambridge, Massachusetts London, England, second edition, 2020.

[53] Mark Towers, Ariel Kwiatkowski, Jordan Terry, John U Balis, Gianluca De Cola, Tristan Deleu, Manuel Goulão, Andreas Kallinteris, Markus Krimmel, Arjun KG, et al. Gymnasium: A standard interface for reinforcement learning environments, 2024.

[54] Unity Technologies. Unity engine: 2d & 3d development platform. Online resource, 2025. Available at: https://unity.com/products/unity-engine. Accessed on 2025-08-04.

[55] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is All you Need, June 2017.

[56] Petar Veličković, Guillem Cucurull, Arantxa Casanova, Adriana Romero, Pietro Liò, and Yoshua Bengio. Graph Attention Networks, 2017. Version Number: 3.

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

[58] Zonghan Wu, Shirui Pan, Fengwen Chen, Guodong Long, Chengqi Zhang, and Philip S. Yu. A Comprehensive Survey on Graph Neural Networks. *IEEE Transactions on Neural Networks and Learning Systems*, 32(1):4–24, January 2021.

[59] Zhiqing Xiao. *Reinforcement Learning: Theory and Python Implementation*. Springer Nature, Singapore, 2024.

[60] Wei Xie, Haobo Jiang, Yun Zhu, Jianjun Qian, and Jin Xie. NaviFormer: A Spatio-Temporal Context-Aware Transformer for Object Navigation. *Proceedings of the AAAI Conference on Artificial Intelligence*, 39(14):14708–14716, April 2025.

[61] Brian Yamauchi. A frontier-based approach for autonomous exploration. In *Proceedings 1997 IEEE International Symposium on Computational Intelligence in Robotics and Automation CIRA'97*, pages 146–151. IEEE, 1997.

[62] Zhen Zeng, Adrian Röfer, and Odest Chadwicke Jenkins. Semantic linking maps for active visual object search. In *2020 IEEE International Conference on Robotics and Automation (ICRA)*, pages 1984–1990. IEEE, 2020.

[63] Jingwei Zhang, Lei Tai, Ming Liu, Joschka Boedecker, and Wolfram Burgard. Neural slam: Learning to explore with external memory, 2017.

[64] Fuzhen Zhuang, Zhiyuan Qi, Keyu Duan, Dongbo Xi, Yongchun Zhu, Hengshu Zhu, Hui Xiong, and Qing He. A Comprehensive Survey on Transfer Learning. *Proceedings of the IEEE*, 109(1):43–76, January 2021.