Rewarding Actions Of Reinforcement Learning Agents On MapWorld

Jacob Löbkens
Universität Potsdam

Individual Project Module

May 11, 2022

Abstract

Reinforcement learning has become one the most interesting fields of research in recent years, with wide applications in disciplines like NLP, robotics and control engineering.

One of the problems for a researcher wanting to use reinforcement learning, is how to incentivize the agent to actually do the task it is designed to do. This is done by rewarding desired actions and punishing undesired ones. But with increasing complexity of the system that is to be solved, deciding what is to be rewarded and how much rewards should be handed out becomes more and more difficult. Therefore considerable effort should be spent on designing such a reward function.

The following work describes how multiple reward functions were designed and evaluated for MapWorld, an environment in which an agent has to find a specific room based on textual descriptions.

The project shows how the different components of the reward functions effect the performance of an agent, without finding a definitive best reward function. However, several ways forward to improve upon the work are suggested.

Contents

1 Introduction

Deep reinforcement learning has become very popular in research over the last years , with some remarkable achievements in various disciplines. These include beating human performance in board-games like Go (Silver et al. [2016]) and video games like Starcraft 2 (Vinyals et al. [2019]), controlling robots (Polydoros and Nalpantidis [2017]) and even controlling nuclear fusion reactors (Degrave et al. [2022]).

As impressive as these achievements are, they would not have happened if considerable effort was not spent on creating the environment with which the agent interacts. The success of any reinforcement learning algorithm is dependent on how the environment behaves.

For this project the environment is MapWorld[1]. It's a representation of a house, where a agent can move from room to room, looking for a specific one. MapWorld comes with a catch, it does not have a reward function. Reward functions are essential for agents to learn from the interactions with the environment (as stated in *Reward is enough* by Silver et al. [2021]).

Therefore the goal of the project is to find a reward function, for the environment MapWorld, that provides sufficient rewards along an episode such that an agent is able to learn how to successfully navigate to a specific room in the environment. To find such a function, several different variations of reward functions were designed and tested against each other. How the reward functions were designed and tested is explained in section 3.

As outlined in section 5 the project failed to find a reward function. However the foundations for further experiments were laid.

All code, results, plots and instructions on how to replicate them can be found on github[2]

2 Related Work

From the description of the environment and the goal of the project its apparent that several things are needed.

First the problem setting calls for using reinforcement learning (RL). RL is especially suited for dynamic environments where labeled data is not easily available.

[1] https://github.com/clp-research/sempix/blob/master/03_Tasks/MapWorld/intro_to_mapworld.ipynb

[2] https://github.com/JacobLoe/Rewarding_actions_of_RL-agents_on_MapWorld

2.1 Reinforcement Learning

Reinforcement Learning (RL) is a sub-field of machine learning the same way supervised and unsupervised learning are.

However instead of using a model to figure out the distribution of a static set of datapoints (with or without labels), in RL an agent chooses actions a depending on observations o in an environment. Agents choose actions based on a policy π. π is function of arbitrary form that maps states s to actions a, essentially saying which action to take given a state. The policy can be seen as a look-up table for actions. After the agent has chosen an action, the environment changes its state s depending on the action, which in turn changes the observations the agent receives. Every action taken also yields a reward r, which is dependent on a (generally unknown) reward function $R(a, s)$ of the environment. The reward is then used to update the policy such, that actions that lead to higher reward are more likely to be chosen again. After the update, the agent chooses another action and the whole cycle repeats. These repetitions of actions of the agent and reactions of the environment (change of state, returning reward) go on until some terminal state is reached. In reinforcement learning this whole sequence $(s_0, a_1, s_1, r_1, ...a_t, s_t, r_t)$ is called an episode.

An easy example would be a robot operating on an assembly line. The robot is the agent and the environment the part of the assembly line it works on. Through a camera the agent can see what is moving with the assembly line. The action space of the robot are its degrees of freedom, for example moving above the line on a x- and y-axis, as well as lowering or raising a manipulator. When an item is moved along the line the robot sees the item with the camera (an observation) and has to pick it up, move to the side of the line and drop the item in a box (choosing the correct sequence of actions). A reward might only be given for dropping the item in the correct box or every time a correct action is chosen.

Because in many problems the number of possible states and actions can become quite large, using a table for the policy would be infeasible. In deep reinforcement learning the policy is approximated with an deep neural network (hence the name deep). In the following sections the names reinforcement learning and deep reinforcement learning will be used interchangeably.

This is only a brief overview over reinforcement learning and only meant to support understanding the general ideas of this project. For an in-depth explanations of the mentioned concepts please refer to the reinforcement learning course by Silver [2015], *Reinforcement learning: An introduction*

by Sutton and Barto [2018] or Reinforcement Learning Made Simple, a blog series on towardsdatascience.com.

2.2 IQA: Visual Question Answering in Interactive Environments

The initial idea to focus on reinforcement learning for this project came from reading the paper *IQA* from Gordon et al. [2018].

They use RL in a task they call *Interactive Question Answering*. In this task an agent has to answer a question by interacting with an environment. The environments are 3d-scenes from the IQUAD v1 dataset[3], as seen in figure 1. The third column shows a view of the whole scene (the agent has no access to this information). The second column shows the initial view of the agent.

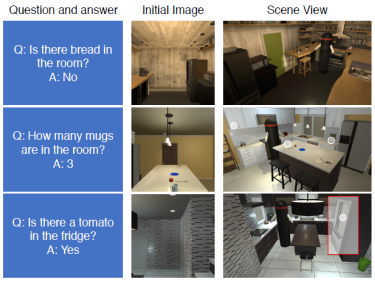


Figure 1: Example questions and scenes in IQUAD v1 [Gordon et al., 2018], page 1

The agent starts in a random spot in the scene and then has to act and plan such that it is able to answer the question corresponding to the room.

As the agent is not only required to understand how to interact with its environment, but also how objects in the scenes are related to each other (e.g. dishes are in cupboards, food is in a fridge), Gordon et al. [2018] developed the *Hierarchical Interactive Memory Network* (HIMN).

HIMN (shown in figure 2) is a hierarchical reinforcement learning algorithm[4], made of a high-level controller (named planner), low-level controllers each responsible for a different task and a memory. At each time-step the planner is responsible for choosing to activate a controller for a specific task (e.g. moving or manipulating something). That controller then chooses how to proceed (e.g. where to move or if information is to be written into

[3] https://github.com/danielgordon10/thor-iqa-cvpr-2018
[4] https://thegradient.pub/the-promise-of-hierarchical-reinforcement-learning/

the memory). The low-level controllers can be trained independent of each other.

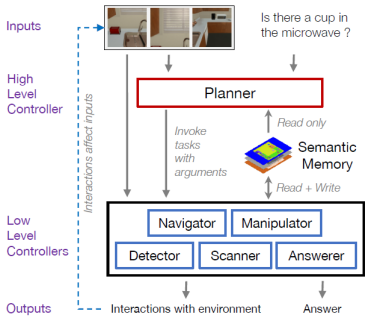


Figure 2: Hierarchical Interactive Memory Network example [Gordon et al., 2018], page 2

The scenes in IQUAD v1 are similar to MapWorld in the way that MapWorld can be seen as a less complex version of it. Every time the agent takes a step or turns in IQUAD v1 it essentially visits a new room (the differences between the rooms are very small). In MapWorld taking a step takes the agent to an entirely new room, however the rooms are still thematically connected.

The "questions" the MapWorld-agent has to answer are less complex as well. The agent has to only find where a room is on the map. This is roughly equivalent to the agent, in IQUAD v1, being asked to find where a certain object is in the scene.

During the project however it became apparent that using HRL was beyond the scope of the project. Especially as just setting up a RL-agent as a baseline took a lot of time and came with enough challenges already. As such it was decided to discard the HRL approach and just focus on the reward functions.

3 Methodology

The work of the project consists of three major parts:

1. implementing MapWorld as a RL-environment
2. designing different reward functions
3. designing, implementing and training of the different agents

The following sections describe each part, its challenges, the solutions to the challenges and its implementation.

3.1 MapWorld

As the environment for the project MapWorld[5] will be used. It is an environment for games which explore dialogue and vision. It can be described as a directed graph where the nodes are the rooms and the edges are the connections between two rooms.

To create a new map a grid is created, which is traversed at random. The nodes of the resulting graph are then assigned rooms. MapWorld has four parameters with which one can influence the created graph.

The first two are the dimensions of the grid the graph is created on. Third is the number of nodes in the graph. The last is a tuple of integers detailing how rooms are assigned to the nodes. It defines how often rooms of certain categories are to be assigned to nodes of the graph. With this room types like bedroom, attic or restroom are ensured to be found more than once. This is done to increase the plausibility of the created maps[6]. All parameters were left at their default values. 4 for the grid dimensions (a $4x4$ grid), 10 for the number of nodes and $(2, 2)$ for the room types.

In the context of reinforcement learning, MapWorld can be modeled as a Markov Decision Process $M = (S, A, P, R, \gamma)$[7], where S is the set of all states, A the set of all actions, P the transition probabilities between states given an action, R the reward function and γ the discount factor.

The states S are a tuple of each an image representing the current room, the caption for the target room and the available directions in the current room.

The action-space is discrete and the available actions A are the cardinal directions (north, south, west, east) and the signal that the agent is done with the game. Not all directions are available in all rooms. The available actions are given in natural language. Figure 3 shows an example of an instance of MapWorld. The squares are the rooms, the black lines the edges connecting the rooms. The black square is the target room the agent has to find. The red square is the current position of the agent. In its current position the agent has three (useful) actions available. It can move west, to

[5] https://github.com/clp-research/sempix/blob/master/03_Tasks/MapWorld/intro_to_mapworld.ipynb

[6] see An Environment: MapWorld, section ADE Maps

[7] Taken from Silver [2015], lecture 2, page 24

the south and end the episode. From its current position it would have to move to the south twice and then end the episode.

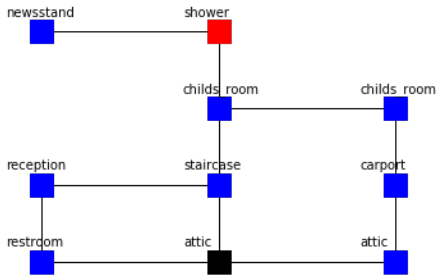


Figure 3: Example map for MapWorld, red is the agents position, black the target room

3.1.1 Gym environment

MapWorld in itself is not ready to be used as a RL-environment. The only way to interact with MapWorld is by moving around in it. There is no way for an agent to communicate with MapWorld that it wants to stop an episode. Also a reward function does not exist, which is an essential part of a RL-environment (and by extension this project).

Therefore a wrapper module was written to allow agents to interact with it. The structure of the wrapper is modeled after the environments in OpenAi Gym[8].

OpenAi Gym is a collection of environments created by Brockman et al. [2016] which all share the same standardized structure. Because of this one can theoretically easily test existing implementations of different RL-models in a few lines of code[9]. Gym environments are defined by three main functions which all algorithms expect to exist: init, reset and step. In the case of the MapWorld-wrapper their specific function will be explained in the following paragraphs.

The init-function loads a dictionary containing captions for all images into the memory, sets up the correct reward function and checks whether the parameters that are used are valid.

With the reset-function an episode is started. A new, randomized map object is created and the initial position of the agent and the target room are

[8] https://github.com/openai/gym
[9] for example Stable Baselines from Raffin et al. [2019]

chosen. The caption corresponding to the target room image is extracted and the image of the room at the current position is loaded into memory. The available directions from MapWorld are always in the form *'You can go: East, West'*. This string is extended by adding *'or select_room'*. Then the caption of the target room and the available directions are concatenated. The reset-function then returns this string and the image of the current room as the initial state of the environment.

When the step-function is called, the environment reacts in accordance to the action it was called with. It returns the current state of the environment and the reward for taking the action that led to the state. Additionally it returns a Boolean value indicating whether the episode has ended or not and an integer indicating if the target room has been found in this episode (1 when its found, 0 when not).

One catch with OpenAi Gym is that it does not support strings (so natural language) as action spaces. Without this support integrating *Stable Baselines* would be a considerable challenge. However, the Gym-like structure was still kept for two reasons. First it makes the code more readable and consistent to work with. Secondly OpenAi Gym may add support for natural language action spaces in future releases, in which case changing MapWorld to accommodate them should be an easy extension.

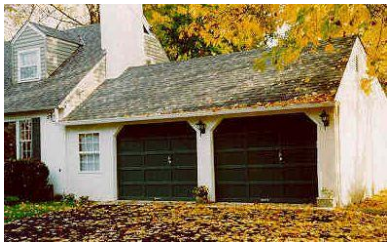
3.1.2 Images and captions

The maps in MapWorld are created with the help of the dataset ADE20K (Zhou et al. [2017], Zhou et al. [2018]). ADE20K contains annotations for about 25.000 images. The annotations range from object- and part-segmentations to object descriptions. The part of the dataset that is relevant to the project is that each image is associated with categories and subcategories. The categories describe broad concepts which are then further broken down into subcategories. For example *home_or_hotel* includes the subcategories *closet* and *shower*, the category *urban* includes *garage_outdoor* and *kiosk_outdoor*. Figure 4 shows two examples images. All images are of varying size and resolution.

From these categories only a subset were selected to make up reasonable configurations of houses. The used categories are hardcoded in a file[10] as *_target_cats*, *_distractor_cats* and *_outdoor_cats* (the exact same categories as in the original MapWorld code). The categories are used like described in MapWorld.

[10] https://github.com/JacobLoe/Rewarding_actions_of_RL-agents_on_MapWorld/blob/main/MapWorld/maps.py

(a)



(b)

Figure 4: Example images for categories *home_or_hotel* and *urban* of the ADE20K dataset. (a) shows a closet, (b) a garage

In addition to the images themselves, captions are needed for the rooms in case it is chosen as the target room, as the agent needs a description of the target room as the initial state. As ADE20K is lacking captions, *Localized Narratives* are used (Pont-Tuset et al. [2020]). It is a dataset containing various annotations for image-datasets such as *COCO*, *Flickr30k* and ADE20K. The most important part is that each image is associated a caption, describing the content of or scene in the image.

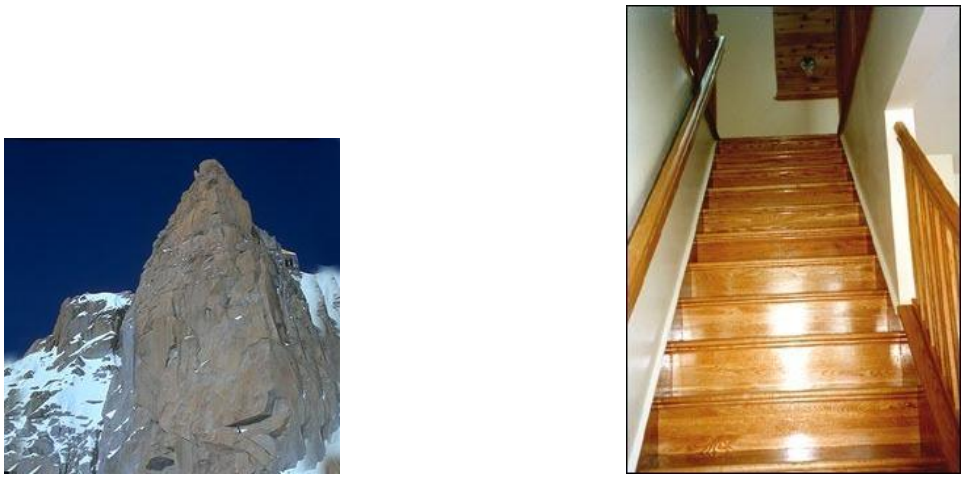
However *Localized Narratives* does not have captions for every image that is used by MapWorld. Therefore the caption-less images were removed from the corpus. This reduced the number of images from 9010 available for MapWorld to 7917. The captions for each image are saved in a .json-file with the file name of the image as keys and the corresponding caption as value.

	ADE20K	*MapWorld*
Minimum caption length	5	7
Median caption length	37	45
Average caption length	42	50
Maximum caption length	213	213
Number of all captions	20210	9010
Number of captions in used categories	–	7917
Vocabulary size	3209	1903

Table 1: Statistics for the ADE20K captions in *Localized Narratives*

Table 1 shows the differences in the captions of *Localized Narratives*. The left column describes the captions for ADE20K, the right column the subset of images of ADE20K that are used by MapWorld. The length of the captions vary a lot, but most of the captions are of lower length, as the median length describes. The same can also be seen in the histogram 6. Figure 5 shows the two shortest captions for both the ADE20K-captions and the MapWorld-captions.

As described in an earlier paragraph, MapWorld does not use images from all categories and even removes images from the used categories if no captions exist for them. The effect of this can be seen in the table. Only 7917, less than half of the 20210 images are used. Because of this the captions in MapWorld only use about 2/3 of the vocabulary of ADE20K.

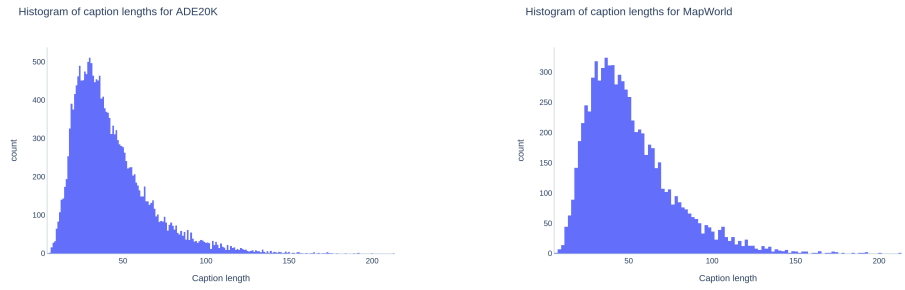


(a) (b)

Figure 5: Example images for the captions. (a) shows a mountain, the corresponding caption *This image is snow [sic] mountain.* is the shortest for ADE20K. (b) shows a staircase, the corresponding caption *In this image I can see stairs.* is the shortest in MapWorld.

Figure 6 compares the histograms of the caption lengths for the ADE20K-captions and the MapWorld-captions. As can be seen in the figure, although only half of the images in ADE20K are used, the distribution of the caption lengths has is very similar.

Alongside every image a feature-vector of the images of dimension 2048 is saved. These vectors are needed for computing the distance between two images, which is important for some reward functions. How the vectors are



(a) (b)

Figure 6: Histograms of the caption lengths for *Localized Narratives*. (a) shows the counts for all ADE20K-captions, (b) the counts for the captions used in MapWorld

used to compute a reward will be explained in section Reward function design. The vectors are created by feeding an image into an image-classification model and saving its output at the second-to-last fully-connected layer. The image-classification model used is the *InceptionV3-model* by Szegedy et al. [2016], with weights pre-trained on *ImageNet*.

3.2 Reward function design

The reward function is an integral part of any reinforcement learning environment, as the reward is used to to update the policy. As already mentioned in section 3.1, out of the box MapWorld has no reward function.

To create a reward function from scratch one has to first gain an understanding about what happens during an episode, as well as what the intended goals during an episode are.

Concerning the latter, for MapWorld the aim is to train an agent that can find the target room consistently or more specifically to maximise the accuracy of finding the correct room. One prior that is known about MapWorld, is that the layout of the environment and the location of the target room are always changing. This means an agent has to occasionally move around to consistently find the target room. Therefore any good reward function has to encourage an agent to explore the environment by visiting different rooms. The reward function can also not be to liberal with returning rewards, as that might lead to the agent learning to never end an episode and earn endless rewards.

Regarding the former, during any timestep in an episode only one of four outcomes is possible:

1. If the correct room is chosen, the episodes ends
2. If the wrong room is chosen, the episodes ends
3. If the agent chooses an action which is allowed, it moves in that direction and gets new observations
4. If the agent chooses an action which is not allowed, nothing happens and the agent gets the same observations again

In table 3.2 items 1 and 2 both represent the action *select_room*, items 3 and 4 represent the movement actions (the cardinal directions). In theory the movement actions can be chained one after another indefinitely, while *select_room* can only ever occur once during an episode. It might not even occur once, in the case the number of steps during an episode is restricted. Considering that the agent can always choose from all actions regardless of whether they are relevant to the current state, infinite episodes are not unlikely.

Note that the only outcome where the agent necessarily has to get a positive reward is when the correct room is chosen, as in the end finding as many correct rooms as possible is the goal. A naive first solution would be to assume that this case, giving only one reward (for selecting the correct room), should be enough. However, this would be equivalent to a game where the only instruction is to win. It leads to a very sparse reward function, as the agent is at maximum, rewarded only once per episode. Sparse rewards usually make learning an optimal policy very hard, if not impossible (Zai and Brown [2020] or Sutton and Barto [2018, Chapter 17.4])

It follows then, that items $2 - 4$ should be used to produce rewards too (positive or negative). In addition to a positive reward for selecting the correct room, the following components were implemented in MapWorld:

1. A constant, negative reward for selecting the wrong room
2. Reward according to the similarity between the selected room and the target room
3. A constant, negative (or positive) reward for taking an allowed step
4. A constant, negative (or positive) reward for taking a disallowed step
5. Linear increasing reward for correct steps

6. Increasing reward according to logistic function for correct steps

These components were combined freely to create *nine* reward functions. The different variations of the reward functions were designed to cover a variety of different scenarios.

Items 1 and 2 of table 3.2 define penalties (negative reward) or rewards for selecting a room. In this context, a constant penalty (or reward) means that it stays the same regardless of the state of the environment. The reward for item 2 is calculated by multiplying a constant reward with the distance between (the images of) the target room and the room selected by the agent. First the distance between two rooms is calculated by computing the euclidean distance (see equation 1) between two feature-vectors mentioned in section Images and captions.

$$d\left(p,q\right) = \sqrt{\sum_{i=1}^{n}\left(q_i - p_i\right)^2} \tag{1}$$

The distance is then normalized by subtracting the mean and dividing it by the maximum distance, as shown in equation 2.

$$x(p,q) = -\frac{d(p,q) - mean}{max} * c \tag{2}$$

To get the mean and the maximum distance, the distance between all feature-vectors was computed and the values then taken as constants for the reward function. This results in an distance distribution going from -0.5 to 0.5 with a mean of 0. As the distance between two identical objects is 0 multiplying the normalized distance with a constant would result in the minimal possible reward, which is the opposite of the expected effect. To invert this the normalized distance is multiplied by -1. c is a scalar representing the true range of the reward. For example 2000, to scale the function to a range of -1000 t0 1000.

As can be seen in figure 7, the distance follows a Gaussian distribution[11].

Item $3-6$ focus on penalizing or rewarding the agent for taking steps. Their goal is twofold. First to teach the agent to not move indefinitely and second to reinforce taking the correct steps. 3 and 4 aim to solve this with

[11] Figure 7 shows only the distribution of computing the distance for a random sample of 1000 feature-vectors against all other feature-vectors, resulting in 7.917.000 distances. The python-library used for creating the plot was not able to handle visualizing a plot with 7917^2 distances. The mean and maximum distance were computed from the full distribution.

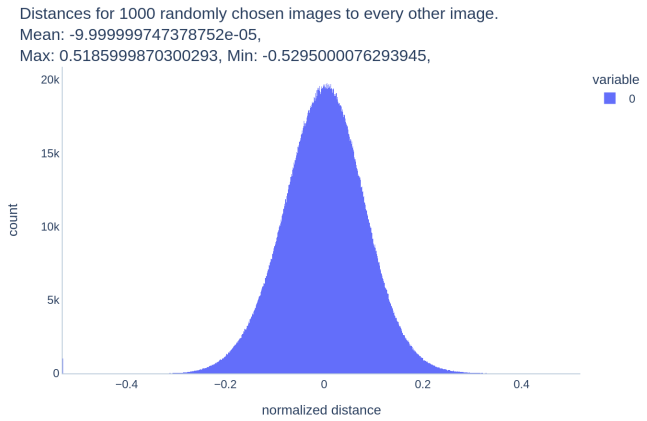


Figure 7: Distribution of the normalized distance for 1000 samples

constant penalties. For 5 the penalty rises linearly with the number of steps the agent has taken. In the function the slope was chosen to be -1, so after 10 steps the agent receives penalties the size of the constant penalty. This was done to not overly punish early exploration while also heavily punishing long episodes. 6 uses the logistic function (see equation 3) instead of a linear one. The reasoning is the same as for 5, but with further emphasis on a light penalty of the early steps.

$$\sigma(n_t) = \frac{L}{1 + e^{-k(x-x_0)}} \tag{3}$$

All components where only tested with a minimal amount of different hyperparameters (scaling factors, constants). An extensive hyperparameter-search would have lead to an considerable amount of time spent for training models. Not doing this was seen as an acceptable tradeoff, as the training time for one model was already huge (this will be further elaborated in section Model Training. The same reasoning is true for creating combinations of the reward function components (and therefore different reward functions).

3.3 Reward functions

The following lists the definitions of the reward functions, as designed with the conditions outlined in section 3.2 in mind.

In the reward function definitions the following substitutes are used:

- A_t are the actions available at each timestep.
- x is the normalized euclidean distance between the target and current room
- n_t is the number of steps taken by the agent
- $\sigma(n_t)$ is the logistic function $\sigma(n_t) = \frac{L}{1+e^{-k(x-x_0)}}$

$$r_1(a_t, s_t) = \begin{cases} 1000, & \text{if } a_t\text{=select correct room} \\ 0, & \text{otherwise} \end{cases} \tag{4}$$

$$r_2(a_t, s_t) = \begin{cases} 1000, & \text{if } a_t\text{=select correct room} \\ -1000, & \text{if } a_t\text{=select wrong room} \\ 0, & \text{otherwise} \end{cases} \tag{5}$$

$$r_3(a_t, s_t) = \begin{cases} 1000, & \text{if } a_t\text{=select correct room} \\ -1000, & \text{if } a_t\text{=select wrong room} \\ -10 * \sigma(n_t), & \text{if } a_t\text{=step and } a_t \text{ in } A_t \\ 0, & \text{otherwise} \end{cases} \tag{6}$$

$$r_4(a_t, s_t) = \begin{cases} 2000 * x, & \text{if } a_t\text{=select\_room} \\ -10, & \text{if } a_t\text{=step and } a_t \text{ in } A_t \\ 0, & \text{otherwise} \end{cases} \tag{7}$$

$$r_5(a_t, s_t) = \begin{cases} 2000 * x, & \text{if } a_t\text{=select\_room} \\ 10, & \text{if } a_t\text{=step and } a_t \text{ in } A_t \\ -10, & \text{otherwise} \end{cases} \tag{8}$$

$$r_6(a_t, s_t) = \begin{cases} 2000 * x, & \text{if } a_t\text{= select\_room} \\ -1 * n_t, & \text{if } a_t\text{=step and } a_t \text{ in } A_t \\ 0, & \text{otherwise} \end{cases} \tag{9}$$

$$r_7(a_t, s_t) = \begin{cases} 2000 * x, & \text{if } a_t\text{=select\_room} \\ -10 * \sigma(n_t), & \text{if } a_t\text{=step and } a_t \text{ in } A_t \\ 0, & \text{otherwise} \end{cases} \tag{10}$$

$$r_8(a_t, s_t) = \begin{cases} 200 * x, & \text{if } a_t\text{=select\_room} \\ -10 * \sigma(n_t), & \text{if } a_t\text{=step and } a_t \text{ in } A_t \\ 0, & \text{otherwise} \end{cases} \tag{11}$$

$$r_9(a_t, s_t) = \begin{cases} 20 * x, & \text{if } a_t\text{=select\_room} \\ -10 * \sigma(n_t), & \text{if } a_t\text{=step and } a_t \text{ in } A_t \\ 0, & \text{otherwise} \end{cases} \tag{12}$$

3.4 Agents

With the question of *how* the agents are rewarded out of the way, its necessary to describe the architecture of the two different agents that will attempt to solve MapWorld. Two different type of agents were chosen to limit the complexity of the project. The two agents chosen are a random agent and a reinforcement learning agent, namely an actor-critic algorithm.

3.4.1 Random agent

The first model chooses actions at random, regardless of the state of the environment. It is used as a baseline-model to have a definitive performance floor for the models and also as a sanity check, to see if the environment behaves as expected.

The probability for the agent to choose any action is 0.2 (assuming five possible actions). Because this probability and the number of rooms of each map are fixed, several assumptions can be made about the performance of the agent.

First, the reward the random agent accumulates during an episode should fluctuate highly and on average, not increase. Second, for the accuracy it only matters whether the agent and the target room are in the same room. The target room has a probability of 0.1 to be a specific room out of the ten. Likewise, the random agent is placed in a random room at the start. Now the agent takes a random number of actions with each action being equally likely. The only action of those that matters however is the last one, as that is the one where the agent decides to choose a room. For a truly random agent this would mean finding a room is the same process as choosing a random number out of ten, just with a very long decision process. Therefore the accuracy of the random agent is expected to also approach, on average, 0.1.

In case of constant rewards being returned for selecting a room, a histogram of the rewards should look exactly like a histogram of the room selections (counting correct and incorrect selections). When the reward is calculated with the euclidean distance, the histogram should follow the Gaussian distribution from figure 7.

Accordingly, a histogram of the correct and the wrong guesses should show the wrong guesses be 10 times the number of the wrong guesses. A histogram for the steps taken each episodes will likely follow a half-normal distribution[12], meaning the higher the number of steps the lower their counts.

3.4.2 Reinforcement Learning Agent

The reinforcement learning model is an actor-critic model. Actor-critic models are a type of algorithms which aim to combine the advantages of value- and policy-based RL-algorithms (Konda and Tsitsiklis [2000]). The core idea is to use two models in tandem. The actor is learning the policy and outputs the best action given a state. The critic is calculating the value of taking the action given the state.

The architectures of the value- and the action-networks to process text and images together are inspired by the work of Fu et al. [2019]. The implementation of the actor-critic algorithm is a combination of tutorials from PyTorch[13] and TowardsDataScience[14].

Figure 8 shows the architecture of the deep learning network and the pre-processing steps. The grey bounding box encloses the actual parts of the model. The grey boxes at the top represent the preprocessing of the environment states to create the image and sentence embeddings. The two blue boxes at the bottom are the value of a state and the action to be taken, respectively.

Because the hardware used for training restricts how big any model and input tensors can be, the decision was made to move out the parts of the model that produce embeddings from the text and images. With this, pre-trained models, optimized for a specific task, can be used as a preprocessing step. In addition to this the training and inference of the remaining model will be faster. The weights of the pre-trained models remain static and are not trained when the RL-model trains.

To process the images of the rooms a pre-trained CNN-model is used,

[12] wikipedia.org/wiki/Half-normal_distribution
[13] github.com/pytorch/examples/blob/master/reinforcement_learning/actor_critic.py
[14] towardsdatascience.com/learning-reinforcement-learning-reinforce-with-pytorch

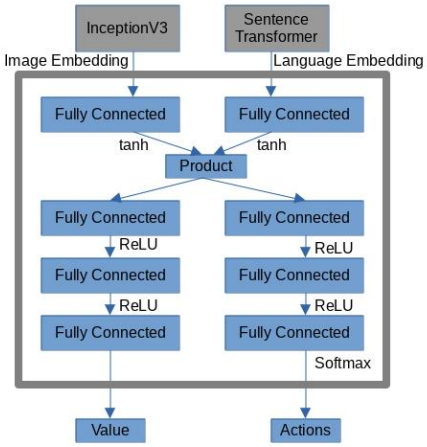


Figure 8: Architecture of the actor-critic model with pre-processing

the Inception-V3[15] by Szegedy et al. [2016]. Before feeding an image into the model, it is transformed to dimensions $299 * 299$ and normalized with a function from PyTorch[16]. The Inception-model outputs a vector of size 2048.

To process the text a pre-trained transformer-model is used. The model is a transformer model called sentence transformer[17] developed by Reimers and Gurevych [2019]. Sentence transformer is based on BERT Devlin et al. [2019]. A pre-trained BERT-model is fine-tuned on natural language inference (NLI) datasets. In NLI a model has to essentially compare whether two sentences are related to each other. The model produces an embedding vector of length 384.

The embeddings created by both these models are used as the input tensors for the RL-model.

The actual RL-model itself consists of only fully connected layers, as can be seen in the area inside the grey box in figure 8. The image- and sentence-embeddings are each fed into a fully-connected layer, to which a *tanH*-function is applied. Both these layers have an output dimension of 200. The outputs of those fully-connected layers are multiplied with each other. The vector resulting from the multiplication now represents the state

[15]The model is the same model that is used by the reward functions for rewarding the selection of a room.

[16]https://pytorch.org/hub/pytorch_vision_inception_v3/

[17]The specific version of the model that is used is called *paraphrase-MiniLM-L6-v2*, taken from the SBert website.

of the environment and is used to both decide on the appropriate action and the value of the current state. In the following these parts of the model will be called action-model and value-model respectively.

Both the action-model and the value-model use two consecutive fully-connected layers with ReLU activation functions. For both layers the output dimension is 100.

As the last layer the action-model uses a softmax-function, to compute a probability-distribution over which of the 5 actions to take in the current state of the environment. The last layer of the value-model is a fully-connected layer with one output, representing how good it is to be in the current state.

The above mentioned model and its parameters were only optimized to keep inference and training reasonable. Another goal was to fit the model into the memory of the GPU while maximising the batch size for the training This was done to make the training of several models possible in a reasonable timeframe.

In addition to the actor-critic agent an agent based on the REINFORCE-algorithm (Williams [1992]) was created. But during the initial tests the agent showed a similar (or worse) behaviour than the actor-critic one. Therefore the decision was made to not include the REINFORCE-agent in the final evaluations, as that would have added an large amount of additional model training to the already long training.

Another thing that was tried was using BERT word-embeddings instead of sentence-embeddings. This was discarded as the resulting embeddings were of too high dimensions to be loaded into the GPU. Especially for batch sizes > 10.

Finally, the initial idea was to create the Actor-Critic agent only as a second baseline first and then focus on implementing a hierarchical reinforcement learning agent as described in Gordon et al. [2018]. This was discarded early as, during the development of the RL-agent, the hardware already limited the size of the one deep learning model used substantially. With HRL at least two models of similar size would have been used, which was not feasible under the constraints already mentioned.

The HRL-model (and the RL-model too) was also to be extended with a module capable of engaging in dialogue with an "all-knowing" director (For example building on the work of Das et al. [2017a], Das et al. [2017b] or Modhe et al. [2018]. Again this idea was discarded when the computational constraints of the GPUs became apparent, as well as the overall complexity of the task.

3.4.3 Action masking

As an additional baseline a subset of the reward functions will be evaluated in a slightly different setting, namely one where some actions are masked. Masking actions means that at any timestep the agent can only choose between actions that it is allowed to see, the rest are "hidden" from it. This consequently ensures that the agent will see a new state at every timestep.

In MapWorld this is implementing by following a blog-post (boring-guy.sh/posts/masking-rl/). First a list of Boolean values is created from the list of available actions (True when the action is available, false if not). With this mask, the masked actions in the output distribution of the action-network are set to $-\infty$. Applying a softmax-function to this new tensor, will result in a new probability distribution over the actions, with zero of the probability mass applied to the masked actions. The distribution is 0 where the actions were masked and the probabilities still sum up to 1.

Action masking is done to see if a theoretical agent with perfect natural language understanding[18] can perform better on MapWorld then the random baseline, using the provided reward functions.

If the agent would not perform better, it might suggest one of several statements (or all of them) are true:

- MapWorld is not solvable
- The reward function is not sufficient for the task
- The chosen RL-algorithm is wrong
- The DL-architecture/preprocessing is wrong

In the case there is a performance increase (relative to no action-masking) one can assume that the general direction of the approach will to be a correct one, even if any of the above statements a true. There is one caveat one would have to keep in mind though, which is just because a reward function returns desirable results with action-masking does not necessarily mean that the same reward function works when action-masking is turned off. Without the masking the agent has to essentially learn a mask (or something similar) from scratch, which may take a considerable amount of episodes more.

[18] In the context of MapWorld perfect would mean that the agent can infer the available directions from the observations just like a human would be able to

3.4.4 Model Training

To train the agents two GTX 1060 with 10 GB VRAM each are used (CUDA version 11.6). The version of the python-libraries used are fixed in the Anaconda environment file *environment.yml* in the GitHub repository.

All agents complete two million episodes. This number was chosen from an experiment with the reward function r_1 (constant reward for correct room selection, no other rewards or penalties and action masking is enabled), as that is the simplest form of all reward functions. The corresponding model was trained for an increasing number of episodes until the accuracy changed noticeably.

Figure 9 shows an distinct increase in accuracy of about 2% at about $1.2M$ episodes. This is assumed to be the minimum training time needed until a model achieves performance increases.

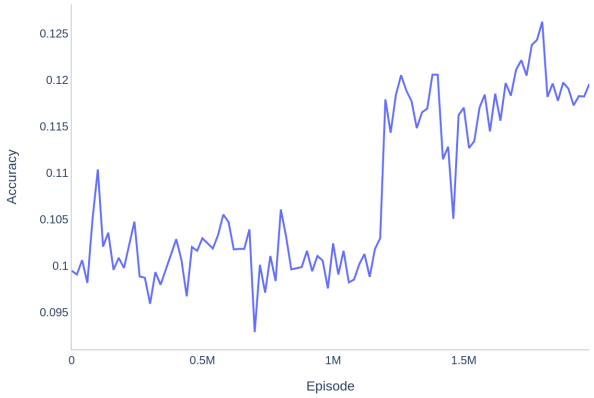


Figure 9: Accuracy of r_1 with action masking over 2m episodes

Following this it seemed reasonable to let other models train for at least double the episodes to see further increases in accuracy. But considering the amount of time the training of one model took coupled with the number of different reward functions to be evaluated, the number of episodes was kept at the aforementioned two million.

To train the actor-critic agents the chosen actions and the rewards gained are saved into batches after the end of every episode. During an episode the agents are restricted to taking 60 steps. This is done to prevent an explosion of the size of a batch in case an agent gets stuck choosing the same actions

over and over again. After 128 episodes the saved batch is used to update the weights of the action- and value-model.

All other hyperparameters (δ, learning rate, etc.) were not optimized and left the same as in the example code. Using all the mentioned parameters training of one actor-critic model for $2M$ episodes took about 38 hours. Running a random agent took about 17 hours.

4 Evaluation

As already mentioned in section 3.4 the reward functions will only be tested on a random agent and a reinforcement learning agent. For all agents trained with the reward functions three characteristic values will be saved after the training is done:

- The reward for every episode
- The number of steps taken in every episode
- Whether the agent found the correct room or not for every episode

From these values all metrics will be computed and/or plotted that are used to evaluate the impact of the reward functions on the training of the agents.

- Averaged accuracy per episode
- Reward per episode
- Steps per episode
- Histograms over rewards, steps and room guesses

The plotting of the accuracy is arguably the most important metric, as any increases directly translate to a better performance in the task. Plotting the reward and steps for every episode gives information about how the agents behave during an episode (are episodes finished immediately, is the agent exploring). The histograms serve as sanity checks, to see whether the reward functions work as intended.

For brevity, any plots of the rewards steps not shown or discussed here have been moved to the appendix. The remaining plots and histograms can be found in the folder results in the repository[19].

[19] JacobLoe/Rewarding_actions_of_RL-agents_on_MapWorld/tree/main/results

4.1 Random agent

First the performance effect of the reward functions on the random agent will be evaluated. Figure 10 shows the accuracy of the agent every 20000 episodes for $2M$ episodes. This means that 100 equal sized chunks where taken from the list of whether the agent found the room or not. The list is just a sequence of 0 and 1, which where summed up and divided by the size of the chunk, getting the accuracy of the agent for 20000 episodes. In essence, the plot shows the change of the accuracy over the course of the training of the agent.

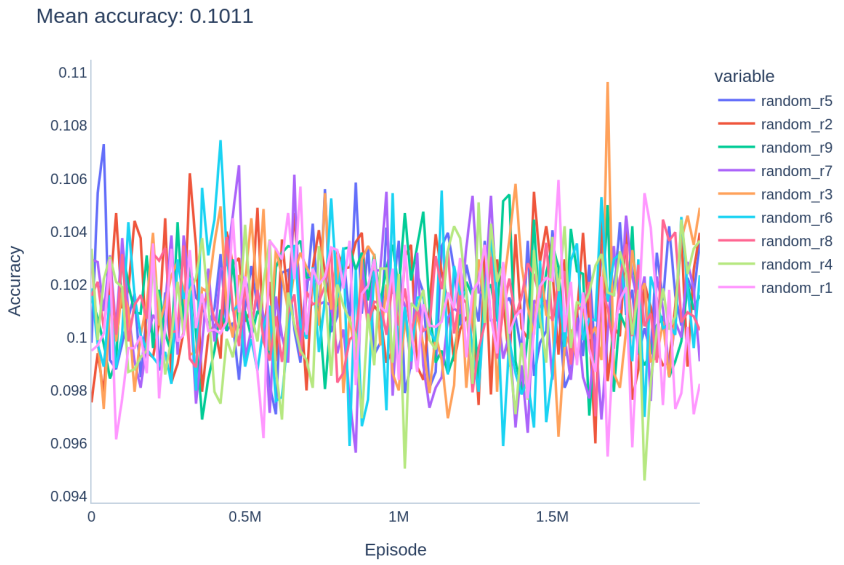
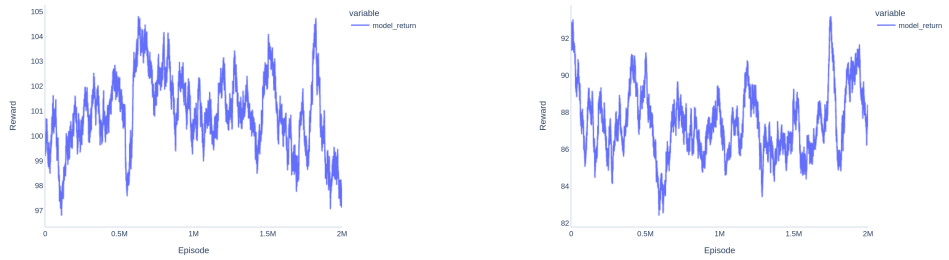


Figure 10: Accuracy of a random agent for reward functions 1-9 over $2M$ episodes. The top left number is the mean accuracy of all agents.

The plot confirms the assumptions made about the performance of the random agent in section 3.4.1. Regardless of the reward function chosen the accuracy stays largely between 0.098 and 0.104. Anytime a spike in accuracy occurs, for example right at the start for $r_5(blue)$ or at $1.7M$ episodes for r_3, it is only temporarily. In these cases the accuracy goes back to the average in just a few episodes. The mean accuracy in the top left of the plot is the average of the accuracies of all random agents. With 0.1011 the accuracy is like the predicted accuracy of 10%. The plots showing the other metrics seem to reinforce these observations.



(a) (b)

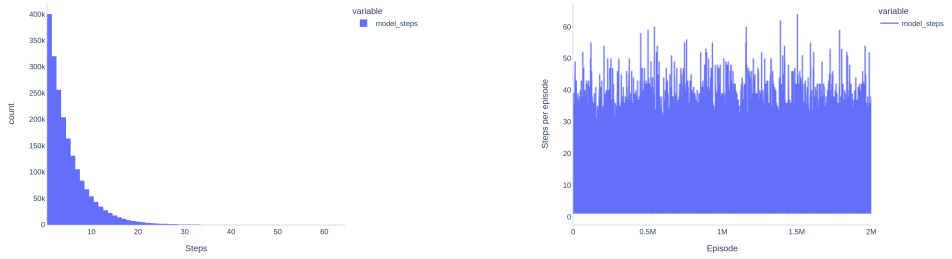
Figure 11: (a) Reward for a random agent with reward function r_1 per episode over $2M$ episodes (b) Reward for a random agent with reward function r_7 per episode over $2M$ episodes

Very similar to the accuracy, the rewards accumulated over the episodes are highly random and do not show any sign of increasing and then stabilizing. Figure 11 shows this trend for two reward functions. Even though the reward functions are very different from each other, r_1 rewards only correct guesses with a scalar, while r_7 rewards guesses based on distance and penalizes taking steps with a logistic function, the plots are essentially identical. The only notable difference is the difference in scale of the reward, which can be explained the differences in how the often agents are rewarded (or penalized) by the reward functions.

One important thing to note is that for better visualization a filter was applied to the plots of the accuracy. The filter is a moving average with a window size of 50000. A moving average is a type of low-pass filter which removes high frequencies from data. For example, without the filter the plot for r_1 would look like one solid block, as the data changes very frequently between 0 and 1000. The filter does preserve trends, unless they happen to occur in timeframes significantly smaller than the window size of the filter.

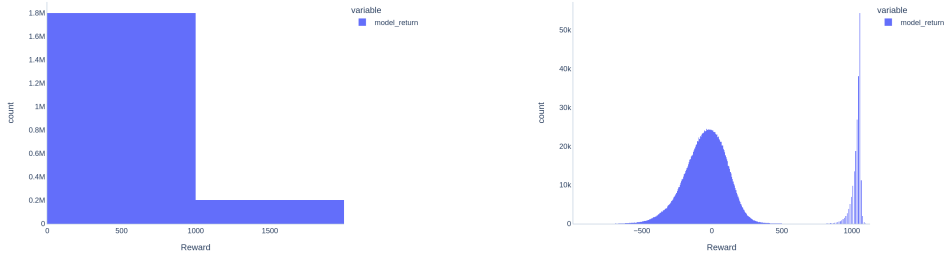
Figure 12 shows a histogram of the number of steps taken during an episode, as well as a plot of the steps taken in each episode. Again, the plots affirm the expectations stated in section 3.4.1.

Figure 13 shows the histograms for the reward per episode for the reward functions r_1 and r_5. Both show the expected behaviour. For r_1, the histogram is essentially a histogram of how many times the agent found the correct room. For r_5, (where the the room selection reward is dependent on distance), the histogram shows the Gaussian distribution shown in figure 7.



(a) 9 (b)

Figure 12: (a) Histogram of steps taken per episode of a random agent with reward function r_1 over $2M$ episodes (b) Steps taken per episode of a random agent with reward function r_1 over $2M$ episodes



(a) 9 (b)

Figure 13: (a) Histogram of the reward per episode of a random agent with reward function r_1 over $2M$ episodes (b) Histogram of the reward per episode of a random agent with reward function r_5 over $2M$ episodes

The big spike around 1000 can be explained with the way the random agent works. As already explained in section 3.4.1, the agent has probability of 0.1 to guess the correct room. This means in 10% of the episodes the agent gets the full reward of 1000 (or something close). The remaining 90% of the time the reward is a random sample from the distance distribution.

All other random agents show similar behaviour, with the only changes being differences in scale due to different values in the parameters of the reward function components. Overall the random agents behaved as predicted and their performance can serve as the baseline for the actor-critic models.

4.2 Actor-Critic Agents

For the evaluation of the actor-critic agents five reward functions (r_1, r_4, r_5, r_7 and r_9) were run twice. Once with action masking enabled, once without.

In figure 14 one can see a very similar behaviour of the actor-critic agent to the random agent. The accuracy has a slightly bigger fluctuation in the amplitude of about 0.095 to 0.105, but the overall trend stays the same. The same is true for the frequency of the occurrences of accuracy spikes. These spikes are of a higher magnitude than the ones of the random agents. The one exception is the masked variant of reward function r_1. It shows a significant increase in accuracy at $1.2M$ episodes and stays at around 0.117 for the remainder of the experiment.

Two reward functions in figure 14 are missing. In fact these two reward functions (r_2 and r_3) could not be evaluated with the actor-critic agent. Whenever an experiment was started the training times were projected to run for about 2000 hours, as opposed to the 38 hours of training-time of the actor-critic agent. Running these experiments for several days, with no change in the projected runtime showed that it was not a visual bug. All parameters, like the correct number of episodes or correctly enable training on the GPU were extensively checked for errors. The .json-files containing the parameters for the experiment can be found in */parameters* in the repository. The exact files were used with the random agent and ran as expected. The only difference to the other reward functions is that both r_2 and r_3 use a constant penalty of -1000 for selecting the wrong room. Other reward functions are also able to return negative numbers as reward for the room, but without having the same issue. From the code perspective the only difference is one *if-condition* which only checks one Boolean. Tests trying to figure out the cause of the problem, could not figure out its roots and fix it. The decision was then made to exclude r_2 and r_3 from the tests.

Figure 15 shows the reward function r_1 with action masking in more

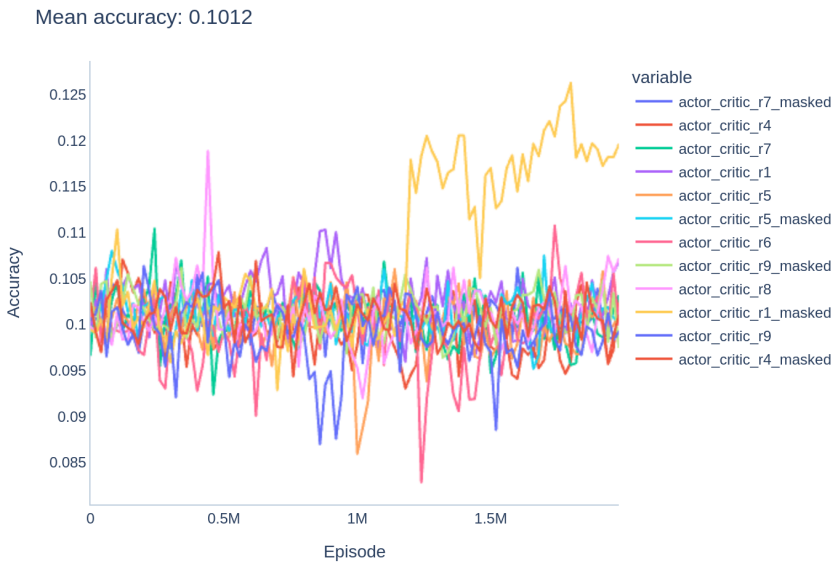


Figure 14: Accuracy of an actor-critic agent for reward functions (normal and masked) over $2M$ episodes. The top left number is the mean accuracy of all agents.

detail. After seeing the accuracy of r_1 in comparison to the other reward functions, r_1 was run four additional times, each time with the same parameters. This was done to see whether the performance increase was just by chance or could be replicated.

For each episode the average of the five experiments was taken, resulting in an average accuracy of r_1 per episode. The plot was then created using the same moving average as the other accuracy-plots.

The plot shows a clear upward trend. After an initial increase from 0.102 to 0.11 the accuracy stays around that level and only decreases for a short number of episodes, but never as low as 0.102 again. For the individual experiments, one plot shows the increase only for the first $5M$ episodes, before falling back to the performance of the random agent. A second agent has increased accuracy for $1.5M$ episodes before falling back to random performance. The two remaining agents achieve the increased accuracy in less episodes than the initial experiment(see figure 9 for reference). Although in both cases there is brief decrease (about $2M$ episodes) of accuracy to 0.1, the agents regain their previous performance[20].

Figure 16 compares the rewards accumulated during an episode of the best performing reward function r_1 to r_9 (as a stand-in for the remaining reward functions). As expected, the reward follows the trajectory of the accuracy. After $1.2M$ episodes r_1 (with action masking) receives more rewards, while all other agents perform the same as the random agent.

In figure 17 one can see an interesting detail. The number of steps taken per episode rise noticeably for r_1 at the same episode the accuracy of the agent rises. The expectation would be that the number of steps negatively correlates with the accuracy. With the reasoning being that with increasing "understanding" of the house the agent needs to visit less rooms to select the correct room. The relationship may also be that number of steps taken influences the accuracy, not the other way round. In this case the agent learned at the $1.2M$ episode mark that taking steps is not penalized. Which in turn leads to the agent taking more steps and increasing its chances of finding the correct room. In contrast the plots for all other reward functions show again the same behaviour as the random agents.

[20]The four mentioned plots for r_1 are in the appendix (figure 37 and following). They were left out of the main text to prevent visual clutter.

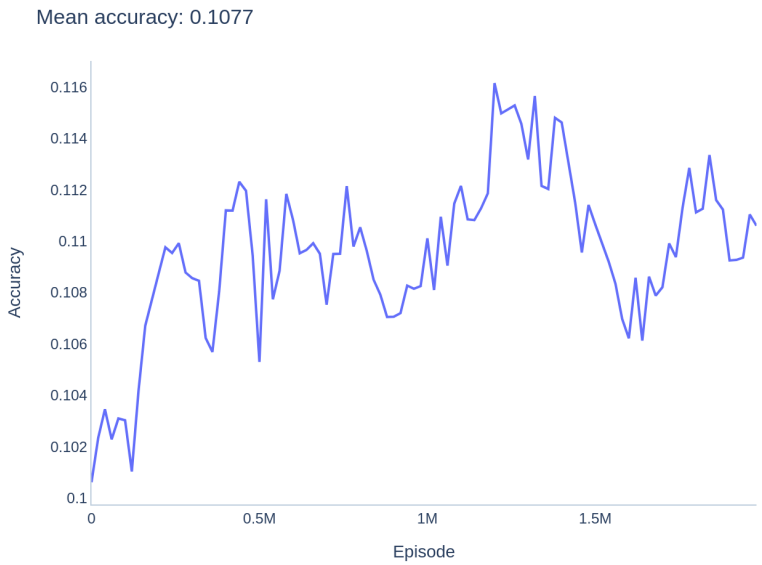
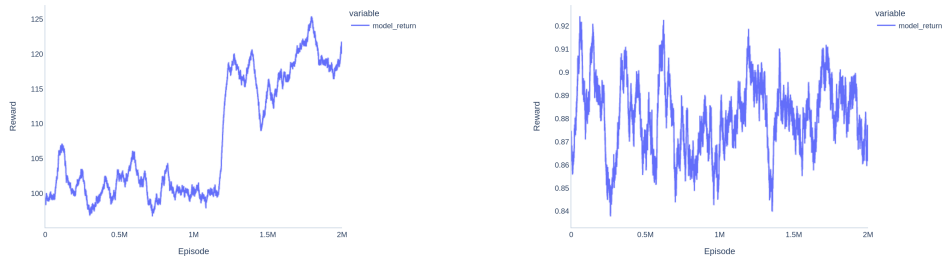
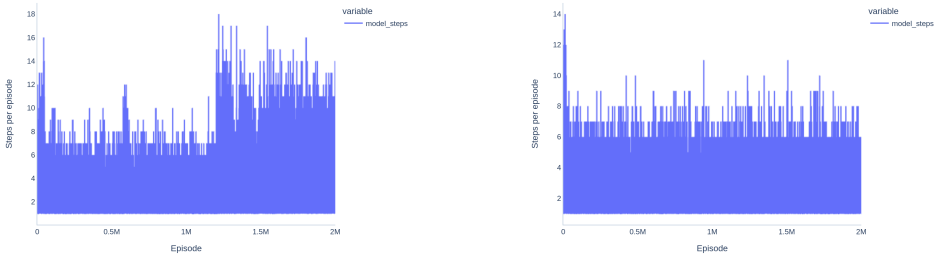


Figure 15: Aggregated accuracy of five actor-critic agents with reward function r_1 and action masking over $2M$ episodes. The top left number is the mean accuracy of all runs.



(a) (b)

Figure 16: (a) reward over $2M$ episodes of an actor-critic agent with reward function r_1 and action masking (b) reward over $2M$ episodes of an actor-critic agent with reward function r_9 and action masking



(a) 9 (b)

Figure 17: (a) steps over $2M$ episodes of an actor-critic agent with reward function r_1 and action masking (b) steps over $2M$ episodes of an actor-critic agent with reward function r_9 and action masking

5 Conclusions

The project managed to show the effects various reward functions have on agents. When used by an reinforcement learning agent, of the nine functions tested, only one yielded better results than randomness. This best performing reward function r_1 was the simplest one and worked only in case the agent was supported by masking the invalid actions. From this it could be inferred that action masking helps the agent. However, although the difference in accuracy is noticeable, it is also a small change (from 10% to 11%). Additionally, for the other reward functions, masking action seemed to have no effect on the performance of the agents.

Now these observations do not necessarily mean that the reward functions themselves do not work. It may be true that several of the problems mentioned in 3.4.3 are present. As *"Mapworld is not solvable"* can only be stated with relative certainty after extensive testing, the remaining three statements need to be looked at first.

The following two sections contain suggestions and ideas on how to improve upon the work described in the project, potentially alleviate some of the issues that are mentioned and get closer to finding a suitable reward function for MapWorld.

5.1 Faster Training

The success of the training is heavily dependent on the available compute. Because of the constraints several concessions in the architecture had to

be made, which should have an impact on the performance of the models. Possibly the biggest issue was that training one model takes an enormous amount of time. If one model takes more than a day to show any signs of an improvement, it severely limits the number of models that can be run and evaluated in a reasonable timeframe.

Following this a first idea for improvements would be to spend time on optimizing MapWorld to run faster and spend time on optimizing the pytorch code to run faster. This is especially important if one considers adding new components to the agents and therefore to its complexity. Not only might this raise the time it takes for one episode to complete, it might also increase the number of episodes needed for the agents to get to an increased performance. Considering one episode currently takes about 0.068 seconds[21], it is possible that there is not much time that can be saved.

Another promising avenue to check out might be to run multiple episodes in parallel. This is obviously only possibly if the given hardware permits this, as one would have multiples instances not only of the environment, but also the tensors representing the state. In return it should yield the greatest improvements with respect to the training time (4 environments run in parallel would effectively mean $8M$ episodes in 38 hours.) For example, Mnih et al. [2016] use parallel environments with the *Asynchronous Advantage Actor-Critic algorithm*(A3C).

5.2 Better Reward Functions

Regardless of whether it is possible to speed up the training further, time should be spent investigating the reward functions.

First would be to revisit the values for the rewards. These were all chosen in relation to the size of the reward for selecting a room in r_1 (1000), here the reward for finishing a game has to be several orders of magnitude bigger than the penalty for taking a step, in order to encourage exploration. This method is easy to use, but has the downside that it is not really based on any heuristics, just guessing.

Creating new components for the reward functions should also be considered. Adjusting the distance distribution of 7 to shift more towards a penalizing or rewarding distribution should be looked into. Its also possible to change the Gaussian distribution to a new distribution which suits the task more. The same is true for the functions which govern how steps are rewarded. Another interesting thing one might test is using two reward

[21] Training an actor-critic agent for $2M$ episodes takes about 38 hours. Therefore: $\frac{38hours}{2M} = 0.068$ seconds per episode)

functions. Where one reward function is used for the first half of the training and then switched out for a second, potentially more strict reward function.

One has to keep in mind that any new component added comes with its own set hyperparameters (like the slope or intercept). Finding the optimal set of hyperparameters of such a huge search space can be quite a challenge. Especially if it's done by hand. Random search[22] is easy to implement but might not find the global minimum. Grid search[23] is more likely to find a global minimum, but at the cost of being way more expensive computationally.

One idea to find an optimal reward function would be to make use of black-box optimization algorithms. Black-box optimization algorithms are a class of algorithms used when a function is expensive to evaluate, no closed-form is available and/or no gradients can be computed. If one considers the whole training of the agents a function, all three of the conditions are true. Black-box optimization algorithms are very popular for optimizing the hyperparameters of machine learning models, with various libraries being in use [24]. For example the objective to optimise could be the accuracy, the total reward over all episodes or an aggregation of both. Instead of optimising the whole reward function one could also only focus on the hyperparameters of the components. For example for a reward function with distance based room selection reward and a logistic step reward adjust only the constants of these functions.

Theoretically one could also optimise the hyperparameters of the RL-algorithms and the deep learning model, with the same process. Either instead of optimizing the reward function or alongside it. The former might not be as useful when the reward function changes and the latter would very likely make the optimization considerably more expensive.

For the RL/DL-hyperparameters it should be more efficient to look into other literature for improvements. Especially in the realm of deep learning models designed to handle images and text simultaneously. For example the work of Das et al. [2017b] who aimed to solve visual question answering challenges by rephrasing them as a multi-agent reinforcement learning problem, where the agents communicate via natural language.

[22] en.wikipedia.org/wiki/Hyperparameter_optimization#Random_search
[23] en.wikipedia.org/wiki/Hyperparameter_optimization#Grid_search
[24] for example: Spearmint, SMAC CMA-ES, Hyperopt

References

Greg Brockman, Vicki Cheung, Ludwig Pettersson, Jonas Schneider, John Schulman, Jie Tang, and Wojciech Zaremba. Openai gym, 2016.

Abhishek Das, Satwik Kottur, Khushi Gupta, Avi Singh, Deshraj Yadav, José MF Moura, Devi Parikh, and Dhruv Batra. Visual dialog. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 326–335, 2017a.

Abhishek Das, Satwik Kottur, José M. F. Moura, Stefan Lee, and Dhruv Batra. Learning cooperative visual dialog agents with deep reinforcement learning. *2017 IEEE International Conference on Computer Vision (ICCV)*, pages 2970–2979, 2017b.

Jonas Degrave, Federico Felici, Jonas Buchli, Michael Neunert, Brendan Tracey, Francesco Carpanese, Timo Ewalds, Roland Hafner, Abbas Abdolmaleki, Diego de Las Casas, et al. Magnetic control of tokamak plasmas through deep reinforcement learning. *Nature*, 602(7897):414–419, 2022.

J. Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. Bert: Pre-training of deep bidirectional transformers for language understanding. In *NAACL-HLT*, 2019.

Justin Fu, Anoop Korattikara Balan, Sergey Levine, and Sergio Guadarrama. From language to goals: Inverse reinforcement learning for vision-based instruction following. *ArXiv*, abs/1902.07742, 2019.

Daniel Gordon, Aniruddha Kembhavi, M. Rastegari, Joseph Redmon, D. Fox, and Ali Farhadi. Iqa: Visual question answering in interactive environments. *2018 IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 4089–4098, 2018.

Vijay R Konda and John N Tsitsiklis. Actor-critic algorithms. In *Advances in neural information processing systems*, pages 1008–1014, 2000.

Volodymyr Mnih, Adria Puigdomenech Badia, Mehdi Mirza, Alex Graves, Timothy Lillicrap, Tim Harley, David Silver, and Koray Kavukcuoglu. Asynchronous methods for deep reinforcement learning. In Maria Florina Balcan and Kilian Q. Weinberger, editors, *Proceedings of The 33rd International Conference on Machine Learning*, volume 48 of *Proceedings of Machine Learning Research*, pages 1928–1937, New York, New York,

USA, 20–22 Jun 2016. PMLR. URL http://proceedings.mlr.press/v48/mniha16.html.

Nirbhay Modhe, Viraj Prabhu, Michael Cogswell, Satwik Kottur, Abhishek Das, Stefan Lee, Devi Parikh, and Dhruv Batra. Visdial-rl-pytorch. https://github.com/batra-mlp-lab/visdial-rl.git, 2018.

Athanasios Polydoros and Lazaros Nalpantidis. Survey of model-based reinforcement learning: Applications on robotics. *Journal of Intelligent & Robotic Systems*, 86:153–, 05 2017. doi: 10.1007/s10846-017-0468-y.

Jordi Pont-Tuset, Jasper Uijlings, Soravit Changpinyo, Radu Soricut, and Vittorio Ferrari. Connecting vision and language with localized narratives. In *ECCV*, 2020.

Antonin Raffin, Ashley Hill, Maximilian Ernestus, Adam Gleave, Anssi Kanervisto, and Noah Dormann. Stable baselines3. https://github.com/DLR-RM/stable-baselines3, 2019.

Nils Reimers and Iryna Gurevych. Sentence-bert: Sentence embeddings using siamese bert-networks. In *Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing*. Association for Computational Linguistics, 11 2019. URL http://arxiv.org/abs/1908.10084.

David Silver. Lectures on reinforcement learning. URL: https://www.davidsilver.uk/teaching/, 2015.

David Silver, Aja Huang, Chris J Maddison, Arthur Guez, Laurent Sifre, George Van Den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Veda Panneershelvam, Marc Lanctot, et al. Mastering the game of go with deep neural networks and tree search. *nature*, 529(7587):484–489, 2016.

David Silver, Satinder Singh, Doina Precup, and Richard S. Sutton. Reward is enough. *Artificial Intelligence*, 299:103535, 2021. ISSN 0004-3702. doi: https://doi.org/10.1016/j.artint.2021.103535. URL https://www.sciencedirect.com/science/article/pii/S0004370221000862.

Richard S Sutton and Andrew G Barto. *Reinforcement learning: An introduction*. MIT press, 2018.

Christian Szegedy, Vincent Vanhoucke, Sergey Ioffe, Jon Shlens, and Zbigniew Wojna. Rethinking the inception architecture for computer vision.

In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 2818–2826, 2016.

Oriol Vinyals, Igor Babuschkin, Wojciech M Czarnecki, Michaël Mathieu, Andrew Dudzik, Junyoung Chung, David H Choi, Richard Powell, Timo Ewalds, Petko Georgiev, et al. Grandmaster level in starcraft ii using multi-agent reinforcement learning. *Nature*, 575(7782):350–354, 2019.

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

Alexander Zai and Brandon Brown. *Deep reinforcement learning in action*. Manning Publications, 2020.

Bolei Zhou, Hang Zhao, Xavier Puig, Sanja Fidler, Adela Barriuso, and Antonio Torralba. Scene parsing through ade20k dataset. In *2017 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 5122–5130, 2017. doi: 10.1109/CVPR.2017.544.

Bolei Zhou, Hang Zhao, Xavier Puig, Sanja Fidler, Adela Barriuso, and Antonio Torralba. Semantic understanding of scenes through the ade20k dataset. *International Journal of Computer Vision*, 127:302–321, 2018.

6 Appendix

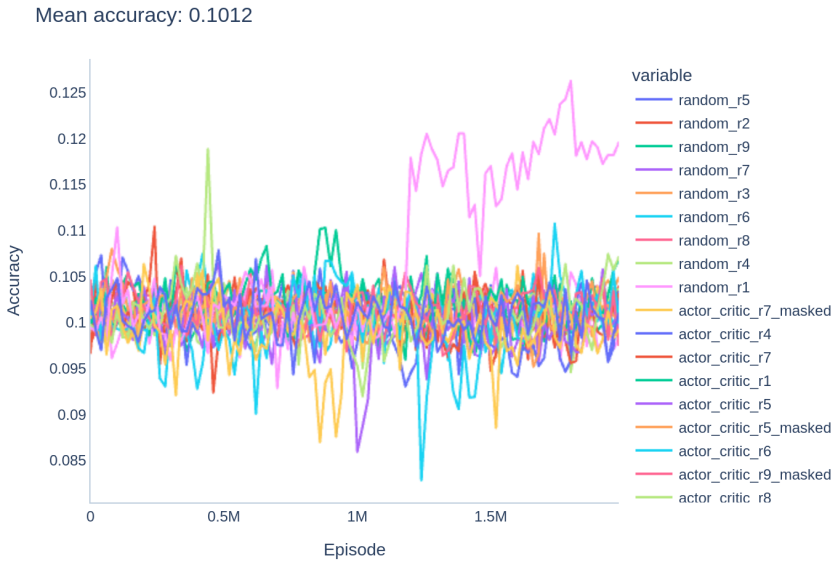
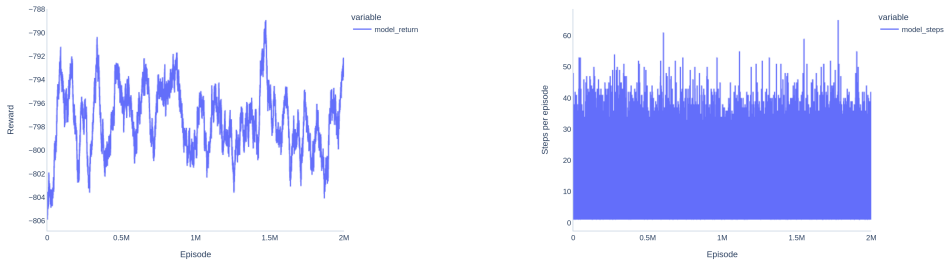


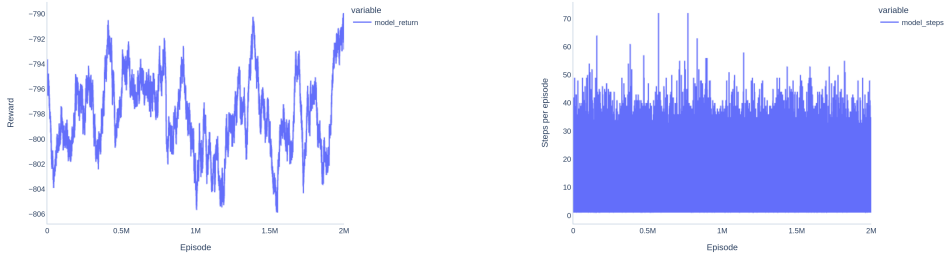
Figure 18: Accuracy of actor-critic and random agent for reward functions (normal and masked) over $2M$ episodes

6.1 Random Agents



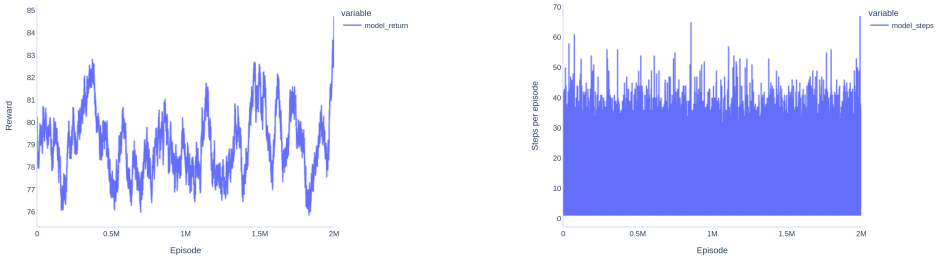
(a) 9 (b)

Figure 19: (a) reward over $2M$ episodes of a random agent with reward function r_2 (b) steps over $2M$ episodes of a random agent with reward function r_2



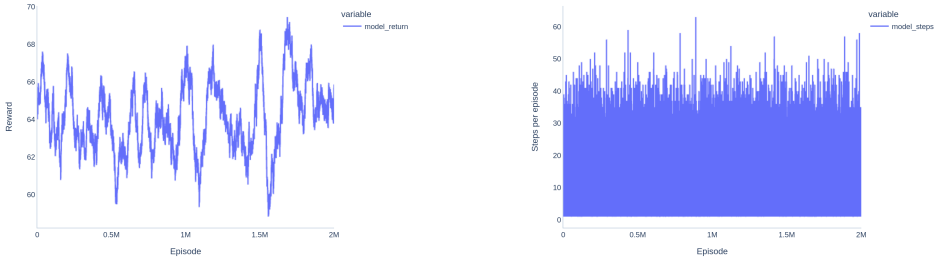
(a) 9 (b)

Figure 20: (a) reward over $2M$ episodes of a random agent with reward function r_3 (b) steps over $2M$ episodes of a random agent with reward function r_3



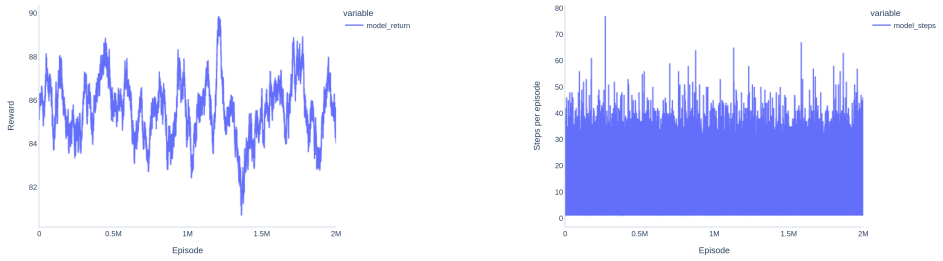
(a) 9 (b)

Figure 21: (a) reward over $2M$ episodes of a random agent with reward function r_4 (b) steps over $2M$ episodes of a random agent with reward function r_4



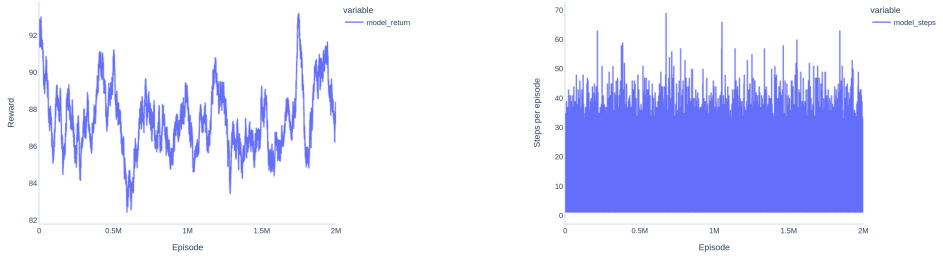
(a) 9 (b)

Figure 22: (a) reward over $2M$ episodes of a random agent with reward function r_5 (b) steps over $2M$ episodes of a random agent with reward function r_5



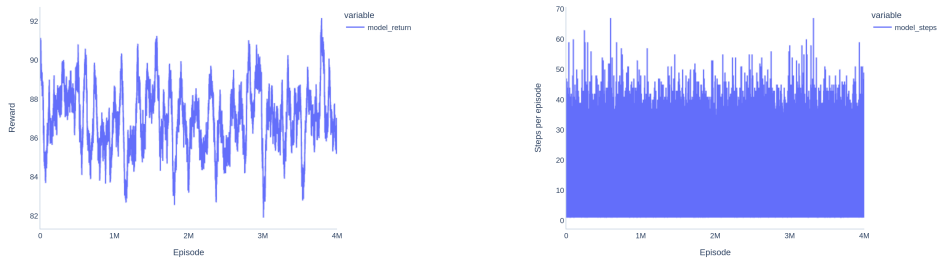
(a) 9 (b)

Figure 23: (a) reward over $2M$ episodes of a random agent with reward function r_6 (b) steps over $2M$ episodes of a random agent with reward function r_6



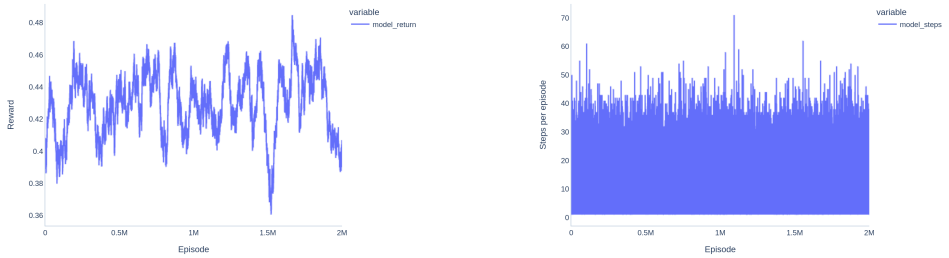
(a) 9 (b)

Figure 24: (a) reward over $2M$ episodes of a random agent with reward function r_7 (b) steps over $2M$ episodes of a random agent with reward function r_7



(a) 9 (b)

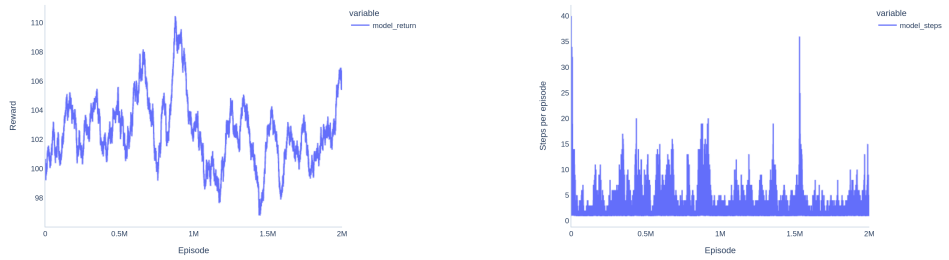
Figure 25: (a) reward over $2M$ episodes of a random agent with reward function r_8 (b) steps over $2M$ episodes of a random agent with reward function r_8



(a) 9 (b)

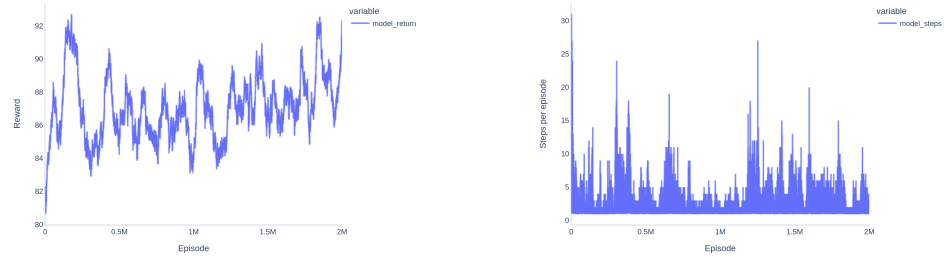
Figure 26: (a) reward over $2M$ episodes of a random agent with reward function r_9 (b) steps over $2M$ episodes of a random agent with reward function r_9

6.2 Actor-Critic Agents



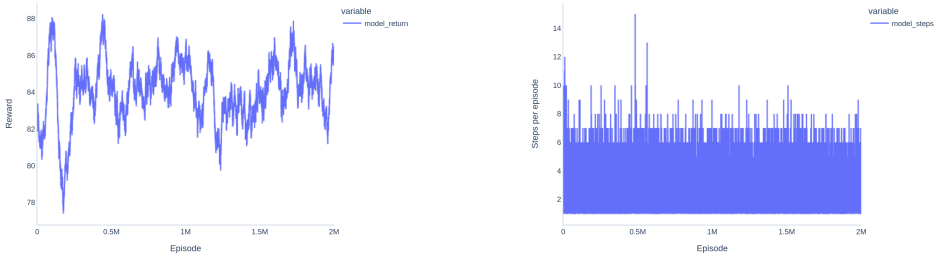
(a) 9 (b)

Figure 27: (a) reward over $2M$ episodes of an actor-critic agent with reward function r_1 (b) steps over $2M$ episodes of an actor-critic agent with reward function r_1



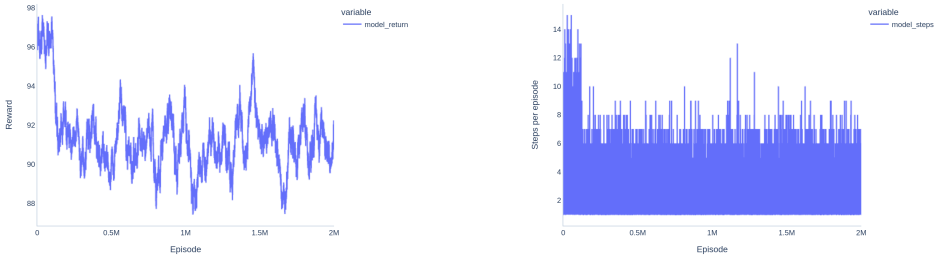
(a) 9 (b)

Figure 28: (a) reward over $2M$ episodes of an actor-critic agent with reward function r_4 (b) steps over $2M$ episodes of an actor-critic agent with reward function r_4



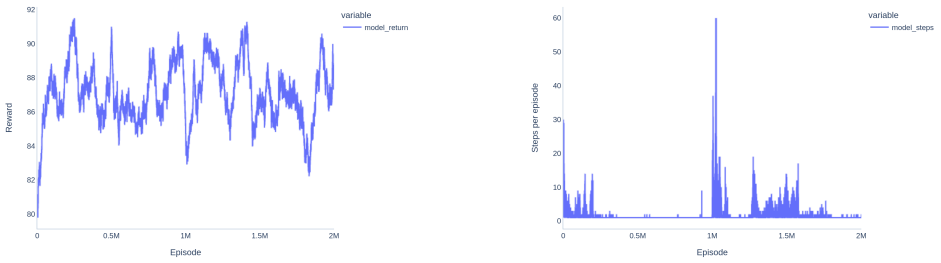
(a) 9 (b)

Figure 29: (a) reward over $2M$ episodes of an actor-critic agent with reward function r_4 and action masking (b) steps over $2M$ episodes of an actor-critic agent with reward function r_4 and action masking



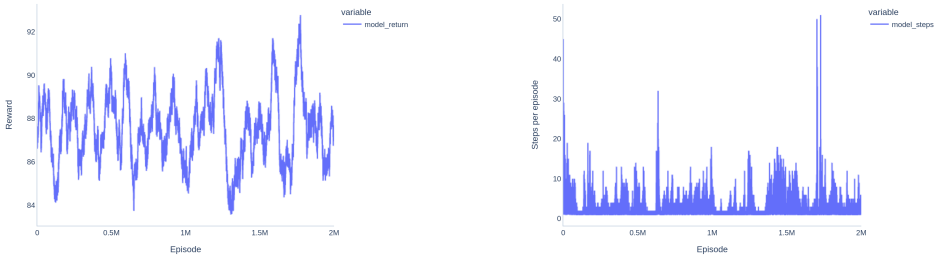
(a) 9 (b)

Figure 30: (a) reward over $2M$ episodes of an actor-critic agent with reward function r_5 and action masking (b) steps over $2M$ episodes of an actor-critic agent with reward function r_5 and action masking



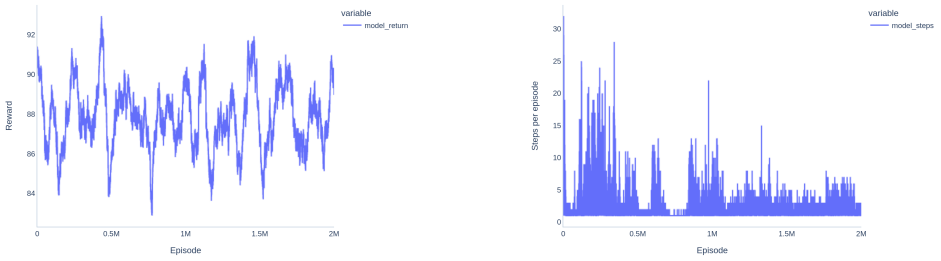
(a) 9 (b)

Figure 31: (a) reward over $2M$ episodes of an actor-critic agent with reward function r_5 (b) steps over $2M$ episodes of an actor-critic agent with reward function r_5



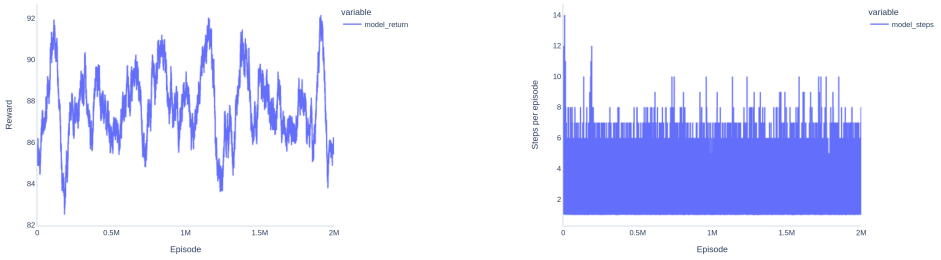
(a) 9 (b)

Figure 32: (a) reward over $2M$ episodes of an actor-critic agent with reward function r_6 (b) steps over $2M$ episodes of an actor-critic agent with reward function r_6



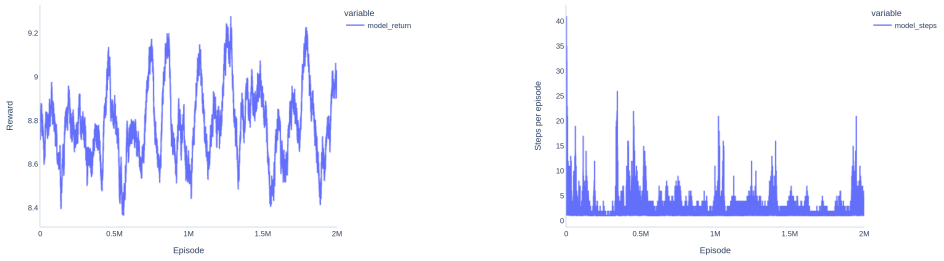
(a) 9 (b)

Figure 33: (a) reward over $2M$ episodes of an actor-critic agent with reward function r_7 (b) steps over $2M$ episodes of an actor-critic agent with reward function r_7



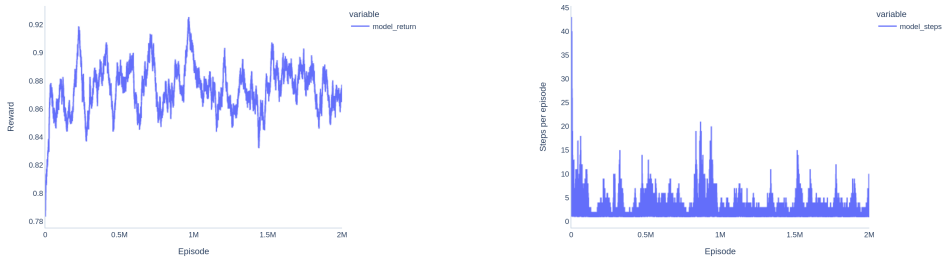
(a) 9 (b)

Figure 34: (a) reward over $2M$ episodes of an actor-critic agent with reward function r_7 and action masking (b) steps over $2M$ episodes of an actor-critic agent with reward function r_7 and action masking



(a) 9 (b)

Figure 35: (a) reward over $2M$ episodes of an actor-critic agent with reward function r_8 (b) steps over $2M$ episodes of an actor-critic agent with reward function r_8



(a) 9 (b)

Figure 36: (a) reward over $2M$ episodes of an actor-critic agent with reward function r_9 (b) steps over $2M$ episodes of an actor-critic agent with reward function r_9

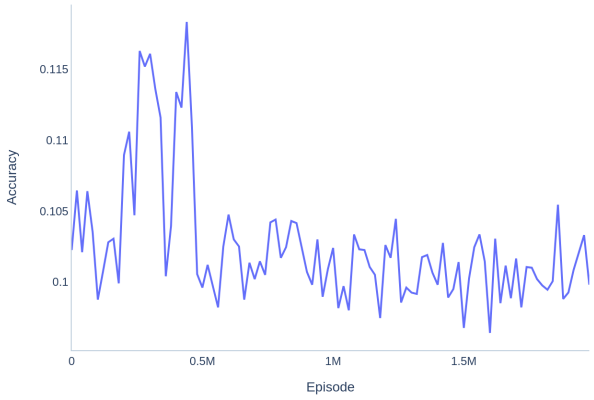


Figure 37: Experiment 1: Accuracy of r_1 with action masking over 2m episodes

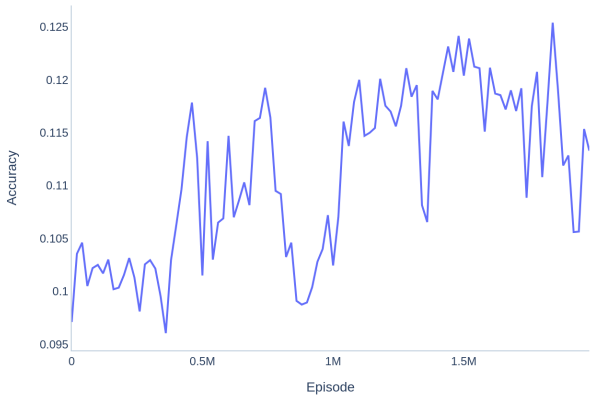


Figure 38: Experiment 2: Accuracy of r_1 with action masking over 2m episodes

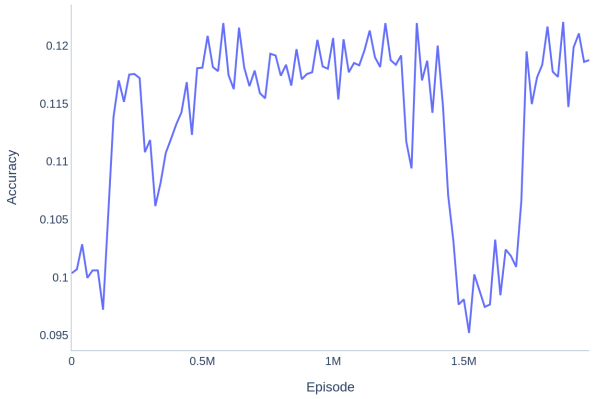


Figure 39: Experiment 3: Accuracy of r_1 with action masking over 2m episodes

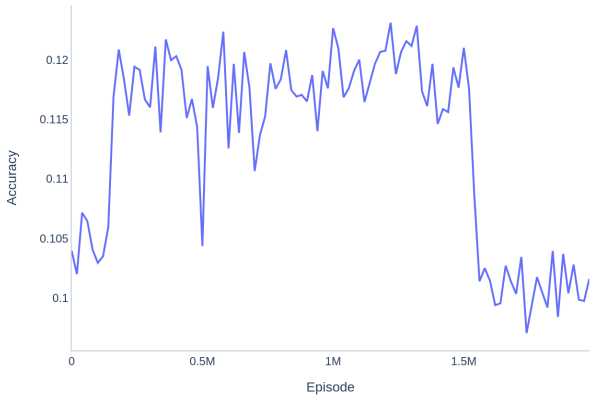


Figure 40: Experiment 4: Accuracy of r_1 with action masking over 2m episodes