University of Padua

Department of Mathematics "Tullio Levi-Civita"

Computer Science

Socially-aware ObjectGoal Navigation using Proximity-based Auxiliary tasks

Master thesis

Supervisor

Prof. Lamberto Ballan

Co-supervisor

Enrico Cancelli

Master Candidate

Davide Franzoso

2023-2024

Abstract

There has been a growing interest in "Embodied AI" in recent years. In an embodied AI task, an agent no longer learns from a static dataset extracted from the internet (e.g. image classification) but from the interaction with the environment. Three main problems are studied in embodied AI: Visual exploration, Visual navigation, and Embodied Question answering.

In this thesis, I have focused my work on a specific visual navigation task: Object-goal navigation. In object-goal navigation, the agent is asked to navigate to a particular object in an unseen environment. This task has been further extended by introducing a social component: the presence of simulated human beings. The task of navigating in an environment with the presence of simulated people and finding an object-goal instance is called social object-goal navigation. Introducing a social component allows the agent to face more complex situations such as avoiding humans and, more in general, coexisting with them.

Indice

Elenco delle figure

Elenco delle tabelle

Capitolo 1

Introduction

The advancement in fields like reinforcement learning, computer vision, and natural language processing have generated a growing interest in developing a general-purpose intelligence. To reach this goal there has been a shift in the paradigm in which an artificial agent is trained: from using static data (Internet AI) to one based on the interaction with a virtual environment (Embodied AI). This shift is justified by the "embodiment hypothesis" proposed by Linda Smith [34] as the idea that intelligence emerges in the interaction with the environment and as a result of the sensorimotor activity, this means that virtual robots should learn by seeing, moving, speaking, and interacting with the world - just like humans. For now, embodied AI is about incorporating traditional intelligence concepts like vision, language, and reasoning into an agent to solve AI problems in a virtual environment.

The main problems studied in this scenario are:

- Visual exploration, where the agent has to gather information about a 3D environment in such a way as to update its internal model of the environment.
- Visual navigation. In this task, the agent navigates to a particular goal without any natural language instruction. The goal of this task can be very different, it can be a point, an object, or a region.
- Embodied Question answering where the agent must answer to a request performed in natural language by the user.

Now that we have defined the settings we need to specify how an agent can be trained to perform these tasks successfully. All the problems that we have seen require an environment where the agent can explore, navigate and interact, and this environment can be given by a simulator. A simulator is a virtual environment replicating the real world in which the agent can be trained. These simulators also facilitate the collection of task-related datasets, which are tedious to collect in the real world and require a lot of human work. There are many simulators like DeepMindLab,AI2-THOR , CHALET, VirtualHome, VRKitchen. Each simulator can be used for different tasks based on its technical properties. For example, if an agent has to perform a simple navigation task then a simulator with a basic physic engine will be enough. If instead, an agent has to learn to manipulate complex objects (e.g. clothes) then we need to use a simulator equipped with more advanced physics capabilities. In this thesis, I've used habitat-sim as the simulator, a flexible, high-performance 3D simulator with configurable agents, multiple sensors, and generic 3D dataset handling. In this context, there are multiple

ways of solving the discussed problems. If we take into consideration the problem of visual navigation we can distinguish between two types of solution: the so-called "map-based" methods that depend on geometry information (e.g SLAM methods for mapping and localization) or "learning-based" methods that are the ones that will be discussed in this work. Unlike map-based methods, which measure and calculate geometry information explicitly, "learning-based" methods take advantage of deep learning techniques, with an implicit understanding of the task and of the environment through the usage of data. These methods can exploit photo-realistic level simulators, and agents that rely on these techniques are generally trained using a reinforcement learning algorithm. Many algorithms can be used to train a reinforcement learning agent, an example can be Proximal Policy Optimization (PPO) or distributed Proximal Policy Optimization.

Thesis experiments and studies The goal of this thesis is to try to tackle object-goal naviagation and social object-goal navigation task. object goal navigation is an emboedied AI task where an agent navigates inside an evnironment and it has to find an instance of an object-goal (e.g. a chair). Social object-goal naviation is the same task where the social component is represented by the presence of simulated people who move inside the environment. In this context, the agent has to find the object-goal without colliding with any person.

The agent that we have studied in this thesis uses the so called auxiliary tasks. Auxiliary tasks are unsupervised tasks that an agent has to learn at training time. This taks help the agent to develop different capabilites that help it to navigate inside the environment and avoid obstacles (e.g. the auxiliary task called "proximity compass" helps the agent to understand the people's position inside the environment).

The experiments study how the agent behaves in object-goal navigation and social object-goal navigation task. Moreover, we studied how the auxiliary tasks introduction change the agent performances and behavior.

Capitolo 2

Background knowledge

In this section, I will introduce and explain some of the concepts that will appear in this thesis. I will cover neural networks, in particular, convolutional neural networks and recurrent neural networks, and I will introduce some general concepts on what is reinforcement learning and state-of-the-art algorithms used in this field.

2.1 Neural network

As we will see in the next chapters, the kind of machine learning technique that I've used to train my agent is reinforcement learning. The main goal of a reinforcement learning agent is to find a policy $\pi(s)$, where $\pi : S \rightarrow A$ is a function that maps each state s to a particular action a. One of the most important properties of this function is that it has to maximize some reward that the agent receives at each step. Nowadays, the function π is approximated using a neural network. In general, the goal of a neural network is to approximate some function f. For example, the function π in the case of reinforcement learning problem or, for a classifier, the function $y = f(x)$ that maps each input to a particular category y. Two kinds of neural networks are used in this thesis: Convolutional neural network and Recurrent neural network.

2.1.1 Convolutional neural network

Convolutional neural networks, or CNNs, are a specialized kind of neural network for processing data that has a known grid-like topology. The most commonly used inputs for this kind of network are images. As the name suggests, the network employs a mathematical operation called convolution. We can define a CNN as a simple neural network that uses convolution in place of general matrix multiplication in at least one of its layers.

Convolution Convolution is an operation on two functions of a real-valued argument. The first argument to the convolution is referred to as the input I, and the second argument as the kernel K. In machine learning applications, the input is usually a multidimensional array of data, and the kernel a multidimensional array of parameters that are adapted by the learning algorithm. So, for example, given a two-dimensional input and a 2-dimensional kernel, the convolutional operation between the two can be

formalized as:

$$S(i,j) = (I * K)(i,j) = \sum_m \sum_n I(m,n)K(i-m, j-n) \tag{2.1}$$

Many neural network libraries (including Pytorch) implement a related function called cross-correlation which is the same as convolution but without flipping the kernel:

$$S(i,j) = (K * I)(i,j) = \sum_m \sum_n I(i+m, j+n)K(m,n) \tag{2.2}$$

Properties There are many advantages to choosing a CNN over a standard feed-forward network, especially when we have to deal with data that has a known grid-like topology. These advantages are given by three properties that distinguish a CNN from other kinds of architecture:

- Sparse interactions
- Parameter sharing
- Equivariant representations

Sparse interactions Traditional neural networks use dense layers. This means that there is a parameter of the input layer for each node of the next layer. If there are many layers or if each layer has a large number of hidden nodes then the final output computation could require a lot of time. CNNs reduce this complexity drastically because they have sparse interactions. This appears because the kernel is different orders of magnitude smaller than the input image. This means that every single node of the neural network is not connected to all the other output nodes but only to a few of them, say k. And this reduces the number of parameters that we need to store and increase the statistical efficiency 2.1.

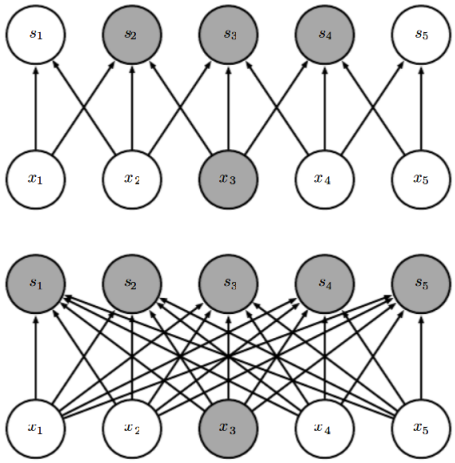

Figura 2.1: Sparse interaction in CNN. In this case the output s_3 is affected only on x_2, x_3, x_4 rather than all the inputs as in the case of standard feed-forward network [12]

Parameter sharing Parameter sharing refers to using the same parameter for more than one function in a model. In a traditional neural network, each weight is used exactly once when computing the output of a layer. In a CNN instead, each member of the kernel is used at every position of the input. So rather than learning a separate set of parameters for every location we learn only one set. Parameter sharing combined with sparse interaction makes CNNs dramatically more efficient than dense feed-forward networks in terms of memory requirements and statistical efficiency 2.2.

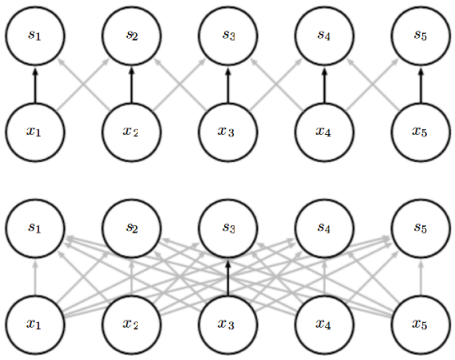

Figura 2.2: In CNN a parameter (black arrow) is used more than once. In standard feed-forward network is used only once [12]

Equivariant representations In the case of convolution, the particular form of parameter sharing causes the layer to have a property called equivariance to translation. This means that if the input changes, the output changes in the same way. Thanks to this property if we consider an image I and we move an object within I then the representation of that object will move the same amount in the output.

2.1.2 Recurrent neural network

Recurrent neural networks, or RNNs, are a family of neural networks for processing sequential data. RNNs take advantage of one of the early ideas found in machine learning: sharing parameters across different parts of a model. This technique makes it possible to extend and apply the model to examples of different lengths and generalize across them. So, at each time step of the sequence, the RNN shares the same weights.

Architecture As said previously, RNNs are neural networks suitable to process sequences of data. Consider the form of a dynamical system:

$$s^{(t)} = f(s^{(t-1)}; \theta) \tag{2.3}$$

where s^t is called the state of the system. This equation is recurrent because the definition of s at time t refers back to the same definition at time $t-1$. In order to approximate a function that involves recurrence, just like f, recurrent neural networks define the values of their hidden units in this way:

$$h^{(t)} = f(h^{(t-1)}, x^t, \theta) \tag{2.4}$$

where h is the state of the RNN. In general, we can think as $h^{(t)}$ as a lossy summary of the task-relevant aspects of the past sequence of inputs up to t.

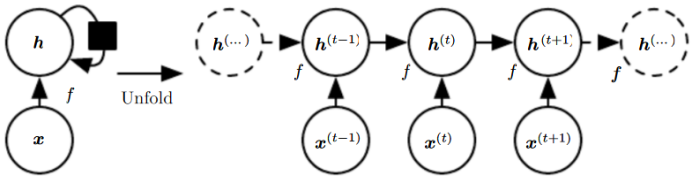

Figura 2.3: RNN with no outputs. This RNN processes information from the input x by incorporating it into the state h that is passed forward through time.

The forward propagation equations for a shallow RNN are applied for each time step from $t = 1$ to $t = \tau$ and are given by:

$$a^{(t)} = b + Wh^{(t-1)} + Ux^{(t)} \tag{2.5}$$

$$h^{(t)} = f(a^{(t)}) \tag{2.6}$$

$$o^{(t)} = c + Vh^{(t)} \tag{2.7}$$

$$\hat{y}^{(t)} = softmax(o^{(t)}) \tag{2.8}$$

where U, V and W are learnable parameters. b and c are biases and $\hat{y}^{(t)}$ is the output of the network at time step t.

Learning process There is no specific learning algorithm to learn using RNNs, we can just apply back-propagation to the unrolled computational graph. This kind of use of the back-propagation algorithm is called back-propagation through time. The problem with this approach is that gradient propagated over many stages tends to vanish (most of the time) or explode, and this aspect affects the capability of the network to learn long-term dependencies through sequences of data. In order to avoid the vanishing of the gradient two RNN architectures have been developed: Long Short Term Memory (LSTM) [21] model and Gated Recurrent Unit (GRU) model [21].

In order to solve the vanishing gradient problem Gate Recurrent Unint networks (GRU) use the update gate and the reset gate, basically two vectors which decide what information should be passed to the output. The update gate z_t for the time step t is computed using the following formula:

$$z_t = \sigma(W^{(z)}x_t + U^{(z)}h_{t-1}) \tag{2.9}$$

where x_t is the input at time t and it is multiplied by the weight matrix $W^{(z)}$. h_{t-1} holds the information from the past and it is multiplied by the weight matrix $U^{(z)}$.

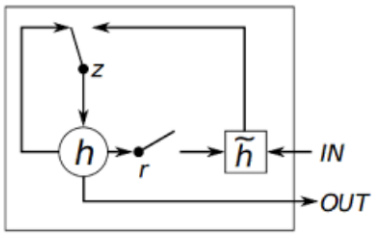

Figura 2.4: Gated Recurrent Unit network. z is the update gate and r is the reset gate.

This gate helps the model to determine how much of the past information needs to be passed along to the future. The other gate used by the GRU model is the reset gate, defined by the following formula:

$$r_t = \sigma(W^{(r)}x_t + U^{(r)}h_{t-1}) \tag{2.10}$$

As we can see the only difference respect to the update gate are the matrixes of the weights applied to x_t and h_{t-1}. This gate helps the network to decide how much of the past information to forget.
Both these gates are used to compute the final memory content of the current time step.
First of all the reset gate is used to discard irrelevant information from the past. For this purpose $\hat{h_t}$ is introduced and it is the vector that store the relevant memories form the previous steps:

$$\hat{h_t} = tanh(Wx_t + r_t \odot Uh_{t-1}) \tag{2.11}$$

In particular the operation $r_t \odot Uh_{t-1}$ is the element-wise product that determines what to remove from the previous step h_{t-1} and keeps only the useful information.
Finally, the model computes h_t which is the vector that holds information for the current unit and passes it down to the network:

$$h_t = z_t \odot h_{t-1} + (1 - z_t) \odot \hat{h}_t \tag{2.12}$$

In this case, the update gate determines what to collect from the current memory content ($\hat{h_t}$) and what from the previous step (h_{t-1}). Notice that each element of both the vectors r_t and z_t are elements between zero and one. In this way, they work as a scale factor for the elements of the vector for which they are multiplied, in our case h_{t-1} and $\hat{h_t}$ and this allows the GRU to discard or retain certain elements of the past.

2.1.3 Semantic segmentation and RedNet

To achieve success in an embodied AI task, having efficient models that perform well in "Internet AI" tasks is crucial. Semantic segmentation is one of the most significant Internet AI tasks and is crucial for achieving successful embodied AI tasks such as object-goal navigation. In object-goal navigation, the agent's objective is to navigate through an environment and locate an example of an object category. To locate such an instance, it must be distinguished and identified from other objects belonging to different categories. The process of categorizing each object and determining its boundaries is known as semantic segmentation. This involves assigning a class label to each pixel of an input image, which allows the agent to generate a mask and detect the borders and category of a specific object.

Another type of segmentation, known as instance segmentation, distinguishes between different objects rather than classes 2.6. In this thesis, I am particularly interested in semantic segmentation. The objective is to identify a specific instance of an object category; therefore, the agent's success is not dependent on which instance of the object category it finds. The agent's semantic segmentation was derived from the dataset's ground truth information during training but during its evaluation, the semantic information is derived from a specialized network known as RedNet (Residual Encoder-Decoder Network for indoor RGB-D Semantic Segmentation) [22].

RedNet

RedNet 2.5 is a neural network for indoor semantic segmentation based on the encoder-decoder structure. The encoder-decoder architectures have a downsample path to extract semantic information from the images, and an upsample path to recover a full-resolution semantic segmentation mask. This structure allows a lower memory consumption during the training steps, and so also a deeper structure. In order to make the predictions, RedNet uses both the RGB and the depth information as inputs, to overcome the problem of similarity in colors and structure of objects placed in the indoor environment.

Network description The network comprises two main parts: the encoder that extracts the semantic information from the image and the decoder that builds the actual semantic mask. The encoder is composed of two branches, one that processes the RGB image and the other that processes the depth information. The only difference between these two branches is the number of channels since depth images are composed of just one channel, and RGB images are composed of three channels. Each branch starts with two downsample operations using a convolutions operation with a 7 x 7 kernel and a 3 x 3 max pooling layer. The rest of the encoder part comprises a series of residual layers, the building block of the Residual network [20]. In brief, the Residual network overcomes the problem of degrading performances as the number of CNN's layers increases. This problem was solved by adding shortcut connections to merge the desired residual mapping of an input x with the identity input (generally through a summation), allowing an easier optimization and better exploitation of the number of layers. In the encoder part of the Rednet architecture, the author uses 3 resNet layers that downsample the feature map and increase the number of channels. In the end, the two branches are fused using element-wise operation as a feature fusing method. The decoder of the network is composed almost exclusively of residual layers that upsample the feature map by a factor of 2. The final block has a single 2 x 2 transpose of the convolutional layer to build the final semantic map.

Training and loss function A common problem for very deep networks is represented by the vanishing of the gradient during the training. To overcome this issue the author proposed a pyramid scheme where each upsample ResLayer outputs an intermediate output from its feature map, for a total of five outputs (including the final one). For each output, the cross entropy loss is computed using the soft-max function:

$$L(s, g) = \frac{1}{N} \sum_i -log(\frac{exp(s_i[g_i])}{\sum_k exp(s_i[k])}) \tag{2.13}$$

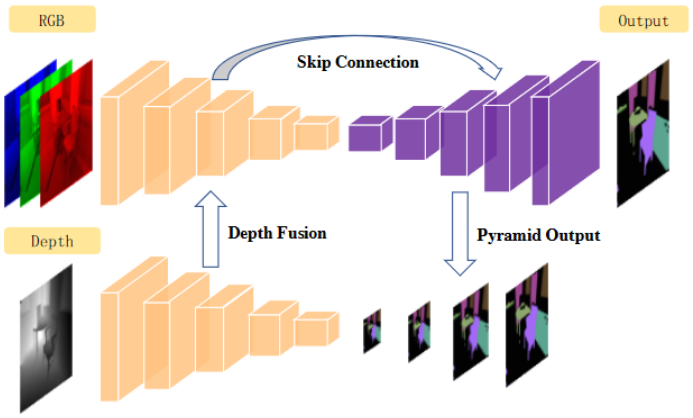

Figura 2.5: High level view of the RedNet architecture [22]. We have the encoder for the depth and rgb data (yellow) and the decoder to build the actual semantic map (purple)

Where g_i denotes the class index on the ground truth semantic map at location i, s_i denotes the final output of the network at location i and N is the spatial resolution of the input.

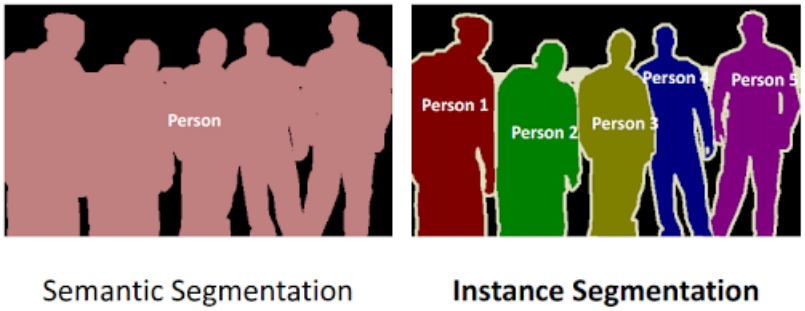

Figura 2.6: As we can see from the image the color of the people in case of semantic segmentation is the same (the belong to the same class). In instance segmentation the colors are all different (they are different people)

2.2 Reinforcement learning

Reinforcement learning is a subfield of machine learning that deals with how an agent can learn to make decisions based on feedback from its environment. In this paradigm, an agent interacts with an environment in a sequence of discrete time steps where, at each step, it receives an observation from the environment and selects an action to perform. The environment responds by generating a reward signal, which reflects how good or bad the action selected was, and a new observation. The goal of a reinforcement learning agent is to learn a policy, that is a mapping from states to actions, that maximize the expected cumulative reward.
We can also describe reinforcement learning (RL) as the science of taking actions in

a particular environment. Reinforcement learning is based on the reward hypothesis: any goal can be formalized as the outcome of maximizing the cumulative reward. Reinforcement learning problems are characterized by three features that make them different from all other machine-learning problems [36]:

- They are closed-loop problems, in the sense that the learning system's action influences its later inputs.
- The learner is not told which action to take, as in any form of machine learning, but instead must discover which actions yield the most reward by trying them out.
- Action not only affects the immediate reward but also the next situation and, through that, all subsequent rewards.

Thanks to these three characteristics we can understand why reinforcement learning problems cannot be solved using the other two machine learning paradigms: supervised learning and unsupervised learning.

Supervised learning is learning from a training set of labeled examples, each example is a description of a situation with a specification, the label. But alone this paradigm is not suited for learning from interaction. In an interactive environment is difficult to obtain examples of desired behavior that are both correct and representative of all the situations in which the agent has to act.

Reinforcement learning is also different from the unsupervised learning paradigm. Unsupervised learning is about finding hidden structures in unlabeled data, but this feature, although it can be useful for RL problems does not address the need for an RL agent to maximize the reward signal.

Reinforcement learning has different challenges that other paradigms don't have. One of the most famous challenges is the exploration-exploitation trade-off. Basically, this problem is about balancing the selection of an action that returns the highest cumulative reward based on the current knowledge about the environment (exploitation) and the selection of actions that might lead to a higher reward in the future (exploration). Undiscovered actions might lead to higher rewards compared to the current best action but can also lead to worse states and so, to a lower reward.

The dilemma is that neither exploration nor exploitation can be pursued exclusively without failing at the task. The agent must try a variety of actions and progressively favor those that appear to be best.

In general, in a reinforcement learning algorithm, we can identify 4 main subelements: policy, reward signal, value function, and a model of the environment [36].

- The policy π is the core of the reinforcement learning agent. It is the function that maps each state to a particular action. In some cases can be a simple lookup table (tabular algorithm) like in the q-learning algorithm. In these cases the lookup table t can be represented as a 2D array of dimension $state\_space$ $\times$action_space$where each entry t(i,j) is the value action of taking action j at state i. This value is more formally c$ is a function then this function can be built using different methods. In the last years, one of the most popular methods to build the policy is to use a

machine learning model, in particular a neural network. The usage of deep neural networks as a policy gave birth to deep reinforcement learning. For this reason, the majority of policies used in deep reinforcement learning algorithms are stochastic.

- The reward signal defines the goal of a reinforcement learning algorithm. From its perspective, the agent doesn't really know the goal of the problem and the only things that it receives as feedback for its action are the rewards. For this reason the engineering of the reward signal is critical for the agent to be able to solve the problem correctly. Consider the scenario where we want to train an agent to play chess. We can formulate the problem like this:

  – For every move done by the agent or done by the opponent, the agent receives a neutral reward (zero)
  – If the agent wins it gets a reward of +1
  – If the agent loses it gets a reward of -1

  If we design the reward signal in this way then the agent has the incentive to win because it would get a higher reward compared to the scenario where it loses. The problem is that it is a sparse reward (the agent gets feedback only when the game finishes), so it doesn't know when a certain move was bad. We need to find a way to give the agent an informative reward during mid-game moves. For example, we can give it a positive reward each time an opponent's piece is removed from the board and a negative when one of its pieces is removed. At this point, we have to pay attention to the magnitude of each reward: if the reward of winning/losing the game is not high enough then the agent is encouraged to remove the opponent's pieces from the board instead of winning the game.

  The reward signal can also be used to add some constraints on how a particular problem must be solved. Consider a scenario where an agent is in a maze and has to learn to find the exit in the least amount of steps. We can add a -1 reward at each step and a high reward at the end of the maze. In this way, the agent is encouraged to exit from a maze as quickly as possible (with the least steps), in order to receive the least amount of negative rewards.

- In the previous point, we have introduced $q(s, a)$ and $v(s)$ which are the action and state value functions respectively. We can informally say that these functions embed the capacity of the agent to select actions by basing its evaluation not only on the immediate reward that an action a taken in a particular state s would give to an agent but also on future rewards.
  For example, a state might always yield a low immediate reward but still have a high value because it is regularly followed by other states that yield high rewards.

- The fourth element is a model of an environment. The model mimics the behavior of the environment. For example, given a state and action, the model might predict the resultant next state and next reward.

2.3 Markov decision process

In this chapter, I will introduce the formal problem of the finite Markov decision process (MDP). MDPs are a classical formalization of sequential decision-making,

where actions influence not just immediate rewards, but also subsequent situations or states, and through those future rewards.
MDPs are a mathematically idealized form of reinforcement learning problem for which precise theoretical statements can be made. The setting is simple: an agent (the decision-maker) interacts with a certain environment through some actions, and the environment responds to these actions with new situations to the agent (state) and with rewards, special numerical values that the agent seeks to maximize over time through its choice of actions 2.7. More specifically, the agent and environment interact

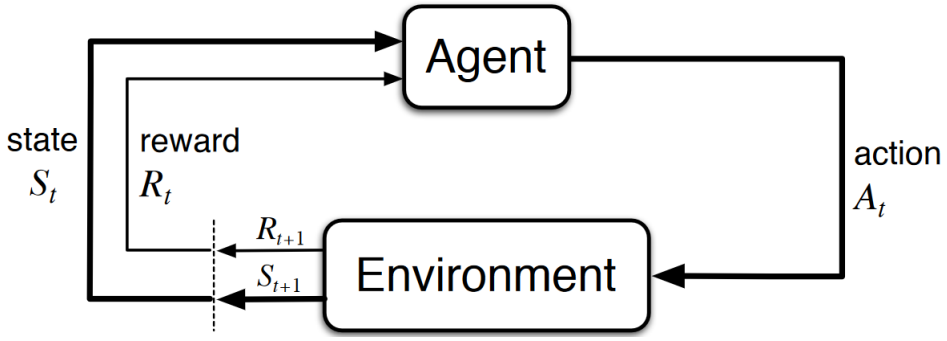

Figura 2.7: Reinforcement learning loop. The agent, given a state S_t, selects an action A_t, the environment, faced with this action, responds with a reward R_{t+1} and a new state S_{t+1} [36]

at each of a sequence of discrete time steps, $t = 0, 1, 2, 3, ...$ At each time step t, the agent receives some representation of the environment's state, $S_t \in S$ and, on that basis, selects an action, $A_t \in A$. One time step later, as a consequence of its action, the agent receives a numerical reward, $R_{t+1} \in R$, and finds itself in a new state, S_{t+1}.
We can formalize a Markov Decision Process as a tuple (S, A, p, r, γ), where:

- S is the set of all possible states
- A is the set of all possible actions
- $p(s', r|s, a)$ is the probability of transitioning to s' , and get a reward r given a state s and action a. The function p defines the dynamics of the MDP.
- $r : SXA \to \mathbb{R}$ is the expected reward, achieved on a transition starting in (s, a).

$$r = \mathbb{E}[R|s, a] \tag{2.14}$$

- $\gamma \in [0, 1]$ is a discount factor that trades off later rewards to earlier ones.

In a Markov decision process, the probabilities given by p completely characterize the environment's dynamics. That is, the probability of each possible value for S_t and r_t depends on the immediately preceding state and action, s_{t-1} and a_{t-1}, and, given them, not on earlier states and actions. The state must include information about all aspects of the past agent-environment interaction that make a difference for the future. If it does, then the state is said to have the Markov property. More formally we say

that:
A state s has the Markov property when $\forall s' \in S$

$$p(S_{t+1} = s'|S_t = s) = p(S_{t+1} = s'|h_{t-1}, S_t = s) \tag{2.15}$$

for all possible histories $h_{t-1} = \{S_1, ..., S_{t-1}, A_1, ..., A_{t-1}, R_1, ..., R_{t-1}\}$.
In the Markov Decision Process, all states are assumed to have the Markov property.

2.3.1 Formalising the objective

In general, the objective of the problem is to maximize the expected return, where the return, denoted G_t, is defined as:

$$G_t = R_{t+1} + R_{t+2} + ... + R_T \tag{2.16}$$

where T is the final step. This approach makes sense in applications in which there is a natural notion of the final time step, that is, when the agent-environment interaction breaks naturally into subsequences, called episodes. Each episode ends in a special state called terminal state. On the other hand, in many cases, the agent-environment interaction does not break naturally into identifiable episodes, but go on continually without limit. We call these continuing tasks. In this case the formulation of G_t is problematic because the final time step would be $T = \infty$, and the return, which is what we are trying to maximize, could easily be infinite. In this case, we need discounting. In particular, the agent chooses A_t to maximize the expected discounted return:

$$G_t = R_{t+1} + \gamma R_{t+1}, \gamma^2 R_{t+3} + ... = \sum_{k=0}^{\infty} \gamma^k R_{t+k+1} \tag{2.17}$$

where $0 \leq \gamma \leq 1$ is the discount rate. The discount rate determines the present value of future rewards: a reward received k time steps in the future is worth only γ^{k-1} times what it would be worth if it were received immediately. The more γ gets closer to 0 the more the agent is "myopic" because it is concerned only with maximizing the immediate rewards (R_{t+1}). In general, this behavior reduces access to future rewards so that the return is reduced. As γ approaches 1, the return objective takes future rewards into account more strongly. Here the agent becomes more farsighted. Another important aspect is that the introduction of the discount factor avoids infinite returns in the cyclic Markov Process. In particular, if $\gamma < 1$, the infinite sum of G_t has a finite value as long as the reward sequence is bounded, and this solves the fact that if the horizon is infinite G_t might grow infinitely.

2.3.2 Policy and value function

The goal of an agent that tries to solve a Markov Decision Process problem is to find a behavior policy π that maximizes the expected return G_t. We can define a policy as a mapping $\pi : SXA \rightarrow [0, 1]$ that, for every state $s \in S$ assigns, for each action $a \in A$, the probability of taking the action a in the state S . Denoted as $\pi(a|s)$. To find a good policy many RL algorithms involve estimating value functions. A value function is a function that, given a state s as input (or a state-action pair), estimates how good it is for the agent to be in the state s. What does it mean "how good"? The notion of "how good" here is defined in terms of future rewards that can be expected or in terms of expected return.

$$v_\pi(s) = \mathbb{E}[G_t|S_t = s, \pi] \tag{2.18}$$

In the definition of v there is also the policy π because the rewards the agent can expect to receive in the future depend on what actions it will take and so, value functions are defined for a particular way of acting, the policy.

Similarly, we define the value of taking action a in state s under policy π, denoted $q_\pi(s, a)$, as the expected return starting from s, taking the action a, and thereafter following policy π:

$$q_\pi(s, a) = \mathbb{E}_\pi[G_t|S_t = s, A_t = a] \tag{2.19}$$

q_π is called the action-value function for policy π. There is a relationship between the functions v and q:

$$v_\pi(s) = \sum_a \pi(a|s) q_\pi(s, a) = \mathbb{E}[q_\pi(S_t, A_t)|S_t = s, \pi], \forall s \tag{2.20}$$

v_π and q_π can be estimated from experience. For example an agent can keep for each state s that it visited, the average of the actual return that has followed that state. The error in the the estimated value of v_π becomes smaller and smaller as the number of visits increases.
A fundamental property of value functions is that they can be defined recursively:

$$\begin{aligned} &= \mathrm{E}[\mathrm{R}_{t+} + \gamma G_{t+1}|S_t = s, \pi] \\ &= \mathrm{E}[\mathrm{R}_{t+} + \gamma v_\pi(S_{t+1})|S_t = s, A_t \sim \pi(S_t)] \\ &= \textstyle\sum_a \pi(a|s) \sum_r \sum_{s'} p(r, s'|s, a)(r + \gamma v_\pi(s')) \end{aligned} \tag{2.21}$$

This property is crucial to find the exact value function v_π of the system. Moreover, this property forms the basis for several ways to compute, approximate, and learn v_π. (2.3.2) is the Bellman equation for v_π. It expresses a relationship between the value of a state and the values of its successor states. The actual value function v_π is the unique solution to its Bellman equation. In MDPs two important tasks are needed to be solved:

- Policy evaluation/prediction: given a policy π compute the value function v_π. To solve this problem we have different possibilities such as solving the system of linear equations given by the system, or using an iterative method for the larger systems.
- Control: finding the optimal way of behaving. Here to solve this problem we can use dynamic programming by leveraging the dynamics of the system, or, if it is unknown, we can sample and apply algorithms like Q-learning or Sarsa.

An important point is that in the control task, at least in the following methods, I am not going to learn directly the optimal policy (the function that maps each state to the action that gives the highest discounted reward) but I am going to learn the optimal state/action value function v^*_π/q^*_π, that is the state/action value function that corresponds to the optimal way of behaving. In this sense, the policy is fixed and deterministic: the greedy policy. The greedy policy is defined as:

$$\pi(s) = \operatorname*{argmax}_a v_\pi(S_{t+1}) \tag{2.22}$$

In the following section, I will briefly describe some methods in order to tackle these two problems. In general, we can divide these methods into two categories:

- Model-based solution: In this case to evaluate or update the policy we need the entire model of the system (e.g $p(s', r|s, a)$)
- Model-free solution: There is no need to have knowledge about the underlying MDP. In general, we solve the problem using samples.

2.4 Model-free prediction/control

In this section, I will briefly discuss model-free solutions. Unlike model-based solutions, here we do not assume complete knowledge of the environment. These methods require only experience by sampling sequences of states, actions, and rewards from actual or simulated interaction with the environment. So, by knowing that:

$$v_\pi(s) = \mathbb{E}[G_t|S_t = s](.) = \mathbb{E}[R_{t+1} + \gamma v_\pi(S_{t+1})|S_t = s] \tag{2.23}$$

With sampling methods, we don't know the dynamics of the environment and so we don't know the expected values that appear in 2.23. In this scenario, we can identify two types of methods:

- Monte-Carlo methods
- Temporal-difference learning (TD) methods

Monte-Carlo methods Monte-Carlo methods are ways of solving the reinforcement learning problem based on averaging sample returns. In this kind of methods, the value $v_\pi(s)$ is updated only at the end of the episode (after the computation of the average return rewarded starting from the state s). The updating formula for $v_\pi(s)$ is:

$$v_\pi(s) = v_\pi(s) + \alpha[G_t - v_\pi(s)] \tag{2.24}$$

Where G_t is the reward at time step t.

The problem with this approach is the high return variance and the fact that the state is updated only at the end of each episode.

Temporal-difference learning methods In TD learning, like Monte Carlo methods, the agent learns from sampling without the need for a model of the environment's dynamics. The difference is that, instead of updating the value $v_\pi(s)$ at the end of the episode, it is updated at each step and its estimation is based in part on other learned estimates, without waiting for the final outcome. This process is called bootstrapping. In this case, the updating formula for $v_\pi(s)$ is:

$$v_\pi(s) = v_\pi(s) + \alpha[R_{t+1} + \gamma v_\pi(s_{t+1}) - v(s_t)] \tag{2.25}$$

$R_{t+1} + \gamma v_\pi(s_{t+1})$ is called TD-target and $R_{t+1} + \gamma v_\pi(s_{t+1}) - v_\pi(s)$ is the TD error. This approach trades a higher bias with a lower variance and it is generally a faster method compared to Monte-Carlo.

2.4.1 Temporal-difference algorithms

In this section, I will briefly discuss temporal-difference control algorithms. An important classification can be made for these kinds of methods: On-policy methods and Off-policy methods.

On-policy vs Off-policy methods

The difference between these approaches is subtle but significant:

- In On-policy methods we use the same policy for acting and updating.
- In Off-policy methods we use a different policy for acting and updating.

The main advantage of Off-policy methods is that they allow to learn a policy different respect to the one used to select an action. This characteristic helps to address the exploration-exploitation trade-off. An example of an On-policy algorithm is Sarsa while an example of an Off-policy algorithm is Q-learning. In the following sections a brief description of both algorithms.

2.4.2 Sarsa

Sarasa is an On-policy temporal-difference algorithm used to approximate the state-value function $Q_\pi(s)$. In Sarsa, the update rule for each state is the following:

$$Q(S_t, A) = Q(S_t, A) + \alpha[R_{t+1} + \gamma Q(S_{t+1}, A_{t+1}) - Q(S_t, A)] \tag{2.26}$$

As described in figure 2.8. Since Sarsa is an On-policy algorithm the action A and

Sarsa (on-policy TD control) for estimating $Q \approx q_*$

Algorithm parameters: step size $\alpha \in (0, 1]$, small $\varepsilon > 0$
Initialize $Q(s, a)$, for all $s \in \mathcal{S}^+, a \in \mathcal{A}(s)$, arbitrarily except that $Q(terminal, \cdot) = 0$

Loop for each episode:
  Initialize S
  Choose A from S using policy derived from Q (e.g., ε-greedy)
  Loop for each step of episode:
    Take action A, observe R, S'
    Choose A' from S' using policy derived from Q (e.g., ε-greedy)
    $Q(S, A) \leftarrow Q(S, A) + \alpha\big[R + \gamma Q(S', A') - Q(S, A)\big]$
    $S \leftarrow S'$; $A \leftarrow A'$;
  until S is terminal

Figura 2.8: Sarsa algorithm [36].

A_{t+1} are selected using the same policy, in the algorithm they use $\epsilon - greedy$ policy (a policy that selects with a probability p the greedy action and with a probability $1 - p$ a random action). Sarsa converges with probability 1 to an optimal policy and an optimal action-value function as long as all state-action pairs are visited an infinite number of times, so the convergence properties of the Sarsa algorithm depend on the nature of the policy that we use to select the next state S_{t+1}.

2.4.3 Q-learning

Q-learning is, just like Sarsa, a temporal-difference algorithm but is an Off-policy method. In Q-learning the update rule for each state-action pair is:

$$Q(S_t, A) = Q(S_t, A) + \alpha[R_{t+1} + \gamma \max_a Q(S_{t+1}) - Q(S_t, A)] \tag{2.27}$$

As also depicted in 2.9. As we can see in 2.27 and in 2.9 the agent selects the next action A with an exploration policy $\epsilon-greedy$ policy, but the TD-target is computed by taking into consideration the maximum state-action pair value among all possible actions a of the state S_{t+1}. In other words the TD-target is $R_{t+1} + \gamma \max_a Q(S_{t+1})$, allowing the Q-learning algorithm to learn action-value function Q that directly approximates Q^*, the optimal action-value function, independently of the policy being followed to explore the environment.

Q-learning (off-policy TD control) for estimating $\pi \approx \pi_*$

Algorithm parameters: step size $\alpha \in (0, 1]$, small $\varepsilon > 0$
Initialize $Q(s, a)$, for all $s \in \mathcal{S}^+, a \in \mathcal{A}(s)$, arbitrarily except that $Q(terminal, \cdot) = 0$

Loop for each episode:
  Initialize S
  Loop for each step of episode:
    Choose A from S using policy derived from Q (e.g., ε-greedy)
    Take action A, observe R, S'
    $Q(S, A) \leftarrow Q(S, A) + \alpha\big[R + \gamma \max_a Q(S', a) - Q(S, A)\big]$
    $S \leftarrow S'$
  until S is terminal

Figura 2.9: Q-learning algorithm [36].

2.5 Policy gradient method

Up to now, we have seen methods that compute the action-value function/state-value function in order to find the optimal behavior of the agent. In these cases, the actual policy of the agent is a function that selects the action based on the learned action-value function (usually it is the greedy policy). With policy gradient methods there is a change of paradigm in the sense that now the goal of the algorithm is to find a parametrized policy that can select the action without consulting any action-value function, but it selects a certain action a based on a certain probability $P(a|s;\theta)$. Where θ are the parameters of the policy. The goal of the algorithm is to maximize the performances, so the update of the parameters is done by using the gradient ascent algorithm, where the gradient is based on some scalar performance measure $J(\theta)$ with respect to the policy parameter:

$$\theta_{t+1} = \theta_t + \alpha\widehat{\nabla J(\theta_t)} \tag{2.28}$$

In other words, we want to control the probability distribution of actions by tuning the policy such that good actions (actions that maximize the return) are sampled more frequently in the future. This approach offers two main advantages with respect to methods that learn the action-value function:

- We don't need to store additional data (action values)
- Policy gradient methods can learn stochastic policy while value functions can't

In particular, with the second point, we have two important consequences :

- Since the policy is stochastic we don't have to manually implement an exploration-exploitation strategy, because the trajectory that we follow is not always the same
- We also get rid of the problem of perceptual aliasing, which is the problem when two states are the same but they need different actions.

2.5.1 Advantage Actor-critic method

One of the most used policy gradient method algorithms is Reinforce. Reinforce is an on-policy method that optimizes the policy directly without using a value function. Reinforce updates its policy using Monte-Carlo sampling (so it uses the reward at the end of the episode) and this led to a high variance in policy gradient estimation. A solution to this problem is given by a family of learning algorithms called Advantage Actor-Critic (A2C). The basic idea behind these kinds of algorithms is that we combine policy algorithm and value base algorithm, by learning:

- A policy that controls how the agent acts: $\pi_\theta(s, a)$ (parametrized by θ)
- A value function to assist the policy update by measuring how good the action taken is: $\hat{q}_w(s, a)$ (parametrized by w)

In this context, the policy is called the actor and the value function is the critic. To explain these algorithms I will briefly describe the training process of a generic Advantage actor-critic learning algorithm:

- The actor receives in input the state S_t and it outputs the action A_t while the critic receives in input both the state S_t and the action A_t to compute the advantage $A(s, a)$. The advantage function is really useful to stabilize the learning algorithm and it describes how good is the action A_t taken in the state S_t compared to all the other possible actions that the agent could have taken in that state, and it is defined as follows:

$$\begin{aligned} A(s, a) &= Q(s, a) - V(s) \\ &= r + \gamma V(s^{'}) - V(s) \end{aligned} \tag{2.29}$$

  Where $Q(s, a)$ is the action-value function that measures how good is the action a in the state s 2.19. $V(s)$ is the value function of the state s 2.18.

- The actor updates its policy using the following formula:

$$\Delta\theta = \alpha\nabla_\theta(log\pi_\theta(s, a))A(S_t, A_t) \tag{2.30}$$

  From this equation, we can see that if $A(s, a)$ is positive then A_t gives a higher reward than the mean of the other actions in S_t and so the gradient is pushed in that direction. If $A(S_t, A_t)$ is negative then the gradient is pushed in the opposite direction because A_t performs worse than the mean of all the other actions in S_t. After the update of θ the actor outputs the next action A_{t+1} for the next state S_{t+1}.

- Finally, also the critic updates its parameter w by using the following formula:

$$\Delta w = \beta(R(s,a) + \gamma \hat{q}_w(S_{t+1}, A_{t+1}) - \hat{q}_w(S_t, A_t))\nabla_w \hat{q}_w(S_t, A_t) \tag{2.31}$$

We can recognize the first part of this equation as the TD-Error of $\hat{q}_w(S_t, A_t)$.

Trust Region Policy Optimization (TRPO)

Theoretical foundations The problem of policy gradient optimization is related to the size of the update step. If the step is too small then the policy will learn too slowly but a large step can lead to a disaster. Moreover In reinforcement learning, the data is generated by the policy itself. In this context, data generated from a policy that undergoes significant changes at each iteration can vary considerably, making the problem more challenging due to increased nonstationarity.

The idea behind TRPO [33] is to use a surrogate objective function in order to constrain the next policy update. As we will see this is the concept also used in Proximal Policy Optimization(PPO).
The work proposed by [24] derived the following result:

$$\eta(\pi_{\text{new}}) \geq L(\pi_{\text{new}}) - \frac{2\epsilon\gamma}{(1-\gamma)^2\alpha^2} \quad \text{where } \epsilon = \max_s \left[\mathbb{E}_{a\sim\pi'(a|s)} A_\pi(s,a)\right] \tag{2.32}$$

Where $\eta(\pi_{\text{new}})$ is the expected reward of the policy π_{new}, $L(\pi_{\text{old}}, \pi_{\text{new}})$ is equal to

$$L(\pi_{\text{old}}, \pi_{\text{new}}) = \eta(\pi_{old}) + \sum_s \rho_{\pi_{old}(s)} \sum_a \pi_{new}(a|s) A_\pi(s,a) \tag{2.33}$$

π_{new} is

$$\pi_{\text{new}}(a|s) = (1-\alpha)\pi_{\text{old}}(a|s) + \alpha\pi_0(a|s). \tag{2.34}$$

2.32 implies that a policy update that improves the right-hand side is guaranteed to improve the true performance η. The authors in the paper used this last result as the foundation for TRPO. In particular they adapted 2.32 with the following formula:

$$\eta(\tilde{\pi}) \geq L_\pi(\tilde{\pi}) - CD_{KL}^{\max}(\pi, \tilde{\pi}) \tag{2.35}$$

Where $C = 4\epsilon\gamma(1-\gamma)^2$,. $D^{\max}\text{KL}(\pi, \tilde{\pi})$ is the Kullback–Leibler divergence, which measures how much the probability given by the policy π is different from the probability given by the policy $\tilde{\pi}$.

Algorithm By maximizing the right-hand side of 2.32, we are guaranteed to improve the true objective η:

$$\max_\theta \left[L_{\theta_{\text{old}}}(\theta) - CD_{KL}^{\max}(\theta_{\text{old}}, \theta)\right], \tag{2.36}$$

Instead of using the parameter C to regulate the step size, we can define a constraint δ on the KL divergence between the new policy and the old policy:

$$\begin{aligned} &\text{maximize}_\theta \quad L_{\theta_{\text{old}}}(\theta) \\ &\text{subject to} \quad D^{\max}\text{KL}(\theta_{\text{old}}, \theta) \leq \delta \end{aligned} \tag{2.37}$$

In this way, we are not limited by the small steps provided by the constant C given by theory, and still provide robust (but larger) steps.

The average KL divergence between the two policies is given by a heuristic:

$$D_{KL}^{\rho}(\theta_1, \theta_2) := \mathbb{E}_{s\sim\rho}\left[D_{KL}(\pi_{\theta_1}(\cdot|s) \,\|\, \pi_{\theta_2}(\cdot|s))\right] \tag{2.38}$$

The final update is then given by solving the following optimization problem:

$$\begin{aligned} &\text{maximize} && L_{\theta_{\text{old}}}(\theta) \\ &\text{subject to} && D_{KL}^{\rho\theta_{old}}(\theta_{\text{old}}, \theta) \leq \delta \end{aligned} \tag{2.39}$$

Proximal Policy Optimization

The Proximal Policy Optimization algorithm (PPO) is the evolution of TRPO. The advantages given by PPO with respect to TRPO are:

- PPO gives better performances
- PPO is easier to implement with respect to TRPO

The goal is the same: we want to learn a parametrized policy, without changing too much the current policy from one update to another. In TRPO the idea was to solve an optimization problem with a constraint on the Kullback–Leibler divergence between the current and the new policy.

In PPO the approach is a little bit different, instead of constraining the Kullback–Leibler divergence the evaluation of the new policy and the old policy difference is given by the ratio of their respective probabilities of taking a particular action a in a state s.

This is the idea behind Proximal Policy Optimization (PPO), the core algorithm I've used in this thesis. To not perform big policy updates the function that the algorithm optimizes is:

$$L^{CLIP}(\theta) = \hat{\mathbb{E}}_t[min(r_t(\theta)\hat{A}_t, clip(r_t(\theta), 1-\epsilon, 1+\epsilon)\hat{A}_t)] \tag{2.40}$$

Where r_t is the ratio function:

$$r_t(\theta) = \frac{\pi_\theta(a_t|S_t)}{\pi_{\theta_{old}}(a_t|S_t)} \tag{2.41}$$

And π_θ is the probability of taking action A_t at state S_t in the current policy, while $\pi_{\theta_{old}}$ is the probability of the previous policy. In this function there are two cases:

- If $r_t(\theta)$ is greater than one then the action A_t at state S_t is more likely in the current policy than the previous one.
- If $r_t(\theta)$ is between zero and one then the action a_t at state S_t is less likely in the current policy than the previous policy.

To avoid big policy updates we need to constrain the objective function by penalizing changes that lead to a ratio too large (or too small). To do that in 2.40 is introduced the clip probability ratio directly in the objective function (Clipped surrogate objective function), which is simply a clip between $1+\epsilon$ and $1-\epsilon$ of the ratio function $r_t(\theta)$. If we take the minimum between the clipped and non-clipped objective, we force the update of the policy to stabilize between the range $[1+\epsilon, 1-\epsilon]$. This stabilization is achieved because the algorithm ignores all those updates that would diverge the

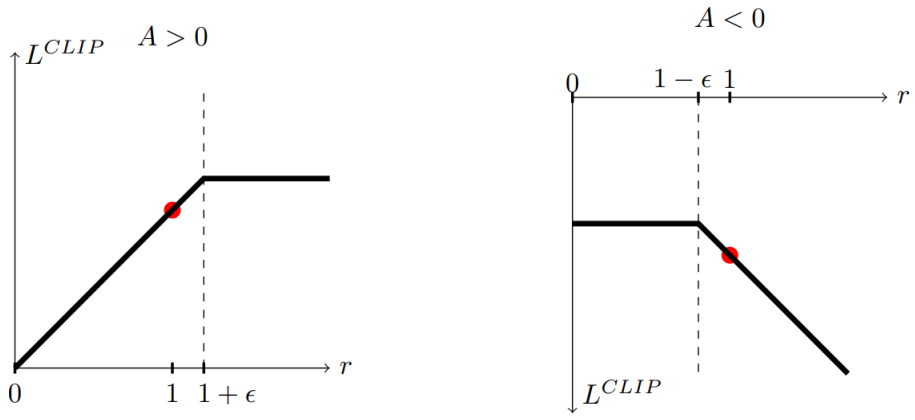

Figura 2.10: PPO objective function if the advantage is positive (left) or negative (right)

current policy π_θ from the old policy $\pi_{\theta_{old}}$. Consider the image 2.10. Here we can see the clipped surrogate objective function in the case where the advantage A is positive (left) and in the case where it is negative (right). In both cases, the function is flat after a certain threshold ($1+\epsilon$ if $A > 0$, $1-\epsilon$ otherwise). This means that if the function reaches this region, and it reaches this region only if the update of π_θ respect to the previous policy is greater than $1+\epsilon$ (if $A > 0$) or $1-\epsilon$ (if $A < 0$), the gradient is zero and so the policy is not updated. The policy might be updated also when the ratio is outside the range, but the advantage leads to getting closer to the range. This happens when:

- The ratio $r_t(\theta)$ is below the range but the advantage is positive.
- The ratio $r_t(\theta)$ is above the range but the advantage is negative.

The final Clipped surrogate Objective Loss for the PPO Actor-Critic agent it's a combination of the Clipped Surrogate Objective function, Value Loss function, and Entropy bonus:

$$L_t^{CLIP+VF+S}(\theta) = \hat{\mathbb{E}}_t[L_t^{CLIP}(\theta) - c_1 L_t^{VF}(\theta) + c_2 S[\pi_\theta](S_t)] \tag{2.42}$$

Where c_1 and c_2 are coefficients, $L_t^{VF}(\theta)$ is the squared-error value loss, and the last term $S[\pi_\theta](S_t)$ is an entropy bonus to ensure sufficient exploitation.

Decentralized Distributed Proximal Policy Optimization

Reinforcement learning, even more than other machine learning fields like supervised learning, rely on significant numbers of training samples. Thus, scaling RL via multi/-node distribution is crucial in AI. The authors of [39] propose a simple, synchronous, distributed RL method that scales well, called Decentralized Distributed Proximal Policy Optimization (DD-PPO). In DD-PPO, each worker alternates between collecting experience in a resource-intensive and GPU-accelerated simulated environment and optimizing the model.

In reinforcement learning the dominant distribution paradigm is asynchronous, where a rollout worker collects experience and asynchronously sends it to the parameter server.

In other fields (e.g. supervised learning), the update is performed synchronously. As a general abstraction in supervised learning the distributed system is implemented as follows: at step k, worker n has a copy of the parameters, θ_n^k, calculates the gradient, $\partial\theta_n^k$, and updates θ via:

$$\theta_n^{k+1} = ParamUpdate(\theta_n^k, AllReduce(\partial\theta_1^k, ..., \partial\theta_N^k)) = ParamUpdate(\theta_n^k, \frac{1}{N}\sum_{i=1}^{N}\partial\theta_i^k) \tag{2.43}$$

Where $ParamUpdate$ is any first-order optimization technique, and $Allreduce$ is the operation that performs the mean over all copies of a variable and returns the result to all workers.
DD-PPO adapts (2.43) for reinforcement learning. The general procedure is the following: experiences are gathered (rollout) using $\pi_{\theta_n^k}$, then parameter-gradients ∇_θ are computed via any policy-gradient method (e.g PPO), and finally, there is a synchronization of all the gradients with the other workers and the model is updated:

$$\theta_n^{k+1} = ParamUpdate(\theta_n^k, AllReduce(\nabla_\theta J^{PPO}(\theta_1^k), .., \nabla_\theta J^{PPO}(\theta_N^k))) \tag{2.44}$$

The major challenge with this synchronous approach is that some resource-intensive environments can take significantly longer time to simulate and this introduces an overhead as every worker must wait for the slowest to finish collecting experience. The possible solution that DD-PPO implements is a preemption threshold where the rollout collection stage is forced to end early once some percentage of the other workers are finished collecting their rollout.

Capitolo 3

Embodied AI

Now that I have presented the primary techniques and algorithms applied in this thesis, this section focuses on introducing the training process and methods.

The machine learning community conducted numerous studies using the "Internet AI" approach to train models. The Internet AI approach involves training a model with static data obtained from the Internet, including images, videos, and text, without any interaction with an environment.

The demand and interest in general-purpose AI systems has increased significantly in the last few years. Due to this trend, recent advances in deep learning, reinforcement learning, computer graphics and robotics have prompted researchers to move away from the Internet AI paradigm, towards a more all-encompassing form of intelligence: Embodied AI. This approach enables artificial agents to acquire knowledge through their interactions with their surroundings.

3.1 History

In 2005 Linda Smith [34] proposed the embodiment hypothesis which suggest that intelligence emerges through the interaction of an agent with its environment and sensorimotor activity. The hypothesis emphasizes the importance of a baby's early grounding in physical, social, and linguistic world for the development of flexible and inventive intelligence in humans. It also states that many cognitive features are deeply dependent on the physical body of an agent, and the agent's body plays a significant role in their cognitive processing. While the initial hypothesis originated from Psychology and Cognitive Science, recent research developments in Embodied AI have largely come from Computer Vision researchers. Embodied AI integrates various interdisciplinary fields, including Natural Language Processing, Computer Vision, Reinforcement Learning, Navigation, Physics-Based Simulations, and Robotics. This convergence of fields allows for the training of artificial agents in realistic 3D environments, where they make decisions based on egocentric perceptual inputs that change with their actions. Embodied AI enables the training of virtual robots and egocentric assistants in simulated environments before transferring their learned skills to reality, representing a shift from traditional "Internet AI" reliant on static datasets to a more dynamic "Embodied AI" approach.

3.2 Motivation

General motivations and challenges In recent years, significant progress has been made in AI, especially in machine learning and deep learning. These advancements are driven by the abundance of data and increased computing power from CPUs, GPUs, and TPUs. However, a challenge arises from the fact that the data used, often termed "Internet AI," lacks a representative first-person perspective of human perception. Predominant data sources, such as YouTube, Facebook, and satellites, produce shuffled, randomized information collected from various origins.

While Internet AI have shown success in computer vision and natural language processing, this approach may not be optimal for building AI that navigates and interacts in the physical world. In nature, creatures, including humans, learn through sequential experiences involving seeing, moving, interacting, and speaking, rather than from shuffled and randomized data.

Moreover, in Internet AI, there's a lack of immediate feedback on specific actions or choices, unlike in nature where individuals often know the immediate rewards or punishments and potential long-term advantages or disadvantages. Embodied AI addresses this by implementing a reward system through reinforcement learning, allowing virtual robots or embodied agents to learn in a way similar to humans—by actively engaging with the world.

Despite its distinct methodology, Embodied AI can benefit from the successes in Computer Vision and Natural Language Processing, especially when ample labeled data is available. Indeed, the tasks that an embodied AI agent must solve, require the solution of several Internet AI tasks. For example, in object-goal navigation problem, the agent must be able to recognize the object's category. This problem is a famous Internet AI problem called semantic segmentation.

The primary disparity between Embodied agents and Internet AI agents is the former's active perception. The said agent may be generated in any location in the environment, and possibly, the pixels containing the answer to its visual target (e.g. a certain object) may not be immediately visible. So, to achieve success, the agent must effectively control the pixels it perceives. The agent must learn how to map its visual input to the proper action based on its perception of the world, the physical limitations present, and its comprehension of the question at hand. Once the agent performs its action, it receives a new observation till it reaches its goal. This process makes the data distribution controlled by the agent, unlike static datasets where the the data is predetermined.
Another difference between embodied AI and internet AI, is given from the fact that an embodied AI agent can approach a specific point from any direction. For this reason, one crucial hurdle to active perception is the ability to withstand visual variations.

An improvement in the performances of embodied AI agents is given thanks to the availability of realistic 3D scenes and simulated environments like SUNCG, Matterport3D, iGibson, Replica, Habitat, and DART. These environments offer greater realism compared to previous research simulators, enhancing the potential for success in Embodied AI. The different kinds of simulators and their characteristics are described more in detail in 3.1.

Applications One application of embodied AI is the training of physical navigation robots, a field of increasing interest evidenced, for example, by Tesla's proposed humanoid robot "Optimus" and Amazon's "Astro".

Embodied AI, provides a framework for virtual world training of such navigational agents, who can be subsequently deployed to the real world. Embodied AI can encompass various scenarios, ranging from a robot navigating in an environment without causing harm, to one that manipulates packages in a warehouse.

Achieving such capabilities requires the agent to overcome challenges that cannot be addressed with the conventional "Internet AI" approach. Some of these problems include safety (in many cases the robot will interact directly or indirectly with humans), lack of information (in the real world the agent doesn't know the properties of an environment. Just think about the fact that it is highly improbable that an agent operates inside an environment in which it has been trained, so it has to perform some kind of exploration), and cluttered environments (an agent has to be flexible for adapting to different situations that require different actions).

3.3 Simulators

As discussed previously one of the main embodied AI's goals is to train virtual agents that can then be deployed in the physical world.
After training an agent in a simulator further steps are required to deploy an actual robot in the real world, this process is called Sim2Real. Often, Sim2Real transfer involves a large amount of manual tuning of black-box parameters to generalize to new tasks. Another important factor is that in some domains, real-world data collection can generate more trustworthy and relevant data.

So why use a simulator? Why not directly train our robot in a physical environment and ignore this middle step? There are many reasons, for example:

- Collecting real-world data is a slow process. The real world doesn't run faster than real-time and cannot be parallelized.
- The process can be dangerous. Poorly trained agents can injure themself and humans or damage the environment
- Training in the real world is more expensive than training in a simulator.
- Some conditions in the real world are difficult to control or reproduce

For this reason, training a virtual agent in a simulator and then tuning it to suit the needs of the real world is almost always the best choice. [11] classifies the simulator according to different parameters. **Environmnet:** The simulator's environment can be built in two different ways: using game-based scene reconstruction or world-based scene reconstruction. The firsts are made using 3D assets while the latter are made using 3D scans of real-world environments. The game-based scene provides built-in physics and object classes well-segmented compared to the ones offered by the world-based scene. The world-based scenes on the other hand provide higher fidelity and, in general, more realism. **Physics:** The physics parameter has a large weight on the choice of the simulator that we want to select in order to complete our task. Some tasks don't require accurate physics (e.g. navigation) and others require advanced physics (object

manipulation). In general, the physics features of a simulator can be divided into basic features (e.g. collisions, rigid body dynamics, gravity), and advanced features (e.g. cloth, fluid and soft-body physics). **Object type:** There are two main sources for objects that are used to create the simulators. The first type is the dataset driven environment, where the objects are mainly from existing object datasets. The second type is the asset driven environment, where the objects are from the net such as the Unity 3D game asset store. **Object property:** Object property determines the level of interactivity that the agent has with the objects inside the environment. **Controller:** Controller parameter determines how the agent is controlled in the environment. We can identify three types of controllers: Python API controller, virtual robot controller, and virtual reality controller. **Action:** The actions that an agent can perform within a simulator range from basic navigation actions to higher-level human-computer actions via virtual reality. Three tiers of actions are identified: navigation actions (basic action for moving in the environment), atomic actions (basic object manipulation) and human-computer interaction (action given by a virtual reality controller). **Multi agent:** A simulator with this feature enables the presence of more than one artificial agent in the scene. The agents can interact with each other, cooperate or compete.

The table 3.1 provides a summary of all the different characteristics of each simulator.

Simulator	Environment	Physics	Object type	Object property	Controller	Action	Multi-Agent
DeepMindLab	G	×	×	×	P,R	N	×
AI2-THOR	G	B	O	I,M	P,R	AA,N	U
CHALET	G	B	O	I,M	P,R	AA,N	U
VirtualHome	G	×	O	I,M	R	AA,N,H	×
VRKitchen	G	B	O	I,M	P,V	AA,N,H	×
Habitat-Sim	W	B	D,O	I	R	AA,N	U
SAPIEN	G	B	D	I,M	P,R	AA,N	×
iGibson	W	B	D	I	P,R	AA,N	U
ThreeDWorld	G	B,A	O	I	P,R,V	AA,N,H	AT

Tabella 3.1: Comparison among different embodied AI simulators. Legend: G: game-based scene, W: world-based scene, B: basic physics, A: advanced physics, D: data-driven, O: asset driven, I: interact-able objects, M: multi-state objects, P: direct python API, R: virtual robot, V: virtual reality, N: navigation, AA: atomic action, H:human-computer interaction, AT: avatar based, U:sensor-based. Classification proposed by [11]

3.4 The tasks

In this paragraph, I will list and describe some tasks that are studied in the field of embodied AI.

- Visual odometry: Odometry is using any sensor to determine how much distance has been traversed, in visual odometry this sensor is of visual type (e.g. camera). In the odometry task the distance traversed is relative to the starting position, which is known. Visual odometry is essential for many tasks in embodied AI because is one of the most essential techniques for pose estimation.

- Global localization: Localization is the problem of estimating the position of an autonomous agent given a map of the environment and agent observations. Localization is considered as one of the most fundamental problems in robotics and it is useful in many real-world applications such as autonomous vehicles,

factory robots, and delivery drones. The global localization problem doesn't assume the initial position (unlike Visual Odometry). Despite the long history of research, global localization is still an open problem.

These are two basic problems that we need to tackle to solve downstream tasks like visual navigation, planning, exploration, or other more complex tasks.
After the basic problem here I'll present more complex tasks like visual exploration and visual navigation. Each of these tasks makes up the foundation for the next task(s), forming a pyramid structure of embodied AI research tasks 3.1. In embodied AI there are also more complex tasks that I won't cover like embodied QA, where visual navigation and question-answering tasks are combined 3.2.

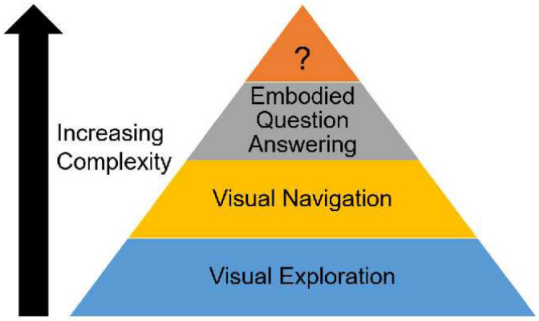

Figura 3.1: Embodied AI task in complexity order. The simpler tasks are the bases for more complex tasks. [11]

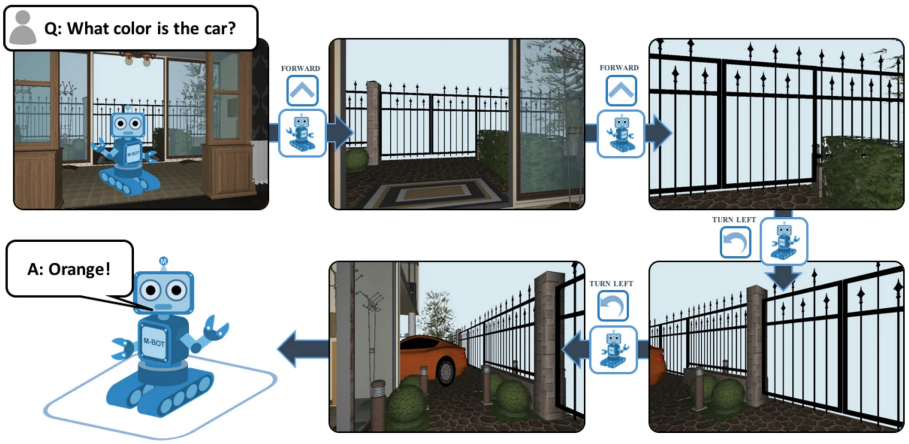

Figura 3.2: Embodied question answering is one of the most challenging tasks in embodied AI. In this task, the agent has to answer correctly to a question about the environment. Source

3.4.1 Visual exploration

In visual exploration, an agent gathers information about a 3D space, typically through motion and perception, to update its internal model of the environment, which might be useful for downstream tasks like visual navigation. The internal model can be in the form of a topological graph map (a map of the environment built as a graph)[1], a semantic map[27], an occupancy map [31], or spatial memory [19] (the ability to plan given an incomplete set of observations about the world). These map-based architectures can capture geometry and semantics, allowing for more efficient policy learning and planning. Visual exploration can be done before a downstream task like visual navigation, or at the same time. In the first case, the agent is free to explore the environment within a certain budget before the start of navigation. In the latter case, the agent builds the map by navigating in an unseen test environment.

SLAM

In classical robotics, exploration is done through passive or active simultaneous localization and mapping (SLAM). Basically, SLAM is a problem of constructing and updating the map of an unknown environment while simultaneously allowing the robot to localize itself in the built map. Active SLAM is formulated as the problem of controlling a robot's motion to minimize the uncertainty of its map representation and localization. Active SLAM can also be seen as adding the task of optimal trajectory planning to the SLAM task. This integration allows a mobile robot to perform tasks such as autonomous environment exploration.

In SLAM, both the robot and the landmark's locations are estimated simultaneously when it explores the unknown environment. The ground truth information of where the robot is and the landmarks are always unknown to the robot. The observations done are noisy in most cases, and they try to give an estimate to the robot about its current location and the landmark's location.

Active SLAM

Active Simultaneous Localization and Mapping (Active SLAM) is the problem of planning and controlling the motion of a robot to build the most accurate and complete model of the surrounding environment, in other words, Active SLAM refers to the joint resolution of:

- Create a map of the environment.
- Localize itself on the environment.
- Control its own motion.

Active SLAM can be seen as a decision-making process in which the robot has to choose its own future control actions, balancing between exploring new areas and exploiting those already seen to improve the accuracy of the resulting map model. Currently, active SLAM is at a decisive point, driven by novel opportunities in spatial perception and artificial intelligence (AI). These include, for instance, the application of breakthroughs in neural networks to prediction beyond line-of-sight, reasoning over novel environment representations, or leveraging new SLAM techniques to process dynamic and deformable scenes.

Curiosity

If an agent uses the curiosity approach to explore the environment then it seeks the states that are difficult to predict [4]. Here the prediction error is used as a reward signal for reinforcement learning which is beneficial in cases where external rewards are sparse.
So in curiosity learning we can divide the reward into two parts:

- Primary reward: This is the reward that comes from the environment. This reward can be awarded by completing the scene or reaching a checkpoint. It is not given at every action and so it is sparse.
- Curiosity reward: This reward is the reward the agent got when it explores new areas, the more these areas are difficult to predict the higher is the reward. It is no more a sparse reward since it is awarded at each action.

A problem with this approach can arise when the forward-dynamics model exploits stochasticity for high prediction error (i.e. high reward), due to factors like the "noisy-TV" problem (i.e. agent's use of a remote controller to randomly change TV channels, allowing it to accumulate rewards without progress.).

Coverage

In the coverage approach, an agent tries to maximize the number of targets it directly observes. Since the agent uses egocentric observations, it has to navigate based on possibly obstructive 3D structures. Usually, this kind of exploration is implemented by giving the agent a specific reward signal to encourage the exploration:

$$r_{explore} = const \, \frac{d^t}{v}$$

where *const* and d are constant, v is the number of steps that the agent spent in a particular area of the map and t is the episode timestep.

3.4.2 Visual navigation

In visual navigation, an agent navigates a 3D environment to a goal with or without external priors or natural language instruction. Many types of goals have been used for this task, such as points, objects, images, and areas. In this Chapter I will focus on the description of Object-goal navigation since, combined with a social component, is the main task that I have used for my experiments. Under point goal navigation the agent is asked to navigate to a specific point determined by x, y, and z coordinates while in object-goal navigation the agent is tasked to navigate to an object of a specific class. Visual navigation in embodied AI aims to learn to navigate in a given environment from data given to the agent, so as to reduce case-specific hand engineering and having an easier integration with downstream tasks.

3.4.3 Point-goal navigation

In point-goal navigation, the agent is asked to navigate to a certain distance from a point defined by cartesian coordinates. Generally, the agent spawned at the origin (coordinates (0,0,0)) or in a random position, and the goal is specified as 3D coordinates (x, y, z) relative to the origin/initial location. In this task, the agent is successful if

it stops from a certain distance (defined a priori) from the coordinates (x, y, z). To complete the task the agent has to have different skills, like: visual perception, episodic memory construction, reasoning and navigation. The agent is usually equipped with GPS+COMPASS sensor that gives it its coordinates inside the environment and its orientation relative to the goal position.

State of the art

Point-goal navigation is a fundamental task, and many studies tried to tackle it in the last years. One of the early studies [26], uses an end-to-end learning approach to tackle point-goal navigation in an unseen realistic environment (no ground map or ground-truth agent position). The algorithm used in this work is Direct Future Prediction (DFP), where inputs like RGB image, depth map and previously taken actions are processed by different neural networks and concatenated to be passed into a two-stream fully connected action-expectation network. The outputs are the future measurement predictions for all actions and future time steps. In 2019 Habitat introduced the point-goal navigation challenge and standard evaluation metrics, evaluation setup and dataset. In this challenge [39] proposed a near-perfect solution for the point-goal navigation task, where their best agent's performance is within 3-5% of the shortest path oracle. To achieve these results they trained an agent using DD-PPO algorithm for 2,5 billion steps. At each time step the agent receives an egocentric observation (depth and RGB), gets embeddings with a CNN, utilizes its GPS and compass to update the target position to be relative to its current position, and then finally outputs the next action and an estimate of the value function. In the validation dataset, they achieved near-perfect results: In Gibson 4+ dataset they got an SPL of 95,6% and a Success rate of 99,6%.

3.4.4 Object-goal navigation

Object-goal navigation is one of the most challenging tasks in embodied AI. In this task, the agent is initialized at a random position and will be asked to find an instance of an object category within an unexplored environment. Object-goal navigation is generally more complex than point-goal navigation since it not only requires many of the same skills such as visual perception and episodic memory construction but also semantic understanding. We have decided to study this task because it offers good complexity since it combines two main elements:

- A navigation component, where the agent has to learn how to efficiently navigate inside an environment
- A visual component, where the agent has to recognize the different objective classes within the environment. This component also includes the capability of the agent to understand which region of the environment has the higher probability of having the objective goal (e.g. it is improbable to find an oven in a bathroom).

State of the art

In this section, I will describe two works that study the object-goal navigation problem from two different perspectives. The first work is proposed by [8], based on the idea that end-to-end learning-based navigation methods struggle with object-goal navigation as they are ineffective at exploration and long-term planning. In their work they propose

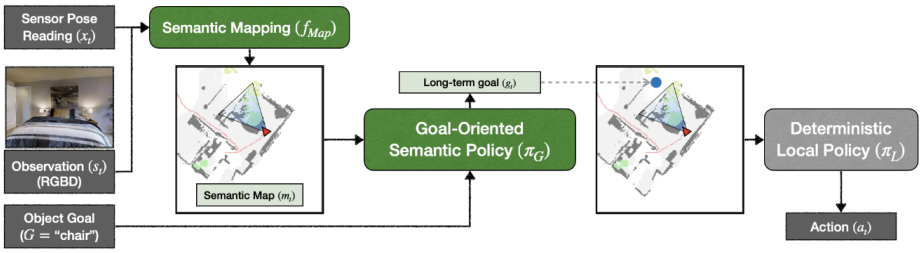

Figura 3.3: Slam based method proposed by [8]

a modular model called 'Goal-Oriented semantic exploration' and it consists of two modules:

- Semantic mapping.
- Goal-oriented semantic policy.

The semantic mapping module builds a semantic map over time and the goal-oriented semantic policy selects a long-term goal based on the semantic map. The semantic map is a $K \times M \times M$ matrix where $M \times M$ denotes the map size. K=C+2 denotes the number of channels in the semantic map, where C is the number of semantic categories. The 2 additional channels represent obstacles and explored areas. In order to build the map the authors predicted semantic segmentation from the first-person view and used differentiable projection to build the actual top-down map. The goal-oriented semantic policy works in this way: if the object goal is observed then it selects the position of the map where the goal was observed as a long-term goal. If the goal has not been observed yet then the module selects as long-term goal a location where the objective is most likely to be found. To do that it needs to learn semantic priors on the arrangement of objects and areas. We can define this proposal as a SLAM method. 3.3

The second work is proposed by [40].Respect to the previous study they propose an end-to-end agent but, instead of using vanilla visual and recurrent modules (CNN + RNN), they added auxiliary learning tasks and an exploration reward. To accelerate the training they added two semantic features: semantic segmentation feature and semantic goal exist feature (SGE). SGE is a scalar that equals the fraction of visual inputs that are occupied by the goal category, this feature is useful because it distills the knowledge that the agent should navigate to the goal once it is seen. The particularity of their solution is represented by a set of belief modules. Each belief module is an independent GRU each associated with a separate auxiliary task 3.4. In the paper, they used 6 different auxiliary tasks:

- 2 instances of CPCA|A [17] and PBL [13]. These tasks use agent states to make prediction about the environment
- Action distribution prediction (ADP) and Generalized Inverse Dynamics (GID). ADP and GID both predict actions taken between two observation k frames apart and are both conditioned on the belief at the first frame and on the visual embedding k frames apart. ADP uses an MLP to predict action distribution and then it evaluates the KL-divergence between the prediction and the empirical

distribution of the next k actions. GID instead uses a GRU to predict the next k actions.

- The last task is Coverage Prediction (CP). Coverageprediction leverages a GPS sensor to predict the change in the coverage at each of the next k steps.

The author uses also a reward to encourage the agent to explore as much as possible and an off-policy training to both teach the agent how to explore but also to encourage efficient object goal navigation. In particular, the total reward is defined as:

$$r_{total} = r_{success} + r_{slack} + r_{explore} \tag{3.1}$$

Where $r_{success} = 2.5$ measures the reward once the agent reaches the goal category object, $r_{slack} = -0.0001$ is a slack reward in order to encourage faster goal-seeking, and $r_{explore} = 0.25\text{x}\frac{d^t}{v}$ is an exploration reward to encourage the agent to explore the environment. $r_{explore}$ is decayed by the number of steps that the agent spent in a particular region (visit count v) and by a constant $d = 0.995$ to ensures the agent prioritizes object-goal navigation.

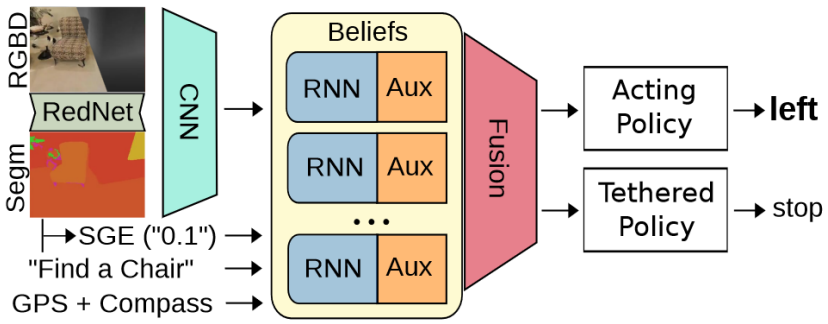

Figura 3.4: Architecture proposed by [40]

3.4.5 Social navigation

As I've described in the previous section, in the embodied AI community many problems are studied such as:

- Visual odometry
- Global localization
- Visual navigation
- Embodied question answering

All these tasks are often studied and performed by not taking into consideration the presence of other humans or entities in the room, and this becomes a problem especially if we consider that one day, the agent that we are training will be transferred to the real world, where the presence of humans can't be ignored.

Many studies focused on the scalability of the number of pedestrians in a particular open space. The indoor environments present additional challenges, most related to constrained spaces such as corridors and doorways that limit maneuverability and influence patterns of pedestrian interaction.

In this work, I've studied solutions for social object-goal navigation tasks. In this problem, the agent must find a particular object instance in the environment and avoid collisions with objects or humans.

State of the art

Navigating safely among humans is a prerequisite for the deployment of mobile robots. This has motivated researchers in autonomous navigation and human-robot interaction to study different aspects of navigation problem, such as navigation within crowds [9], using anticipatory signals [38], and the effects of distinct robot navigation strategies on pedestrians [25]. Despite these studies, reliable and safe mobile robots remain a challenge for the research community. Many were based on two components:

- A motion planner that computes the path to follow.
- A controller that executes the plan.

The problem with this approach is the "freezing robot problem" in which the planner finds all feasible or safe solutions but, at the moment of execution, the agent continuously stops and a new path must be recomputed due to the pedestrians' movement. Researchers have overcome this limitation by considering machine learning approaches. Some approaches are based on imitation learning [30][3]. Imitation learning is a framework for learning a behavior policy from demonstrations, usually, these demonstrations are presented as a form of state-action trajectories, with each pair indicating the action to take at the state being visited. The problem with imitation learning is that it doesn't generalize well outside the states that the agent has seen during the training. More recent approaches proposed solutions based on deep reinforcement learning and developed multi-agent collision avoidance focusing on optimizing path efficiency. In 2020 [28]proposed a method that takes advantage of the complementary strengths of motion planning and reinforcement learning (RL), in which the RL component learns

to handle the local interactions with pedestrians as it pursues the globally planned trajectory (through the usage of waypoints) and adapts to the current conditions of the environment observed with on-board sensors. The set of waypoints followed by the agent are given by a motion planner that takes in input the map of the environment. In this last paper, the reward function is defined as:

$$r_{total} = r_{goal} + r_{timestep} + r_{collision} + r_{potential} + r_{waypoints} \tag{3.2}$$

Where r_{goal} is a sparse reward assigned once the agent reaches the desired goal, $r_{timestep}$ is a dense negative reward designed to encourage the agent to reach the goal faster, $r_{collision}$ is a negative reward assigned to the agent in case of collision with an object, and $r_{waypoints}$ is a dense reward assigned at each timestep that measures how close is the agent to the next waypoint. The policy is learned using a soft-actor critic algorithm, that provides velocity references to a low-level velocity controller. Compared to the majority of papers in this field the architecture provided by the authors doesn't include any CNN since the inputs given to the agent are not images but are provided by a LiDAR sensor and from the waypoints that it has to follow.

Another contribution proposed by [28] is a compositional multi-layout training regime using canonical walls-worlds layout and the demonstration that a policy that is trained in this simple world is able to generalize well in 3-D scans of real and complex environments ??.

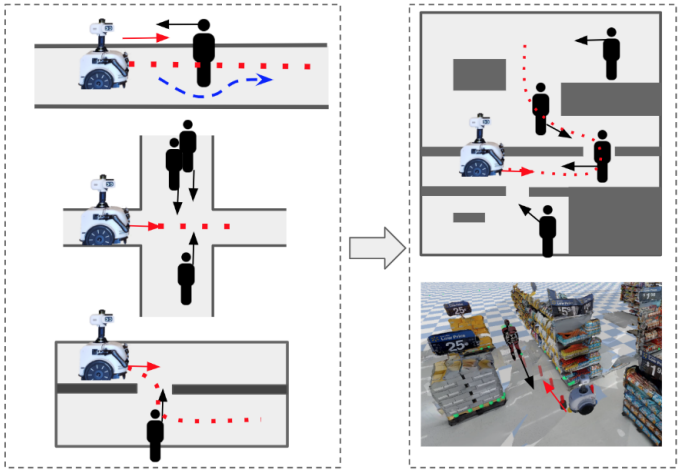

Figura 3.5: Architecture proposed by [28]. On the left we have the simple walls-world layout where the agent is trained (the waypoints are represented by the red dots). On the right we can see the generalization on more complex walls-worlds or on a real and complex 3D environment.

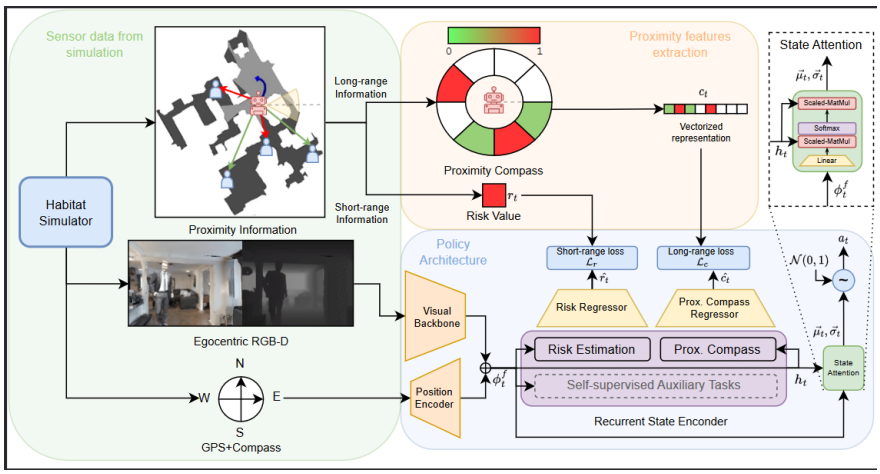

Figura 3.6: Architecture proposed by [6]

Another study in this field was conducted by [6] 3.6. In this research, the authors proposed an end-to-end architecture that exploits special auxiliary tasks called proximity tasks. These proximity tasks can inject a common-sense social behavior into the agent's policy. In particular, they encoded two proximity tasks:

- Risk estimation. This task is designed to deal with short-range social interactions.
- Proximity compass. Unlike risk estimation, this task gives to the agent more general social information, like the direction of human beings inside the environment.

These proximity-aware tasks represent a crucial point in this thesis since I've conducted several experiments in order to measure how much, and in which order, these proximity

tasks affect the agent's performances in the context of social object-goal navigation. For this reason, a more in-depth analysis of risk estimation and proximity compass can be found in 5.2.3

Evaluation Metrics

In this section, I will list some of the evaluation metrics that I have used to evaluate the agents in the context of social object-goal navigation. These evaluation metrics are divided into two sub-groups: generic object goal navigation evaluation metrics, and evaluation metrics specific to a social context. The first sub-group is about evaluation metrics that are important in object-goal navigation, regardless of whether a social component is present or not. The evaluation metrics that belong to the second group are the ones that are explicitly used in a social environment (so, with the presence of simulated human beings).

Generic object goal evaluation metric The two main evaluation metrics that belong to the group of generic object-goal navigation are the following:

- Success rate
- Success weighted by (normalized inverse) path length (SPL)

The success rate is simply the percentage of episodes the agent completed successfully. The term "successfully" depends on which way the task is defined, in our specific case an episode is called "successful" when the agent stops with the object goal inside its field of view within 0.1 meters from it.

The success rate tells us how many times the agent found the object goal, but it tells nothing about its efficiency in finding it. We want a metric that also measures the quality of the path followed by the agent in finding the object goal. Success weighted by path length (SPL), is a metric that combines the success rate and the length of the path traveled by the agent:

$$SPL = \frac{1}{N} \sum_{i=1}^{N} \frac{l_i}{max(p_i, l_i)} S_i \tag{3.3}$$

Where N is the number of episodes, l_i is the shortest path length, p_i is the agent's path length and S_i is a boolean success indicator for the episode i. This is a very strict metric because in order to be equal to one, the agent had to be successful in all the episodes, and at the same time it had to follow the shortest path from its random start position to the object goal. This strictness can be a problem, especially in the first steps of the training where the agent tends to make a lot of mistakes and never reach the objective goal. In this case, the SPL is fixed to zero and we can't understand if the agent is actually learning or not. In order to overcome this issue, other metrics were introduced such as softSPL which is similar to SPL with a relaxed soft-success criteria. In softSPL, instead of a boolean, the success is now calculated as 1 - (ratio of distance covered to target). Another useful metric is the reward itself (should increase over episodes), or the "distance to goal" (should decrease over episodes), which measures the average distance from the goal at the end of the episode.

Social Object goal navigation task Now I will describe those metrics that are used in a social context, which means in the presence of (simulated) humans. Keep in mind that this scenario is a sub-task of object goal navigation, so in addition to the metrics that I will list in this paragraph, I've used also the metrics that I've described in the previous paragraph. The two main metrics in this scenario are:

- Human collision
- Personal space compliance

Human collision is defined as the percentage of episodes terminated due to a collision with a human. The collision with a human occurs when the agent is closer than a certain threshold τ to a human. In this thesis $\tau = 0.1$.

It's important that the agent doesn't collide with humans, but it's also important that the agent doesn't take up their personal space. To measure this value another metric was introduced: personal space compliance. Personal space compliance is a metric that is equal to the percentage of timesteps that the agent maintains a certain distance, defined by a threshold, with the people. In our case, the threshold was fixed at 0.5 meters.

3.4.6 Datasets

Multiple datasets are used in embodied AI, each one with different characteristics and suitable for certain tasks. For example, ReplicaCAD [37] is a dataset designed in particular for object rearrangement by including more than 90 objects with convex collision geometry and physical properties. In navigation tasks, such as the one considered in this thesis, one of the most popular datasets is Matterport 3D a large-scale RGB-D dataset for embodied AI.

Matterport 3D The Matterport dataset [7] is one of the most used datasets in embodied AI, especially in navigation and, in general, in those tasks that don't require a lot of interactivity with the objects' environment (e.g. rearrange task). The dataset comprises a set of 194,400 RGB-D images captured using a Matterport camera. At the time the paper was published it was the most complete dataset by including depth and color, 360° panoramas for each viewpoint, human-height viewpoints, instance-level semantic segmentation into regions and object categories, and by providing data collection from living spaces in private homes.

Capitolo 4

Simulator

As already mentioned, in this work I've used an active approach to train the agent, and this approach takes the name of "Embodied AI". Embodied AI is the science and engineering of intelligent machines with a physical or virtual embodiment. This represents a paradigm shift from "internet AI", based on static datasets (e.g. ImageNet or COCO) to a method that acts within realistic environments, uses an active perception, long-term planning, learning from interaction, and holds a dialog grounded in an environment.
Habitat [32] enables the training of such embodied AI agents in a highly photorealistic and efficient 3D simulator, before transferring the learned skills to reality.

In this chapter, I'm going to explain and describe the platform used in my work: Habitat. Habitat is a platform for embodied AI research and it includes:

- Habitat-Sim: a flexible high-performance 3D simulator with configurable agents, multiple sensors, and 3D dataset handling.
- Habitat-Lab: a modular library for end-to-end development in embodied AI.

4.1 Habitat

We can see Habitat-lab as the framework for end-to-end development in embodied AI. Here we can define an embodied AI task (e.g. navigation, interaction, instruction following, question answering), train agents (e.g.using reinforcement learning), and benchmark their performance on the defined tasks using standard metrics. In order to perform all these operations Habitat-Lab needs a simulator: Habitat-Sim. Habitat-Sim is a high-performance physics-enabled 3D simulator that allows an agent to navigate in particular scenes and interacts with the environment.
So in order to set up and perform the task we need to act from the point of view of the simulator Habitat-Sim (how can the simulator be set up for these tasks?) and from the point of view of the framework itself (how can we perform the task using Habitat-Lab?).
The philosophy of Habitat is to prioritize simulation speed over the breadth of capabilities. If we use a scene from the Matterport3D dataset, Habitat-Sim achieves thousands of frames per second running in a single thread and over 10,000 fps in multi-thread on a single GPU.

In the following section, I will describe in detail how habitat-sim works and the main habitat-lab's functionalities. In particular, for habitat-sim, I will explain how we can configure the simulator and an agent that operates inside it and I will briefly describe the main mechanism that allows the agent to navigate insided an environment. For Habitat-lab I will describe its structure and the main modules that are needed in order to set up the task.

4.1.1 Habitat-sim

Habitat-Sim is a physics-enabled 3D simulator with support for:

- 3D scans for indoor/outdoor spaces.
- CAD models for spaces and piecewise-rigid objects.
- Configurable sensors.
- Possibility to describe a robot via URDF (an XML format for representing a robot model).
- Rigid-Body mechanics (via Bullet).

Configuration

Habitat-Sim is the actual simulator of the Habitat platform. It allows configuring agents with their sensors and handling 3D datasets. Thanks to its flexibility Habitat-Sim can support a wide range of tasks from the ones that involve navigation (point-goal navigation, object-goal navigation, etc...) to the ones that involve interaction with the environment like rearrangement tasks or social navigation itself. To run the simulation, we need to create a configuration that is understandable by our simulator. Such configuration consists of 2 parts:

- One for the simulator backend. It specifies the parameters that are required to start and run the simulator. For example, the scene to be loaded or to enable physics or not.
- One for the agent. It describes the parameters to initialize an agent, such as height, mass, or the config for the attached sensors. There is also the possibility to define the amount of displacement e.g., in a forward action and the turn angle.

We can define the configuration of the simulator backend using the SimulatorConfiguration class. The SimulatorConfiguration class has a lot of attributes that help us to configure the simulator to perfectly fit our needs, here are some examples:

- allow_sliding: A boolean value that indicates whether or not the agent can slide on NavMesh collision
- default_agent_id: The default agent id is used during initialization and functionally whenever alternative agent ids are not provided.
- enable_physics: Specifies whether or not dynamics is supported by the simulation if a suitable library (i.e. Bullet) has been installed.
- scene_dataset_config_file: The location of the scene dataset configuration file that describes the dataset to be used.

Moreover, we can add task-specific configuration options, for example for the social navigation task we can specify:

- NUM_PEOPLE: The number of people that are present in the scene
- PEOPLE_LIN_SPEED: The linear velocity with which each person moves.
- PEOPLE_ANG_SPEED: The angular velocity with which each person turns.

To define the configuration of the agent the AgentConfiguration class is used. Also here we can define a lot of parameters in order to create the agent that satisfies our needs. Here are some examples:

- sensor_specification: This parameter accepts a list of objects of type SensorSpec, the class that we have to use to define the sensors of the agent.
- height: The height of the agent in meters
- radius: The radius of the agent in meters
- action_space: A parameter that accepts a dictionary that is generally on the of the form [string, ActionSpec], where ActionSpec is the class that defines a particular action of the agent.

4.1.2 Navigation

Once the configuration for the simulator and for the agent is done, the agent is now capable of actually navigating inside the scene that we have loaded in the simulator. This process can be performed by calling the function "step" provided by the class "Simulator" of habitat-Sim. The function "step" takes as an argument an action that belongs to the agent's action space.

NavMesh

When the agent navigates in a particular environment we can't take the navigation constraints and collision response for granted. Habitat-Sim provides a mechanism, light and fast, to enforce such constraints: the NavMesh.
A navigation mesh (NavMesh) is a collection of two-dimensional convex polygons that defines which areas of an environment are traversable by an agent with a particular embodiment. An agent can freely navigate around these areas unobstructed by objects, walls, gaps, or other barriers that are part of the environment. Adjacent polygons are connected to each other in a graph enabling efficient pathfinding algorithms to chart routes between points in the NavMesh.

Just like the simulator and the agent also the NavMesh has configuration options to customize it and make it suitable intended use case.

An important configuration that affects the NavMesh is to allow sliding or not. Most game engines allow agents to slide along obstacles when commanding actions that collide with the environment. While this is a reasonable behavior in games, it doesn't reflect the real world and this results in faster training and better simulation performances but worse performances when the trained policy is transferred to the real world. This increase in simulation performances is because we can consider sliding as a cheating factor for the agent. The agent exploits this sliding mechanism to take an

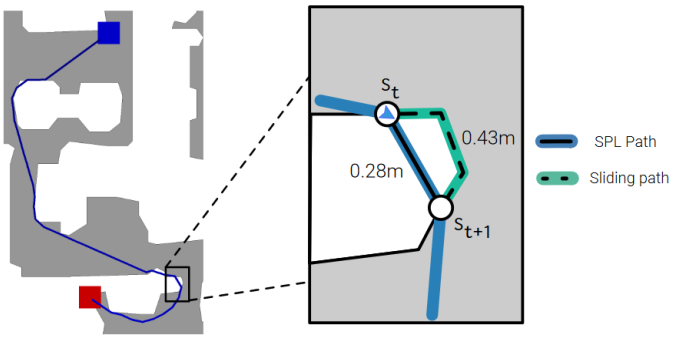

Figura 4.1: Sliding effect. At timestep, t the agent slides within the wall and, as a result, the SPL computed is not representative of the movement taken by the agent. [23]

effective path that appears to travel through non-navigable regions of the environment [23]. Let s_t denote the agent position at time t, where the agent is already in contact with an obstacle. The agent executes a forward action, collides, and slides along the obstacle to state s_{t+1}, as shown in figure 4.1. The path taken during this maneuver is far from a straight line, however, Habitat computes SPL (the metric that the agent optimizes) as the Euclidean distance traveled $||s_t - s_{t+1}||_2$. This is equivalent to taking a straight line path between s_t and s_{t+1} that goes outside the navigable regions of the environment. On the one hand, this behavior from the agent shows the success of large-scale reinforcement learning because these moves maximize SPL. On the other hand, this is a problem when we transfer the trained policy in the real world for the lack of proper physical properties in the simulator.

4.2 Habitat-lab

Habitat-lab is the framework that we use to set up different aspects of the work that we are working on like the dataset, the task itself, and which kind of simulation configurations we want to load. In the following section, I will describe the general structures of Habitat-lab and the main modules that are necessary to set up the job.

4.2.1 The structure

An important objective of Habitat lab is to make it easier for users to set up a variety of embodied agent tasks in 3D environments. The process of setting up a task involves using environmental information provided by the simulator (Habitat-sim), connecting the information with a dataset, and providing observations that can be used by the agent 4.2. Habitat provides the following key concepts as abstractions that can be extended:

- Env: The fundamental environment concept for Habitat. All the information needed for working on embodied tasks with a simulator is abstracted inside an Env. Env consists of three major components: a Simulator, a Dataset (containing Episodes), and a Task, and it serves to connect all these three components together.

- Dataset: contains a list of task-specific episodes from a particular data split and additional dataset-wide information. Handles loading and saving of a dataset to disk, getting a list of scenes, and getting a list of episodes for a particular scene.
- Episode: A class for episode specification that includes the initial position and the orientation of the agent, a scene id, a goal position, and optionally the shortest path to the goal. Basically, we can define an episode as a description of one task instance for the agent.
- Task: This class builds on top of the simulator and dataset. The criteria for episode termination and measures of success are provided by the Task.
- Sensor: a generalization of the physical Sensor concept provided by a Simulator, with the capability to provide Task-specific Observation data in a specified format.
- Observation: data representing an observation from a Sensor. This can correspond to physical sensors on an Agent (e.g. RGB, depth, semantic segmentation masks, collision sensors) or more abstract sensors such as the current agent state.

Note that the core functionality defines fundamental building blocks such as the API for interacting with the simulator backend, and receiving observations through Sensors. Concrete simulation backends, 3D datasets and embodied agent baselines are implemented on top of the core API.

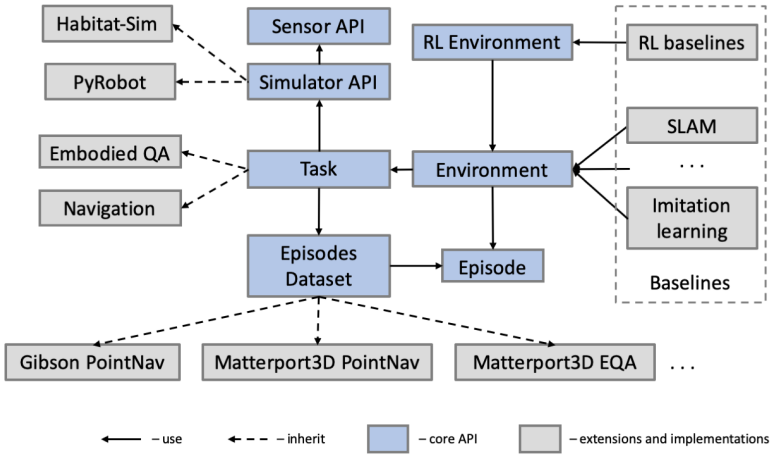

Figura 4.2: Habitat-lab structure.

Capitolo 5

Experimental settings and implementation

In this chapter, I will describe in detail the settings in which I have conducted my experiments. In particular, I will divide this section into three parts:

- In the first part I'm going to describe the agent and how it was configured
- In the second part I will explain the tasks.
- In the last part I will describe some implementation details and how I have used the RedNet to approximate semantic segmentation information.

The framework that I've used to perform all my experiments is habitat-lab V. 0.1.7. As a simulator I've used habitat-sim V. 0.2.1.

5.1 The agent

Let's start with the description of the agent architecture 5.1. The architecture is the same as the one proposed by [40], described in section 3.4.4, the differences here are that some auxiliary tasks like action distribution prediction and coverage prediction were not implemented. On the other hand, I've used additional auxiliary tasks specific for social object-goal navigation (proximity aware tasks). Another important difference is that in [40] they used a tethered policy to improve the exploration-exploitation tradeoff. Due to time limitations, we decided to not implement this feature.
The agent moves around in the environment by using the following set of discrete actions: [move forward, turn left, turn right, and stop]. Where "move forward" means that the agent moves forward 0.25 meters and can turn at an angle of 30 degrees.

5.1.1 Agent architecture

The high-level agent's process can be described in this way:

- Visual inputs (RGB, semantic information, Depth) are fed to a CNN to extract visual features. These visual features are then concatenated to a value called SGE, which represents the percentage of the object goal inside the agent's view, the goal itself, the GPS+Compass value and RISK+ SOCIAL COMPASS value.

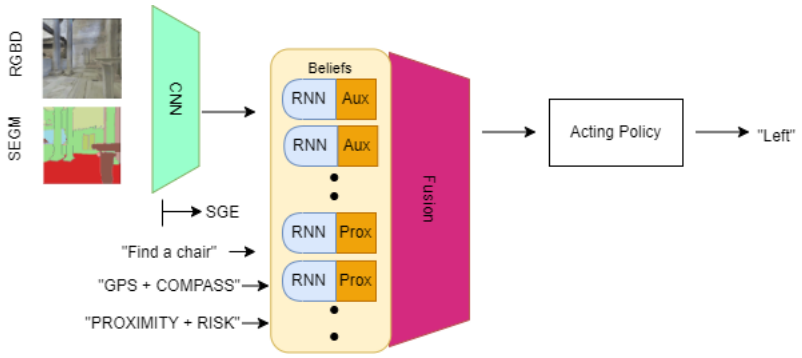

Figura 5.1: Agent's architecture. This agent is inspired by the one used by [40] 3.4.

.

- A series of GRUs networks integrate those observations embedding over time. Each GRU represents a particular auxiliary task that is learned simultaneously with the policy. These auxiliary tasks are important to help the agent to acquire a better understanding of the environment and of the consequences of his actions. Each GRU is called "belief module" and its output is called "belief". We can define a belief as sufficient statistics of the information seen so far.
- The beliefs are then fused with an attention layer and used in a linear actor-critic policy head. Each module contributes to the total loss, as shown in 5.1.5.

5.1.2 The inputs

At each step, the agent decides which action to take based on its input. The input can be divided into "visual input" and "information input". Visual inputs are those data that are collected from the visual sensors. In this case, the visual sensors are composed of an RGB sensor, a depth sensor, and a semantic sensor, each one with a resolution of 256 x 256, a FOV of 79°, and they are all positioned on the top of the agent.
The semantic sensor had a little customization to adapt it to our needs. In particular, by default, the semantic sensor is designed to make the segmentation of each single instance in the dataset. This means that the agent, with this type of sensor, sees two objects that belong to the same category as two completely different objects. The changes that have been made were done to address this problem. In practice, a dictionary was used to map each object to its object category and then, for each object that the agent saw, in the semantic sensor, the instance id of that object was replaced with its category id as depicted in 5.2 These visual inputs are then fed to a ResNet18 [20] to form an embedded visual. Besides these visual inputs, the agent gets other information, that helps it to reach the goal:

- The goal object ID, that is the ID of the object that the agent must find
- Semantic goal exists (SGE), that describes the fraction of the frame occupied by the goal object.
- GPS+Compass sensor to help the agent to navigate in the environment. GPS+Compass sensor provides the agent's current location and orientation information relative to the start of the episode.

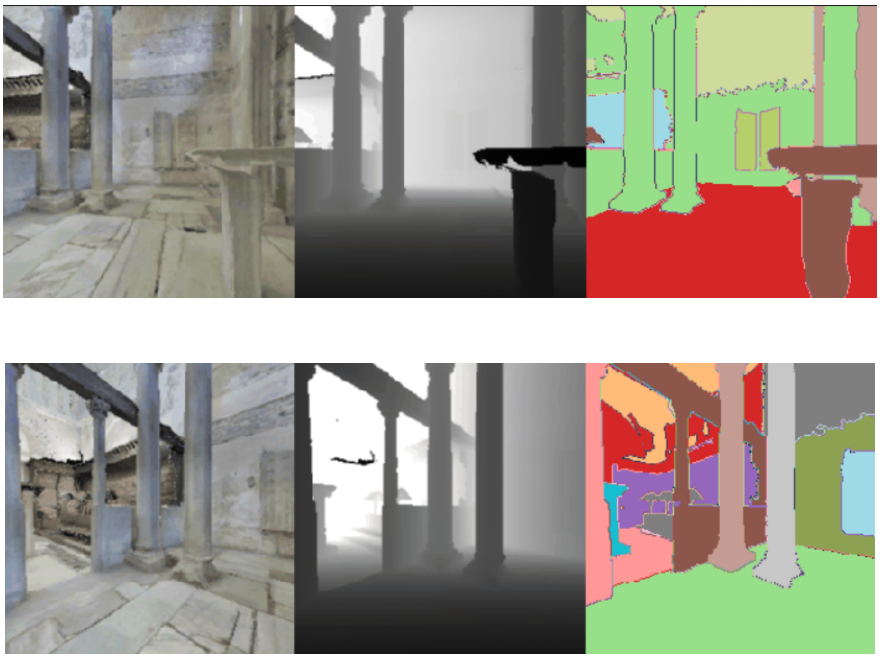

Figura 5.2: The agent's visual inputs during the training. The first image (top) depicts the set of visual inputs where we have a semantic category sensor, in the second image (bottom) we have a semantic instance sensor. Note how the color for objects of the same category (e.g. columns) is the same in the top image (objects grouped by categories) and different in the second image (objects are seen as different instances). Then, for both images, we have an RGB sensor (left) and a depth sensor (center) where the darker the object the closer it is.

Additional sensors are used when the agent uses also proximity aware tasks:

- SOCIAL COMPASS, which gives the relative position of the people inside the environment.
- RISK, which gives the"social risk" to which the agent is subjected (the social risk depends on how much the person are close to the agent).

The goal object ID is embedded into a 32-dimensional vector and concatenated with the SGE, the GPS+Compass sensor, and the embedded visual to form an observation embedding.

5.1.3 Beliefs modules and acting policy

The observation embedding is then fed to a series of belief modules, each associated with an auxiliary task. For this thesis, I've used the following auxiliary task:

- 2 instance of CPC|A
- Generalized Inverse Dynamics(GID)
- PBL

Better described in 5.2. An independent GRU represents each belief module, each having 2 recurrent layers with a hidden size of 196. The output cell states from all GRUs are called beliefs and they are fused using an attention layer conditioned on the observation embedding. The fused belief is then used in a linear actor-critic policy head. In addition to these auxiliary tasks, when the agent is trained in a social context (with people who also navigate inside the environment) other auxiliary tasks are added [6], in particular:

- Risk estimation
- Proximity Compass

This kind of auxiliary task (called proximity aware task) helps the agent to develop some social abilities useful to navigate and coexist in a social environment.

5.1.4 Reward signal

The agent is trained using PPO, a reinforcement learning algorithm. As explained in the previous sections, in reinforcement learning defining a good reward measure is crucial to be successful in the task that we want to complete. In habitat lab, a basic reward measure designed for point goal navigation is already implemented and takes into consideration the success, how much the agent is getting closer to the target, and a time penalty:

$$r_t = r_{success} + (d_{t-1} - d_t) + r_{slack} \tag{5.1}$$

where $r_{success}$ is a measure that defines if the agent completed the task successfully. If this is the case then $r_{success} = 2.5$ otherwise $r_{success} = 0$. $d_{t-1} - d_t$ returns the geodesic distance variation to the goal and $r_{slack} = -0.001$ is a time penalty to encourage faster goal-seeking.

In object-goal navigation, in contrast to point-goal navigation, an exploration of

the environment is really important, especially in the first training steps. So we add another contribution to the total reward r_{total}, $r_{explore}$. To measure this reward the map of the environment was divided into a voxel grid with $2.5m \times 2.5m \times 2.5m$ voxels and the agent was rewarded for visiting each voxel. With this reward, there is the risk that the agent prefers to explore the environment instead of finding its goal. To overcome this issue $r_{explore}$ has been decayed by the number of steps that the agent spent in the voxel, and by a constant based on episode time step t to prioritize the object goal. The final reward is defined as:

$$r_t = r_{success} + (d_{t-1} - d_t) + r_{slack} + r_{explore} \tag{5.2}$$

Where $r_{explore} = 0.25\frac{d^t}{v}$, v is the visitation frequency of each voxel and $d = 0.995$ is the decaying factor.

5.1.5 The training process

The agent was trained using reinforcement learning in particular, I've used the PPO algorithm. The total loss used is the following:

$$L_{total} = L_{rl}(\theta_m) - \alpha H_{action}(\theta_m) + L_{aux}(\theta_m; \theta_a) \tag{5.3}$$

$$L_{aux}(\theta_m; \theta_a) = \sum_{i=1}^{n_aux} L^i_{aux}(\theta_m, \theta^i_a) - \mu H_{attn}(\theta_m) \tag{5.4}$$

Where H_{action} is the entropy of the actions distribution, L_{aux} is the auxiliary tasks' loss, H_{attn} is the entropy of the attention distribution over the auxiliary task and L_{rl} is the reinforcement learning loss that is defined in this way:

$$L_{rl}(\theta_m) = \hat{\mathbb{E}}[L^{clip}_t(\theta_m) - c_1(V_\theta(S_t) - V^{target})^2 + c_2 S[\pi_\theta](S_t)] \tag{5.5}$$

The set of PPO's hyperparameters that I have used for the training are the ones pictured in 5.1

Hyperparameter	Value
Clip ϵ	0.2
aux coeff	2.0
lr	2.5e-4
epoch	4
Discount factor γ	0.99
Mini batch	1
Steps	5

Tabella 5.1: PPO's hyperparameters

5.2 Auxiliary Tasks

The auxiliary tasks, in particular the ones specific for social navigation (proximity-aware task) play an important role in this thesis. For this reason, I am going to describe in detail how these proximity tasks work and what are the advantages of using them. As anticipated the kind of auxiliary task used in this thesis belongs to two different categories:

- General purposes auxiliary task. This category includes those tasks that give the agent some general abilities to navigate inside the environment. For example, here we can find tasks where the agent has to learn to predict the environment's properties (e.g. predict future observations) or predict action distribution given two visual inputs k frames apart.
- Auxiliary tasks for social navigation (Proximity aware tasks). These kinds of tasks are specific for social navigation, so when there are simulated human beings in the simulated environment. Since the agent must not collide with people, these tasks help it detect the position of people within a certain radius and detect the risk value given by their presence.

Our tests are based on studying how much the introduction of these auxiliary tasks changes the agent's performance.

5.2.1 Learning process of each belief module

The idea behind using auxiliary tasks is that each belief (the output produced by each belief module) can inject different signals inside the embeddings. To learn the weights associated with each belief module, at training time, we have associated each belief module with a unique auxiliary loss jointly optimized with the optimization step of the policy network. This process is done by forecasting a series of auxiliary features $\{\hat{s}_j\}_{j\in[t,t+k]}$ in the range $[t, t+k]$ through a network that computes our auxiliary-task prediction, where k is the number of auxiliary features to predict. The network associated with each auxiliary task is conditioned on the corresponding belief h_t^i and on the sequence of performed actions $\{a_j\}_{j\in[t,t+k]}$. Given a set of auxiliary features $\{s_j\}_{j\in[t,t+k]}$ the task aims to optimize a specific loss for each auxiliary task. In other words, an auxiliary task is an unsupervised task, that incentive the construction of meaningful beliefs h_t to make good predictions about a set of features ${s_j}_{j\in[t,t+k]}$.

The auxiliary features are computed only at training time and at test time the regressor networks are detached.

5.2.2 General-purpose auxiliary task

Here I will describe the general-purpose auxiliary tasks that help the agent to develop some additional abilities in the navigation of the environment.

CPC|A CPC|A [18]is the task that has the goal of predicting future observations (next k observations) conditioned on future actions. This task is represented by a GRU network that takes in input the embedded observations and actions to output a belief state s_t for the current time step t. In the paper, the author fed an MLP network with the belief s_t and positive or negative observations to predict 1 or zero. The actual CPCA task is performed by a transposed CNN. The transposed CNN takes in input the belief state s_t produced by the GRU and the set of previous actions and outputs the next observation o_{t+1}.

GID Generalized Inverse Dynamics (GID) [40], is the task of predicting actions taken between two observations k frames apart (i.e. $\{a_j\}_{j\in[t,t+k]}$) and it is conditioned on the corresponding belief at time step t, s_t, and from the visual embeddings ϕ_{t+k}(i.e.

CNN output). GID predicts individual actions using a separate GRU and two linear layers, f and $f^{''}$. The GRU's output, g_t^{GID} is defined as:

$$g_t^{GID} = f(s_t, \phi_{t+k}) \tag{5.6}$$

The GRU is updated in the following way:

$$g_{t+i}^{GID} = GRU(a_{t+i-1}, g_{t+i-1}^{GID}) \tag{5.7}$$

The loss for this auxiliary task is:

$$L = \sum_{i=1}^{k} CrossEnt(f^{'}(g_{t+i}^{GID}), a_{t+i}) \tag{5.8}$$

PBL Predictions of Bootstrapped Latents (PBL) [14]consist of two predictions:

- a forward prediction from agent's states to future latent observations. With this kind of prediction, the agent is able to learn meaningful observation features into a single representation.
- a reverse prediction from latent observations to agent's state.

Forward prediction trains agent states to predict future latent observations. Reverse prediction trains latent observations to predict immediate agent states. Together, they can form a cycle, bootstrapping useful information between compressed histories and latent observations.

The function that the agent tries to optimize here is:

$$\sum_t ||g'(\phi_t) - s_t||_2^2 \tag{5.9}$$

Where ϕ_t is the embedded observation and s_t is the agent state. g' is a feed-forward network.

5.2.3 Proximity-aware tasks

In this section, I will describe in detail the auxiliary tasks (proximity-aware tasks) that allowed the agent to develop a common-sense social behavior: risk estimation and proximity compass [6]. As we will see the first is more focused on closer social interactions while the latter gives the agent some general information about the direction of the simulated human beings inside the environment.
For both tasks, the loss to optimize is the following:

$$L_f = \frac{\sum_{j \in [t,t+k]} \text{MSE}(s_j, \hat{s_j})}{k} \tag{5.10}$$

Where $\{\hat{s_j}\}_{j \in [t,t+k]} = M(h_t^i, \{a_j\}_{j \in [t,t+k]})$ M is the regressor network 5.3 and s_j is the ground truth feature given by the RISK or SOCIAL COMPASS sensor.

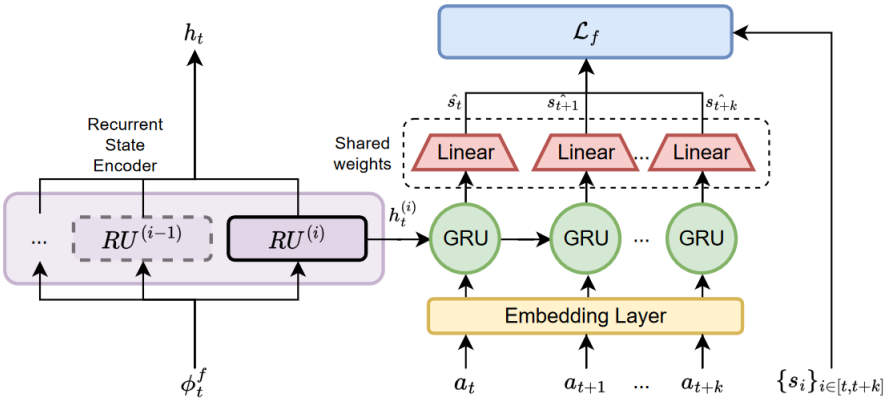

Figura 5.3: The regressor network [6]. It takes in input the set of actions $\{a_j\}_{j\in[t,t+k]}$ and the GRU's belief state. It predicts the set of proximity features $\{\hat{s_j}\}_{j\in[t,t+k]}$ and then is used to compute the associated auxiliary loss L_f with the ground truth auxiliary features $\{s_j\}_{j\in[t,t+k]}$.

.

Risk Estimation Risk estimation is the proximity-aware task designed to deal with short-range social interactions, to inform the agent about imminent collision dangers. The risk value is represented as a scalar value with a range [0,1], that represents how close is the closest human being to the agent within a certain radius D_r. In this sense, the value zero means that the nearest person is further than D_r meters away, and the value one means that the agent and the person are colliding. Formally the risk estimation is defined as:

$$risk_t = clamp(1 - \frac{min||\delta_t^i||_2 \, \delta_t^i \in SI_t}{D_r}, 0, 1) \tag{5.11}$$

Where $clamp(., 0, 1)$ limits the value to [0,1] range and δ_t^i is the position of the person i at timestep t among the set of all the people distances SI_t.

Proximity Compass The proximity compass task models the long-distance component of social interactions. This feature not only captures interactions on a larger area with a radius $Dc > D_r$, but it is also able to give the agent some weak information about the direction of the people within this radius. Thanks to these kinds of features the agent might develop some social abilities that would enable it to perceive the position of the humans inside the environment using only their previous positions.

In this compass, the north is positioned in the direction where the agent is looking and it is partitioned in a series of non-overlapping sectors. Each person $i \in P$ is associated with a specific sector of this compass, by computing its angle position θ_i with respect to the north of the compass. The final proximity compass is given by computing, for each sector k, the risk value by considering all those people that belong to k. Finally, these sectors are unrolled to form a single vector $comp_t$ defined as:

$$\text{compt}[j] = \left[\text{clamp}\left(1 - \frac{\min\left\{\|\delta_i^t\|^2 \mid \delta_i^t \in \Theta_j\right\}}{D_c} 0, 1\right)\right] \tag{5.12}$$

with

$$\Theta_j = \left\{ C\delta_i^t \in SI_t \,|\, \theta_{a\to i} \in \left[\frac{2\pi}{k} \cdot j, \frac{2\pi}{k} \cdot (j+1) \right] \right\}$$

$\forall j \to [0, k-1]$

5.3 The tasks of the experiments

In this thesis I've tackled two important tasks in embodied AI:

- Object goal navigation
- Social object goal navigation

5.3.1 Object goal navigation

The **goal** in social object-goal navigation is to reach an object-goal. The agent is **succesfull** if it calls the action "STOP" with the object goal instance inside its field of view and within 0.1 meters from it.

The **task setup** is the following:

- The agent starts at a random position
- The agent's policy outputs a discrete action belonging to the following set: ["MOVE_FORWARD", "TURN_LEFT", "TURN_RIGH", "STOP"]
- The reward feedback is ObjectGoal oriented
- The agent terminates its episode if one of the following conditions happens:
  - The agent finds the object goal and respects the success condition defined above.
  - The agent took 500 steps in the environment (unsuccessful episode).

5.3.2 Social object-goal navigation

The task of social object-goal is simply an extension of the task seen for object-goal navigation. In social object-goal navigation, the agent has to reach an instance of the object-goal without colliding with simulated human beings. Pedestrians are simulated as [28]. They move towards randomly sampled locations and their movement is simulated using the social-forces model ORCA [2] integrated in the simulator. The social settings used in this thesis are the following:

- The number of people present in our experiments is three.
- Unlike the agent each person moves with continuous actions, in particular, it proceeds with a velocity equal to 1 meter per second.

The **goal** and the **success conditions** are the same as the ones seen in object-goal navigation.

Also, the agent's conditions are the same, so it starts at a random position and gets the same input and the same reward. The termination conditions are a little bit

different. In fact, in addition to the conditions described previously, the agent here can terminate (without success) its episode with a collision with a human. So the human is a completely different entity compared to a classical object, the collision with an object other than a human doesn't terminate the episode. The fact that a human collision terminates immediately the current episode, forces the agent to learn to avoid them. This learning is derived from the fact that the agent loses all the future rewards that it could get without the human collision.

5.4 Implementation details

In this section, I will describe the technical implementation of the tasks and of the RedNet used to approximate the semantic segmentation information during the agent's evaluation.

5.4.1 Object-goal navigation

In the code base, this task is implemented thanks to a series of classes and task-specific sensors. The dataset is loaded using the class "ObjectNavDatasetV1". "ObjectNavDatasetV1" reads a JSON file and returns a dataset that is a dictionary that, for each episode, it stores the goal object category. The episodes belong to the class "ObjectGoalNavEpisode". The goal is an instance of the class "ObjectGoal", which provides information about the object targeted for navigation. This can be specified by an object id, position, or, just like in our case, by an object category. The goal category is given to the agent using "objectgoal" a sensor that belongs to the class "ObjectGoalSensor". This sensor simply returns the id of the object's category, by converting in a NumPy array the dictionary "category_to_task_category_id" that maps a category name to its id.

5.4.2 Social object-goal navigation

The introduction and management of simulated people inside the environment are done by a different simulator and a different task compared to the ones used in classic object-goal navigation. In object-goal navigation, the typical simulator that we use is "Sim-v0", a built-in simulator inside habitat-lab. To introduce a social component we have used the simulator "iGibsonSocialNav". "iGibsonSocialNav" is a subclass of "Sim-V0" (implemented by the class "HabitatSim"), but with the addition of methods like "reset_people" useful to add people in the environment. Each person is an instance of the class "ShortestPathFollowerv2", that, given a series of parameters like the linear velocity or angular velocity, allows it to move from a starting point to an ending point by following a series of waypoints. During the running of the program, the people's movements are managed by the task "SocialNav-v0", using the function "step()".

5.4.3 Semantic segmentations

For both tasks, the evaluation of the agent had different settings compared to its training. The most important is the way the semantic sensor was used. During the training, we used ground truth semantic information provided by the dataset. For a realistic implementation of our agent, its testing could not be done using these data. Imagine if it were in a real-world scenario, since the real world doesn't provide any semantic information about what the agent is seeing, it wouldn't understand the object

that has in front. To overcome this issue, during the testing, instead of using the ground truth provided by the simulator, the agent predicts semantic category from RGBD using a RedNet (Residual Encoder-Decoder Architecture) [22]. The RedNet was originally trained on SUNRGBD [35], and then fine-tuned by [40] on 100K randomly sampled forward-facing views from MP3D, in order to predict 21 goal categories. The problem is that [40] didn't fine-tune the architecture to recognize people, and this results in poor classification performance when we evaluate our social agent in a social context. The problem was overcome by further fine-tuning the network on front-facing views from MP3D that contain also images of human beings.

Fine tuning details

The RedNet fine-tuning comprises two steps:

- Dataset collection
- Actual fine-tuning

Dataset collection The goal of this fine-tuning was to teach the RedNet to recognize people. To do that I've collected some data from the simulator using the MP3D dataset. In particular, the collection was performed by running our best object-goal navigation checkpoint, on a sample of 700 training episodes randomly collected. To avoid biases during the agent evaluation, only training scenes were used to collect the dataset. In these scenes, for each observation, I've collected the output of the RGB sensor (480 x 640 x 3), the output of the depth sensor(480 x 640 x 1), and the semantic information that then will be used as ground truth for the RedNet fine-tuning (480 x 640 x 1). The dimension of each observation is 480 x 640 because the original RedNet was trained with inputs of these dimensions. I've collected both RGB and depth data because, as explained in the previous chapter, to build the semantic map, RedNet exploits RGB and depth information. The dataset was collected with an equal amount of samples that contain a person (or a part of it) and samples that don't contain any people at all. To do that I've used a script that, every time a new sample that doesn't contain a person should be inserted, it checks how many samples with people have been saved. If this quantity, plus a random percentage computed on it, is greater or equal than the number of samples that don't contain any person, then the sample is inserted. The same reasoning can be applied when a sample that contains a person should be inserted. In total, The RedNet was fine-tuned on a 39988 samples and validated at each epoch on 3597 samples.

Fine-tuning The fine-tuning of the network was performed by starting from the checkpoint given by [40]. At this point, the network has been trained on the SUNRGBD dataset and then fine-tuned on 100k mp3d images, but it can't recognize people, since it has never seen them. In order to teach people recognition I've used a script that trains the RedNet on the total train dataset for ten epochs, using batches of size 32. For each batch, the prediction of the semantic mask was computed and then used to compute the loss function. The loss function used in this script is very similar to the one used in the original RedNet paper [20], that is the cross entropy function. The only difference is that instead of computing the cross-entropy loss for each intermediate

output the loss is computed only for the final output:

$$L(s,g) = \sum_{i}^{batch\_size} -log\frac{exp(s_i[g_i])}{\sum_{c=1}^{C} exp(s_i[c])} \tag{5.13}$$

Where g_i and s_i are the target mask and the predicted mask at i^{th} batch sample and C is the number of classes.
As described in the previous chapter, the RedNed is composed of an encoder layer that is useful for extracting semantic information and a decoder layer that is used to compute the actual semantic map starting from the extracted semantic features. In order to avoid the destruction of any information used to extract the semantic feature in the encoder, all the layers of the encoder were frozen, so the gradient is not computed for these layers, and only the layers of the decoders, the ones that compute the semantic map, were fine-tuned. With this strategy, the decoder will turn the old features into predictions for the new dataset that contains also people without affecting the actual semantic feature extraction.

The dataset was augmented using different operations such as rotation, horizontal flip, and vertical flip. All these operations were performed using the library "Albumentations" [5]. Albumentations is a free and open-source Python library for image augmentations. It supports different computer vision tasks like classification, semantic segmentation, instance segmentation, object detection, and pose estimation.

Capitolo 6

Experimental results

In this chapter, I will describe the results that I've obtained in object-goal navigation and social object-goal navigation tasks. The policies that I've trained can be categorized by the type of auxiliary tasks they used, they are the following:

- *no_proximity*. Agent trained using only the auxiliary task (CPCA, PBL and GID).
- *risk*. Agent trained using only risk estimation as a proximity task.
- *compass*. Agent trained using only social compass as proximity task.
- *proximity*. Agent trained both risk and compass as proximity tasks.

Each policy has all the standard auxiliary tasks (PBL, GID and CPCA) but different proximity-aware tasks subsets. *risk*, *compass* and *proximity* are all trained in social object-navigation task. *no_proximity* has been trained both in an object navigation setting and in a social object-goal navigation task. In this regard, we will call:

- *social_no_proximity*: *no_proximity* policy that was trained in social object-navigation task.
- *obj_no_proximity*: the *no_proximity* policy that was trained in standard object-goal navigation task.

Each policy was evaluated on a validation dataset with scenes never seen by the agent before. For some policies, we have also tried to evaluate them in a type of environment different from where they have been trained. For example, *obj_no_proximity* was also evaluated in a social scenario, and vice-versa. These tests were conducted in order to understand how the policies behave in the context where they don't have any experience.

Each agent is trained using 3 GPUs in 3 different nodes, for a total of $\sim 125M$ frames. The training of each agent lasted about 15 days.

In each training, a checkpoint was saved every $\sim 12M$ frames for a total of 10 checkpoints. Then, each checkpoint was evaluated on a subsample of 500 episodes of unseen scenes with 4 different seeds to get statistically accurate results. The results that I will present in the following section are given by the checkpoint that, on average, performed better in these 4 runs. The best checkpoint was selected based on the SPL

metric, that is the percentage of success (i.e. find an instance of the object goal) of the agent weighted by its path length. For some policies, we wanted to check how much the usage of the RedNet influences the policy performances compared to the ground truth semantic segmentation. To test it, we took the best checkpoint and evaluated it using semantic segmentation information given by the dataset.

The experiments were tracked using Weight and Biases. An MLOps tool is used to track all the experiments and visualize the data (such as the agent's input).

This chapter is divided into two parts: in the first part I will discuss the result obtained by the policy trained in an object-goal navigation context (without the presence of people in the environment), while in the second part, I will discuss the results obtained by the policies trained in a social setting.

6.1 Object-nav agent results

obj_no_proximity results				
Evaluation setting	Metrics MP3D			
	Success	SPL	H-Coll	PSC
obj-nav	**16,28%** ±1,28	**5,70%** ±**0,33**	-	-
social obj-nav	12,25% ±2,88	4,78% ±1,27	**36%** ±**1,75**	**12,10%** ±**2,84**

Tabella 6.1: Results obtained by *obj_no_proximity* policy in different tasks evaluation.

In this section, we will describe the results obtained by the *obj_no_proximity* policy. The results that I'm going to describe in this section refer to the table 6.1.

With the policy trained and evaluated in an environment without simulated human beings, We have achieved a success rate of 16,3% and an SPL of 5,7% (row 1 6.1). We have evaluated the same policy in a context that has never been before: with the presence of human beings in the environment. With this experiment, we were able also to evaluate the capability of policy's generalization. From the results, the best checkpoint in this context achieved a success rate of 12,25% and an SPL of 4,78% (row 2 6.1). We can notice that, despite the presence of moving obstacles inside the environment, the performances didn't degenerate too much, and this means that the policy can generalize not only in unseen environments but also in unseen situations.

Another aspect that we can notice is that the presence of moving people inside the environment doesn't seem to affect the optimality of the path followed by the agent. This can be inferred from the fact that both the success and the SPL of the agent that has been evaluated in a social environment has scaled by a similar factor (1,3 and 1,2 for success and SPL respectively [1]) respect to the ones obtained by policy evaluated

[1]Here I consider that, for each successful episode, the obtained SPL won't differ too much. To test this hypothesis I've collected 120 non-zero spl and I've computed the coefficient of variation that is a standard measure of dispersion for a distribution. It is defined as $\frac{\sigma}{\mu}$ where σ is the standard deviation and μ is the mean. By computing the coefficient of variation on the collected SPLs I've obtained a value of $\sim 0,5$. In general distributions with a coefficient of variation less than one are considered to be low-variance.

in the obj-nav scenario. This means that the agent, in the social context, followed the optimal path at the same rate as when it finds itself in a no-social context. This conclusion has also another implication: The fact that the optimal trajectory following rate didn't change too much means that the agent doesn't consider the presence of humans in the environment. If it would then the spl should have gone down more (higher scaled factor) because to dodge a human the agent had to deviate from the optimal path, resulting in a lower SPL.

Another test is to check how much the performances change in the case where ground truth semantic segmentation are used, instead of the ones computed by the rednet RedNet (6.2 first column). As we can see from the table,the success almost doubled (31,2%) and the SPL is more than doubled (11,8%). The fact that the metrics were higher, was expected. For two main reasons:

- The ground truth semantic segmentations are more precise and less noisy than the ones produced by the RedNet. A noisy input gives the agent a more complex signal and less stable recurrent dynamics. This factor brings to a minor capability of main memory over long-term trajectory [40].
- The agent is originally trained using the ground truth semantic segmentation.

6.1.1 Policy generalization test

When transferred from training to evaluation, the policy had to tackle two main difficulties:

- The first one is the fact that it has to face environments that it has never seen before.
- The second one is that it doesn't have the semantic segmentation information provided by the dataset, like in the training, but the information is approximated using the RedNet.

We wanted to check how much, each of these two factors, influences the policy performances, so we studied them individually. To do that we also evaluated our model in these two settings:

- Evaluation on the training set with ground truth semantic segmentation
- Evaluation on the training set with approximated semantic segmentation (RedNet)

	VAL	TRAIN
Success	16,3%(31,2%)	35,20%(46%)
SPL	5,7%(11,8%)	18,8% (23,8%)

Tabella 6.2: Here I summarize the results obtained by the *obj_no_proximity* policy in the training set and in the validation set with the usage of the RedNet or ground truth semantic segmentation. The data is reported as "with RedNet segmentation (with GT segmentation)". The results in the training set were collected using the best checkpoint given by the evaluation procedure. The evaluation setting used is obj-navigation

In 6.2 we can see a summary of the results obtained by using the *obj_ no_ proximity* policy evaluated in the training and evaluation dataset using ground truth semantic segmentation or RedNet semantic segmentation. From these results, we can infer some considerations. First of all, we can see that as long as the agent has ground truth semantic segmentation it was able to generalize well, especially regarding the success rate, passing from 46% (training set) to 31,2% (validation set). It struggles to follow the optimal path to the object goal by passing from an SPL rate equal to 23,8% (training set) to an SPL equal to 11,8% (validation set). Concerning the results obtained with the RedNet, we pass from a success rate equal to 46% (GT semantic) to a success rate equal to 35,2% (RedNet), and from an SPL equal to 23,8% (GT semantic) to an SPL equal to 18,8% (RedNet). The main problems emerge when we evaluate the agent on the validation set using the RedNet as a semantic segmenter. If we confront the results obtained using the RedNet and the ground truth in the validation set we would expect to see a difference that resembles the difference between the usage of the RedNet and the ground truth in the training set, but in the validation set the gap is larger (14.9% vs 10,8% loss for success and 5,1% vs 5% loss for SPL). These missing performances can be justified by the fact that the RedNet was trained using data collected from the mp3d's training partition. To test how much performance was lost using the RedNet in the mp3d's evaluation partition, I've tested its accuracy in this scenario.

As we can see from 7.3 the gap between validation and training set is small but considerable. This difference implies a larger gap between the difference in performance obtained by the agent using ground truth semantic segmentation and Rednet semantic segmentation between Matterport's validation and training set.

6.2 Social object-goal navigation results

In this section, I will describe and examine the results obtained by social agents. The task that these agents have to solve is the same as the one performed by the previous agent, the *obj_ no_ proximity* agent. The only difference here is that they have to solve it with the presence of simulated human beings inside the environment.

This section will be divided into two parts. In the first part, I'm going to compare the results, in social object-goal navigation, obtained by *obj_ no_ proximity* and *social_ no_ proximity*. While in the second part, I'm going to analyze the contribution of the proximity-aware tasks during the policy training.

6.2.1 Social train and no-social train comparison

no_proximity eval. in social object-goal navigation (social vs no-social train mode)				
Training Mode	Metrics MP3D			
	Success	SPL	H-Coll	PSC
no social	12,25% ±2,88	4,78% ±1,27	**36% ±1,75**	**12,10% ±2,84**
social	**13,4% ±1,00**	**5,24% ±0,36**	49,9% ±3,61	13,2% ±1,03

Tabella 6.3: Metrics obtained by *obj_ no_ proximity* (row 1) and *social_ no_ proximity* (row 2) in social object-goal navigation.

I will start by comparing the results obtained by *obj_no_proximity* and *social_no_proximity*, evaluated in social object-goal navigation task (6.3). As expected *social_no_proximity* achieved better performance than *obj_no_proximity*, especially if we consider the success rate and the SPL (13,4% vs 12,25% for success and 5,24% vs 4,78% for the SPL).
Interestingly, the human collision (H-Coll) obtained by the *obj_no_proximity* is much lower (36%) than the one obtained by the other policy (49,9%).

Recalling that a collision with a human leads to an immediate failure (so the agent hasn't got any opportunities to continue the task once it collides with a human being) and since *social_no_proximity* has a higher success rate compared the one obtained by *obj_no_proximity*, we can infer that, for the subset of episodes where there isn't a collision with a human, the first policy has a greater capability of reaching the goal.

no_proximity eval. in object-goal navigation (social vs no-social train mode)		
Training Mode	Metrics MP3D	
	Success	SPL
no social	16,28% ±1,28	5,70% ±0,33
social	**21,20% ±2,46**	**7,80% ±0,95**

Tabella 6.4: Metrics obtained by *obj_no_proximity* (row 1) and *social_no_proximity* (row 2) in object-goal navigation.

To test this last hypothesis I've evaluated *social_no_proximity* on object-goal navigation tasks. The results that I've obtained are shown in 6.4. As we can see *social_no_proximity* in object-goal navigation outperforms *obj_no_proximity* in both success (21,2 % vs 16,28%) and SPL (7,80% vs 5,70%). These results match our previous hypothesis: *social_no_proximity* is a policy with greater capabilities in terms of object-goal navigation than *obj_no_proximity*.

We hypothesize that social training makes the policy greedier, and so, it tries to find the goal as fast as possible. This behavior is justified by the fact that, since the social policy has been trained in a social context, the agent doesn't want to explore too much because the greater the exploration of the environment the greater the probability of finding a human, colliding with it and loosing the future rewards. This dynamic makes the policy greedier preferring the immediate reward instead of the future reward. This last statement is supported by watching the figure 6.1. In this figure, I've compared three parameters(num_step, visit_count and coverage_step) for both *obj_no_proximity* and *social_no_proximity*. These three parameters represent how much the agent explores the environment:

- num_steps is the average number of steps done by the agent in each episode
- visit_count is the average number of times that the agent visits the same region in the environment
- coverage_step is the number of steps that the agent performed to explore the environment

The lower these three parameters are the less the agent explores the environment. From 6.1 we can see that all three parameters are lower for the *social_no_proximity*, especially visit_count. This means that not only does it explore less the environment but it is also more efficient because it doesn't return to a region that it has already

visited.

Environment exploration is useful in contexts where we have to deal with complex and large environments. The scenes proposed by Matterport 3D are mainly composed of small interiors with not a large number of big rooms. An open point in this regard is that if *obj_no_proximity*, thanks to its greater exploration, behaves better than *social_no_proximity* as we increase the general environments' size.

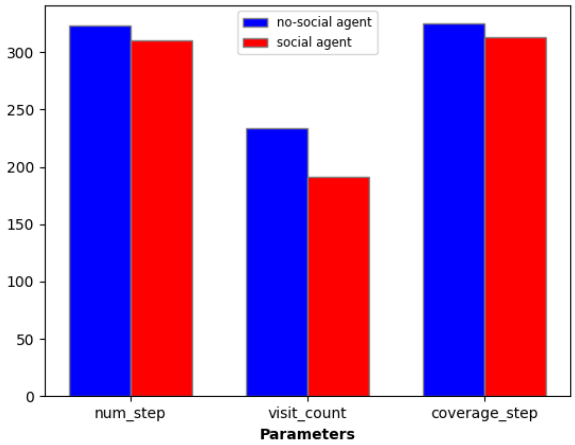

Figura 6.1: Plot that displays the three main parameters (num_steps, visit_count and coverage_step) in order to check the exploration of *social_no_proximity* (red bar) and *obj_no_proximity* (blue bar) in object-goal navigation setting. *social_no_proximity* has lower values in all three parameters, thus suggesting a greedier policy.

6.2.2 Proximity tasks study

Method	Proximity		Metrics MP3D			
	Risk	Compass	Success	SPL	H-Coll	PSC
no_proximity(social train)			13,4 % ±1,00	5,24%±0,36	49,9%±3,61	13,2%±1,03
risk	✓		**14,8%±0,36**	**6,0%±0,36**	**44,15%±0,36**	14,68%±0,36
compass		✓	7,89% ±3,40	3,52% ±1,2	45,72%±5,20	**7,81% ±3,43**
proximity	✓	✓	12,8% ±1,82	5,8% ±0,74	49,4% ±3,51	12,67% ±1,77

Tabella 6.5: A summary of the metrics obtained in social object-goal navigation task with proximity tasks configurations.

Here we are going to analyze how much, and in which form, the introduction of proximity tasks (proximity compass and risk estimation) has changed the agent's behavior. In this subsection, I will first compare the policy that doesn't use any proximity task (*social_no_proximity*) with the policy that uses all the proximity tasks (*proximity*), then I will study how the introduction of only a subset of these proximity tasks (*compass* and *risk*) changes the agent's performances. In the last part of this section, I will briefly make some considerations on proximity tasks' optimization.

Proximity tasks contibution

Here we will directly compare *social_no_proximity* and *proximity*. From the metrics in 6.5 (row 1 and 4) we can see that the two policies are not very different, the success is similar (13,4% vs 12,8%) just like the SPL (5,24% vs 5,8%), the human collision(49,9% vs 49,4%) and the PSC (13,2% vs 12,67%). We will later discuss why the difference between *social_no_proximity* and *proximity* is not larger.

Table 6.5 portrays the results given by the best checkpoint in 4 different evaluations with 4 different seeds. In general, the best results are given by the last checkpoint since they represent policies with more experience.

What we have noticed is that *social_no_proximity* achieves good results only in the last few checkpoints, while *proximity* is superior for the majority of all the other checkpoints as depicted in 6.2. This result might suggest that *proximity*, thanks to the addition of the proximity tasks, learns the task faster and can achieve better performances with fewer updates than *social_no_proximity*.

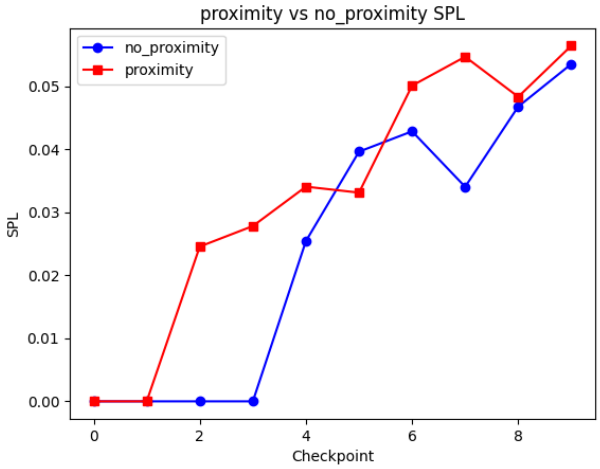

Figura 6.2: In this figure we can see the SPL achieved for each checkpoint. As we can see the SPL achieved by *proximity* (red line) is above in almost all the checkpoints with respect to *no_proximity* (blue line).

Individual proximity task study

In this section, we will study the results obtained by the policies equipped only with a proximity tasks subset (i.e. *risk* and *compass*).
From their results (rows 2 and 3 6.5) we can see that they are at the extremes of the performance spectrum we achieved for social object goal navigation. *risk* is the policy that achieved the best results overall (SPL: 6,0%, success: 14,8%), while *compass* is our weakest policy in terms of object goal navigation performances (SPL: 3,52%, success: 7,89%). Both policies achieve similar results for what concern social abilities such as human collision (44,15% vs 45,72% for *risk* and *compass* respectively) and PSC (14,68% vs 7,81% for risk and compass respectively).
This difference in performance can also be seen during the training of these two policies, with *compass* having more difficulties in actually learning the task. In 6.3 we can see the curves that represent the success rate for both policies during the training. *compass* started to have some success (i.e. find the object goal) only after $\sim$ 23000 updates, the double respects to the ones required by *risk*. This means that:

- Given a fixed amount of training steps, the final performance will be worse for *compass*, because it started to learn how to accomplish the task only in the end.
- The number of updates required by compass to get the same performances as the ones obtained by risk are higher. Here we have a similar situation as described in 6.2.2, where one policy (*risk*) is more efficient in terms of the number of updates to accomplish the task, than another (*compass*).

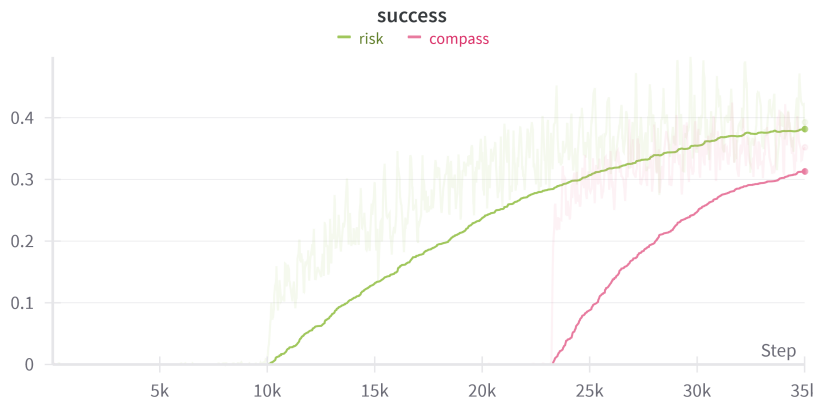

Figura 6.3: The training graphs (success rate) of risk (green) and compass (pink). The first agent started to be successful in the task much before respect to the second agent.

Possible causes What can cause this difference in performance? The policy parameters are the same, the only difference is in one proximity task: one policy uses risk and the other uses proximity compass. What we suspect is that the *compass* performance degradation is caused by a poor proximity compass hyperparameters selection. Proximity compass hyperparameters are retrieved from [6] without further changes. In [6] they used Gibson4+ and Habitat-Matterport3D as datasets, that have different characteristics in terms of the size of the environment or percentage of open space environment compared to the dataset used in this thesis (Matterport3D). The proximity compass is sensible to these characteristics since it detects the presence of other people within a long-range radius, so a difference in the size of the environment would also require a change in the parameter that regulates the radius range (D_c). This could be also the cause of why the performances of *proximity* are not so different from the ones obtained by *social_no_proximity*. Further studies are required to test this hypothesis and select an optimal hyperparameter set through a cross-validation process.

RedNet performances (social part)

In this part, I will briefly analyze RedNet' 's social performances (i.e. how good it is to recognize people). This social classification is crucial for the agent's social performance. If it learns to avoid contact with humans during the training but can't recognize them in the validation, then its social performance will surely drop.

To test how well the RedNet can do that, we evaluate its accuracy. In particular, we evaluate its capability to recognize a person.
From 7.4 we can see that the RedNet is able to perform semantic segmentation on people with good performances in both the training and the validation dataset, by achieving in both cases an accuracy above 92%. This means that for what concerns social abilities, for the social agent it doesn't change too much from using the ground

truth or the RedNet, and this can be translated in similar social metrics between the agent that uses ground truth semantic information and the one that uses the RedNet.

To test this hypothesis we tested *compass* with ground truth semantic segmentation instead of RedNet semantic segmentation6.6. The first thing that we can notice is that the RedNet doesn't seem to affect too much social performance. The personal space compliance (with respect to the corresponding success) and the human collision are similar, with or without the usage of the RedNet in both the validation and training sets (row 3 and 4). This small gap in the social performances between the agent that uses the RedNet and the agent that uses the ground truth semantic information means that the fine-tuned worked well. The agent didn't find any difficulties in passing from ground truth information to semantic information provided by the RedNet, at least for what concerns the social component.

Regarding the policy generalization to never-seen environments, the social agent has less performance degradation passing from the training set to the validation set than *obj_no_proximity*.

For what concerns the generalization of the agent's social abilities in never-seen environments, the agent manages to keep a good PSC. Still, the human collision increases, passing from 37,75% to 45,25% (GT segmentation) and from 38% to 41,97% (RedNet segmentation).

	VAL	TRAIN
Success	9,38%(16,50%)	18,00%(28,5%)
SPL	3,79%(6,50%)	8,09% (14,76%)
HC	41,97%(45,25%)	38%(37,75%)
PSC	9,20%(16,39%)	17,76%(27,92%)

Tabella 6.6: Results obtained in the training set and in the validation set with the usage of the RedNet or ground truth semantic segmentation of the *compass*. The data is reported as "with RedNet segmentation (with GT segmentation)". The results in the training set were collected using the best checkpoint given by the evaluation procedure.

6.3 Quantitive results

In this section, I will show a successful and failed episode. In particular, I will show how *risk* agent seems to have developed a strategy to avoid contact with humans and a cause of failure that doesn't depend on the policy that the agent has learned. In both images, the agent is depicted as an arrow and the rest are people. The path taken by the agent during its traversal is blue while the path followed by the human is red. The red areas of the images represent the goal position for that episode.

6.4 is an example of an episode failure due to human contact. In this episode, the agent starts at a random position (top), it goes straight (middle), but then it collides with a human that comes from a blind spot (bottom). These collisions don't depend on the agent, since it doesn't know the human path (red path) and didn't see it coming. Collisions of these types represent a significant part of the totality.

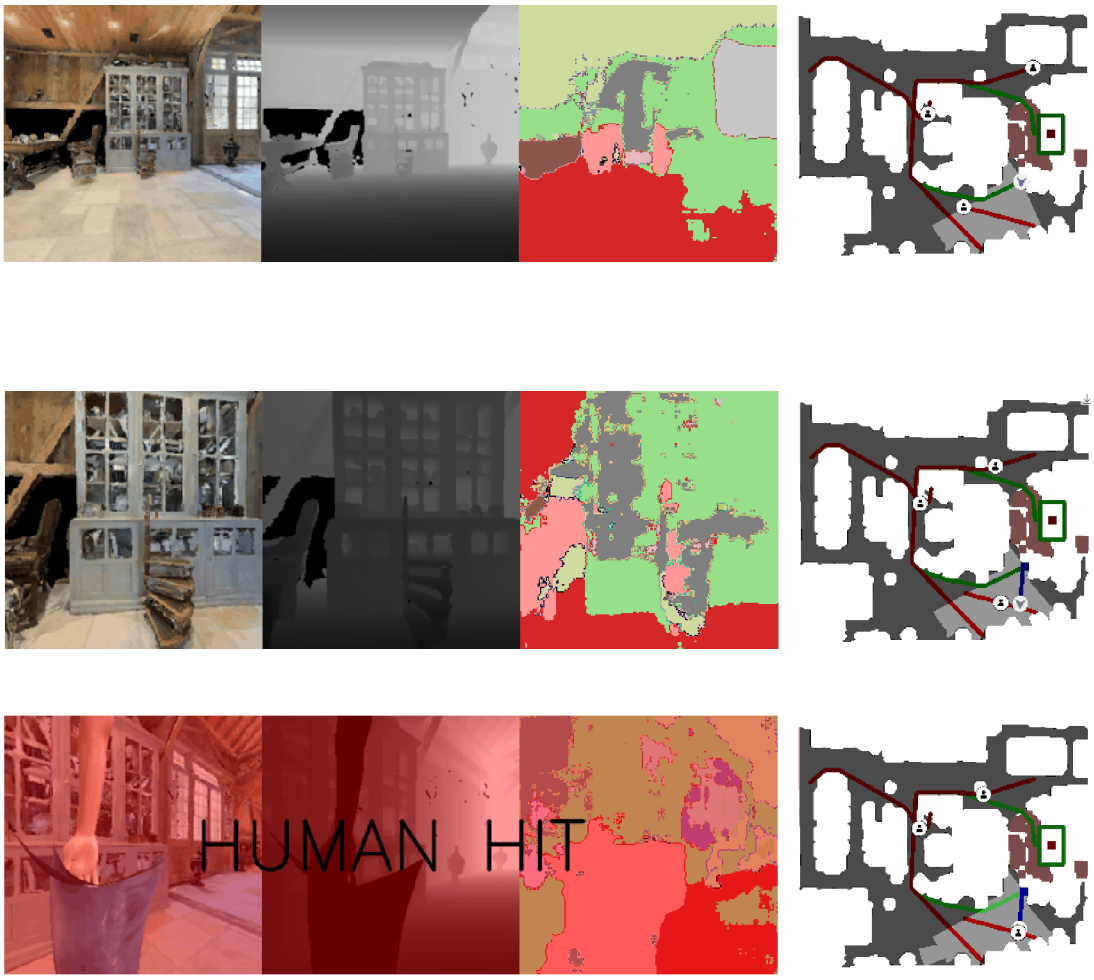

Figura 6.4: Episode failure

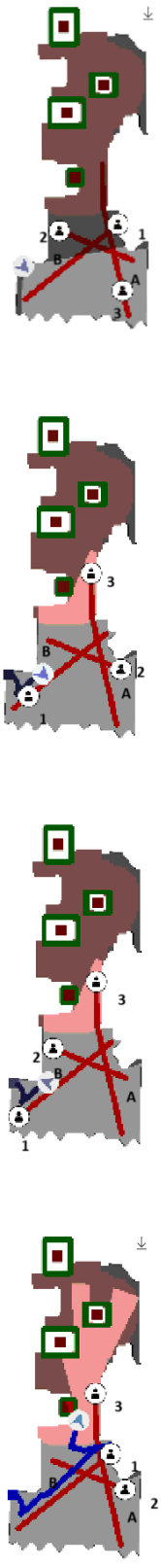

Figura 6.5: Strategy developed by the agent to avoid people.

In 6.5we can see what seems a strategy developed by the agent *risk* to avoid contact with humans. Despite the presence of three simulated human beings inside a restricted area, the agent managed to avoid contact with them by reaching its goal. The agent starts at a random position (first image), moves to not cross the path of person "1" (second image), and then finds itself in front of the path of person "2". If the agent kept going it would collide almost surely with the person "2". The agent saw the person "2" moving from "B" to "A" so instead of continuing and risking a collision, it started to spin to keep the same position. When person "2" returned to point "B" the agent continued its traverse immediately. The same behavior happened with the person "3". When it reached the red spot (its goal, in this case, a chair), it stopped, by calling the action "STOP" and the episode terminated. Further studies are required to understand the percentage of times that the agent applies this strategy to avoid contact with humans and not get too close to them.

Capitolo 7

Appendix - Rednet performance

In this section, I will briefly describe the results that I've obtained with the RedNet. In particular, I will show its accuracy in semantic segmentation with or without the presence of people in the environment.
The RedNet performances are given by computing the accuracy of the predicted mask with respect to the target mask of a specific sample.
The accuracy is first computed on the whole image (the average of the accuracy among all the classes, 22 classes, that appear in a particular sample), and then I computed the accuracy only for the class "person".
This choice was made because fine-tuning goal was to teach to the RedNet people recognition, and thanks to this last metric, I had the chance to check the actual progress on this front [1].

In 7.1 we can see the semantic segmentation provided by the not fine-tuned (top) and by the fine-tuned (bottom) RedNet. First of all, we can see that the semantic input in both images is much noisier than the ones depicted in 5.2. This is because in 5.2 the semantic information was retrieved from the dataset (unrealistic scenario) while here the semantic information is retrieved from the RedNet output. We can see that in the top image, the RedNet assigns to the person a not uniform semantic mask, a mask that contains different classes (the person is colored with different colors). In the bottom image instead, the person has a uniform segmentation mask, where the RedNet assigns a single class to the person (the person is colored with a single color).

[1] The general accuracy is computed by counting all the predicted pixels that are equal to the target pixels and then I divided this quantity by the total of all the pixels evaluated. The accuracy for the class "person" was performed similarly: in the target mask, I selected those portions of the image that contain a person, counted how many predictions were done correctly on those portions, summed the total pixels of the selected regions, and then divided the corrects predictions by the total pixels.

Figura 7.1: Output provided by the not fine-tuned RedNet (top) and the output provided by the fine-tuned RedNet(bottom). As we can notice the not fine-tuned RedNet is not able to recognize correctly the class person by assigning multiple classes to it.

7.0.1 Training performance

Here I will present the results obtained during the fine-tuning RedNet's validation. The results are given by evaluating the RedNet on an hold-out ($\sim 20k$ samples) of the training dataset. Both the training set and the validation set are composed of scenes extracted from the Matterport's training set.
First I will present the results obtained using the checkpoint provided by [40] in the validation set 7.1. As we can see from the table at the beginning the rednet reached a good general accuracy but bad performances when it has to recognize a person.

	General accuracy	People accuracy
Accuracy	66,23%	2,26%

Tabella 7.1: General accuracy and people accuracy obtained by the checkpoint provided by [40]

In 7.2 we can see the results obtained by the fine-tuned checkpoint. As expected the network learnt to recognize people, by achieving an accuracy of 92,2% for the corresponding class. The general accuracy instead decreased (63,24%). These metrics suggest that recognizing a person is generally easier than the average of all the other objects, probably due to the very different shape that a person has compared to the other inanimate objects inside an indoor environment.

	General accuracy	People accuracy
Accuracy	63,24%	92,2%

Tabella 7.2: General accuracy and people accuracy obtained by the fine-tuned checkpoint.

7.0.2 RedNet generalization

In this section, I will compare the performances obtained by the fine-tuned RedNet in the MatterPort's training partition and MatterPort's validation partition.
This comparison is done for both general accuracy and person accuracy. As we can see from 7.3 the RedNet performances don't change too much from MatterPort's training and validation dataset.

	VALIDATION	TRAIN
Accuracy	58,53%	63,24%

Tabella 7.3: I've evaluated the general RedNet's accuracy on mp3d's validation partition. The validation was done on $\sim$ 9000 MatterPort's validation sample. It's worth mentioning that I've obtained a lower accuracy in the validation dataset with respect to the one obtained in the original paper (69,01%)

In 7.4 we can see the RedNet's capability to recognize people inside MatterPort's training and validation dataset. As we can notice the performance of the network in this task doesn't change too much from the training set (92,19%) and the validation set (92,04%).

	VAL	TRAIN
People accuracy	92,044%	92,19%

Tabella 7.4: I've tested the capability of the network to recognize correctly the people inside the environment. The network has good performances both in the MatterPort's training and validation set. The validation was performed using $\sim$ 9000 samples from mp3d's validation dataset.

Capitolo 8

Conclusions

In this thesis, I've covered the main techniques to train and develop an Embodied AI agent. In particular, I've explained what is reinforcement learning and how it can be used in this field. I've introduced why Embodied AI is studied and two embodied AI tasks: object-goal navigation and its variant, social object-goal navigation. I've conducted several experiments on the model proposed by [40]. In particular, I've tested it on object-goal navigation and social object-goal navigation. The tests were performed using a realistic scenario where the semantic information was not given by the dataset but was approximated using the RedNet, a network for semantic segmentation. In this setting, the obj-nav agent reached a Success rate of 16,3% and an SPL of 5,7%. The social object-navigation task was tackled by introducing a special set of auxiliary tasks called proximity tasks (risk estimation and proximity compass). Thanks to their introduction we managed to reach a Success rate of 14,8% and an SPL of 6,0%. We studied the effects of these proximity tasks on the agent. In particular, the introduction of the proximity compass seems to decrease the performance, probably due to poor hyperparameter selection. Despite that, the policy with all the proximity compass enabled achieved better performances with respect to the one without any proximity task. Moreover, we studied how the different training condition affects the policy behavior. In particular, we have noticed that training in a social environment makes the policy greedier, and so less prone to exploration. In this last context we have reached a success of 21,20% and an SPL equal to 7,80%. For each of our experiments, we also measured how the introduction of the RedNet changes the results.
Many open points should be addressed in the future:

- What kind of performances can reach the social agent? In our test, we managed to get good results but some aspects can be improved (e.g. human collision). These aspects could improve with a proper hyperparameter selection.
- We saw in the quantitive results how the social agent (*proximity*) uses a technique to avoid contact with humans. How often does it do it compared to other policies?
- We have seen that training in a social environment leads to a greedier policy that performs less exploration. With the settings that we have used this dynamics leads to better performances (Success and SPL). Can this trend change by changing the type of the environment? If the policy performs less exploration of the environment then it might find some difficulties in larger environments where a good exploration of the environment is crucial.

Bibliografia

[1] Edward Beeching et al. «Learning to plan with uncertain topological maps.» In: *European Conference on Computer Vision*. 2020.

[2] Jur van den Berg, Ming Lin e Dinesh Manocha. «Reciprocal Velocity Obstacles for real-time multi-agent navigation». In: *2008 IEEE International Conference on Robotics and Automation*. 2008, pp. 1928–1935. DOI: 10.1109/ROBOT.2008.4543489.

[3] Mariusz Bojarski et al. «End to End Learning for Self-Driving Cars». In: *CoRR* abs/1604.07316 (2016). arXiv: 1604.07316. URL: http://arxiv.org/abs/1604.07316.

[4] Yuri Burda et al. *Large-Scale Study of Curiosity-Driven Learning*. 2018. arXiv: 1808.04355 [cs.LG].

[5] Alexander Buslaev et al. «Albumentations: Fast and Flexible Image Augmentations». In: *Information* 11.2 (2020). ISSN: 2078-2489. DOI: 10.3390/info11020125. URL: https://www.mdpi.com/2078-2489/11/2/125.

[6] Enrico Cancelli et al. *Exploiting Proximity-Aware Tasks for Embodied Social Navigation*. 2023. arXiv: 2212.00767 [cs.CV].

[7] Angel Chang et al. *Matterport3D: Learning from RGB-D Data in Indoor Environments*. 2017. arXiv: 1709.06158 [cs.CV].

[8] Devendra Singh Chaplot et al. «Object Goal Navigation using Goal-Oriented Semantic Exploration». In: *CoRR* abs/2007.00643 (2020). arXiv: 2007.00643. URL: https://arxiv.org/abs/2007.00643.

[9] Changan Chen et al. «Crowd-Robot Interaction: Crowd-aware Robot Navigation with Attention-based Deep Reinforcement Learning». In: *CoRR* abs/1809.08835 (2018). arXiv: 1809.08835. URL: http://arxiv.org/abs/1809.08835.

[10] Kyunghyun Cho et al. *On the Properties of Neural Machine Translation: Encoder-Decoder Approaches*. 2014. arXiv: 1409.1259 [cs.CL].

[11] Jiafei Duan et al. «A Survey of Embodied AI: From Simulators to Research Tasks». In: *CoRR* abs/2103.04918 (2021). arXiv: 2103.04918. URL: https://arxiv.org/abs/2103.04918.

[12] Ian Goodfellow, Yoshua Bengio e Aaron Courville. *Deep Learning*. http://www.deeplearningbook.org. MIT Press, 2016.

[13] Daniel Guo et al. *Bootstrap Latent-Predictive Representations for Multitask Reinforcement Learning*. 2020. arXiv: 2004.14646 [cs.LG].

[14] Daniel Guo et al. *Bootstrap Latent-Predictive Representations for Multitask Reinforcement Learning*. 2020. arXiv: 2004.14646 [cs.LG].

[15] Zhaohan Daniel Guo et al. «Bootstrap Latent-Predictive Representations for Multitask Reinforcement Learning». In: *CoRR* abs/2004.14646 (2020). arXiv: 2004.14646. URL: https://arxiv.org/abs/2004.14646.

[16] Zhaohan Daniel Guo et al. «Neural Predictive Belief Representations». In: *CoRR* abs/1811.06407 (2018). arXiv: 1811.06407. URL: http://arxiv.org/abs/1811.06407.

[17] Zhaohan Daniel Guo et al. *Neural Predictive Belief Representations*. 2019. arXiv: 1811.06407 [cs.LG].

[18] Zhaohan Daniel Guo et al. *Neural Predictive Belief Representations*. 2019. arXiv: 1811.06407 [cs.LG].

[19] Saurabh Gupta et al. *Cognitive Mapping and Planning for Visual Navigation*. 2019. arXiv: 1702.03920 [cs.CV].

[20] Kaiming He et al. *Deep Residual Learning for Image Recognition*. 2015. arXiv: 1512.03385 [cs.CV].

[21] Sepp Hochreiter e Jürgen Schmidhuber. «Long Short-Term Memory». In: *Neural Computation* 9.8 (nov. 1997), pp. 1735–1780. ISSN: 0899-7667. DOI: 10.1162/neco.1997.9.8.1735. eprint: https://direct.mit.edu/neco/article-pdf/9/8/1735/813796/neco.1997.9.8.1735.pdf. URL: https://doi.org/10.1162/neco.1997.9.8.1735.

[22] Jindong Jiang et al. *RedNet: Residual Encoder-Decoder Network for indoor RGB-D Semantic Segmentation*. 2018. arXiv: 1806.01054 [cs.CV].

[23] Abhishek Kadian et al. «Are We Making Real Progress in Simulated Environments? Measuring the Sim2Real Gap in Embodied Visual Navigation». In: *CoRR* abs/1912.06321 (2019). arXiv: 1912.06321. URL: http://arxiv.org/abs/1912.06321.

[24] Sham Kakade e John Langford. *Approximately Optimal Approximate Reinforcement Learning*. Proceedings of the Nineteenth International Conference on Machine Learning, 2002.

[25] Christoforos Mavrogiannis et al. «Effects of Distinct Robot Navigation Strategies on Human Behavior in a Crowded Environment». In: mar. 2019. DOI: 10.1109/HRI.2019.8673115.

[26] Dmytro Mishkin, Alexey Dosovitskiy e Vladlen Koltun. «Benchmarking Classic and Learned Navigation in Complex 3D Environments». In: *CoRR* abs/1901.10915 (2019). arXiv: 1901.10915. URL: http://arxiv.org/abs/1901.10915.

[27] Medhini Narasimhan et al. *Seeing the Un-Scene: Learning Amodal Semantic Maps for Room Navigation*. 2020. arXiv: 2007.09841 [cs.CV].

[28] Claudia Pérez-D'Arpino et al. «Robot Navigation in Constrained Pedestrian Environments using Reinforcement Learning». In: *CoRR* abs/2010.08600 (2020). arXiv: 2010.08600. URL: https://arxiv.org/abs/2010.08600.

[29] Julio A. Placed et al. *A Survey on Active Simultaneous Localization and Mapping: State of the Art and New Frontiers*. 2023. arXiv: 2207.00254 [cs.RO].

[30] Dean Pomerleau. «ALVINN: An Autonomous Land Vehicle In a Neural Network». In: *Proceedings of (NeurIPS) Neural Information Processing Systems*. A cura di D.S. Touretzky. Morgan Kaufmann, 1989, pp. 305–313.

[31] Santhosh K. Ramakrishnan, Ziad Al-Halah e Kristen Grauman. *Occupancy Anticipation for Efficient Exploration and Navigation*. 2020. arXiv: 2008.09285 [cs.CV].

[32] Manolis Savva et al. *Habitat: A Platform for Embodied AI Research*. 2019. arXiv: 1904.01201 [cs.CV].

[33] John Schulman et al. *Trust Region Policy Optimization*. 2017. arXiv: 1502.05477 [cs.LG].

[34] Linda Smith e Michael Gasser. «The Development of Embodied Cognition: Six Lessons from Babies». In: *Artificial life* 11 (dic. 2005), pp. 13–29. DOI: 10.1162/1064546053278973.

[35] Shuran Song, Samuel P. Lichtenberg e Jianxiong Xiao. «SUN RGB-D: A RGB-D scene understanding benchmark suite». In: *2015 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. 2015, pp. 567–576. DOI: 10.1109/CVPR.2015.7298655.

[36] Richard S. Sutton e Andrew G. Barto. *Reinforcement Learning: An Introduction*. Second. The MIT Press, 2018. URL: http://incompleteideas.net/book/the-book-2nd.html.

[37] Andrew Szot et al. «Habitat 2.0: Training Home Assistants to Rearrange their Habitat». In: *Advances in Neural Information Processing Systems (NeurIPS)*. 2021.

[38] Vaibhav Unhelkar et al. «Human-Robot Co-Navigation using Anticipatory Indicators of Human Walking Motion». In: mag. 2015. DOI: 10.1109/ICRA.2015.7140067.

[39] Erik Wijmans et al. «Decentralized Distributed PPO: Solving PointGoal Navigation». In: *CoRR* abs/1911.00357 (2019). arXiv: 1911.00357. URL: http://arxiv.org/abs/1911.00357.

[40] Joel Ye et al. «Auxiliary Tasks and Exploration Enable ObjectNav». In: *CoRR* abs/2104.04112 (2021). arXiv: 2104.04112. URL: https://arxiv.org/abs/2104.04112.