Reader: Model-based language-instructed reinforcement learning

Anonymous ACL submission

Abstract

We explore how we can build accurate world models, which are partially specified by language, and how we can plan with them in the face of novelty and uncertainty. We propose the first model-based reinforcement learning approach to tackle the environment Read To Fight Monsters (Zhong et al., 2019), a grounded policy learning problem. In RTFM an agent has to reason over a set of rules and a goal, both described in a language manual, and the observations, while taking into account the uncertainty arising from the stochasticity of the environment, in order to generalize successfully its policy to test episodes. We demonstrate the superior performance and sample efficiency of our model-based approach to the existing model-free SOTA agents in eight variants of RTFM. Furthermore, we show how the agent's plans can be inspected, which represents progress towards more interpretable agents.

1 Introduction

Intelligent agents have the ability of re-composing known concepts to draw conclusions about new problems and this translates into the acquisition of very robust and general behaviours. Current Reinforcement Learning (RL) agents typically lack this ability and they need to be re-trained for every new problem; in contrast language models exhibit greater generalization abilities and provide a more flexible approach to solve multiple tasks with a single model. Thus language-conditioned RL is a flourishing area of research.

On the other hand, language models are trained exclusively on textual inputs and struggle to ground the meaning of the words to real world dynamics. Multiple interactive environments have been proposed as a testbed for learning how to ground language (Chevalier-Boisvert et al., 2018; Zhong et al., 2019; Ruis et al., 2020; Küttler et al., 2020). While prior work mostly focuses on Behavioural Cloning or model-free RL, we argue for a Model-Based Reinforcement Learning (MBRL) approach, as this effectively decouples the problem of understanding how the world works from the problem of acting optimally in the world in order to solve one or more tasks. Concretely, MBRL inherits the advantages of model-free RL of learning from scratch or from sub-optimal behaviour, while being orders of magnitude more sample efficient than the model-free counterpart. Furthermore, it has the added value of being more interpretable and explainable. In fact, a decision made by a MBRL agent can be accompanied by human-interpretable examples of likely future trajectories that are taken into account by the model in making such a decision.

In this work, we focus on Read To Fight Monsters (RTFM), a challenging benchmark for testing grounded language understanding in the context of reinforcement learning proposed by Zhong et al. (2019). RTFM tests the acquisition of complex reading skills in RL agents in order to solve completely new tasks based on written descriptions of the task dynamics and goal. Critically, the written information provided is not enough on its own to obtain an optimal policy, but the agent needs to cross-reference it multiple times with the current state of the environment, in order to figure out a plan of action.

We make the following contributions: first, we formulate a language-instructed MBRL method for solving RTFM and show how to train an agent in this environment (see Fig. 1). Our method, named Reader (for REinforcement learning Agent for Discrete Environments with wRitten instructions), explicitly models the stochastic changes in the discrete environment and performs planning with a stochastic variant of Monte Carlo Tree Search (MCTS, Kocsis and Szepesvári, 2006). We then demonstrate the effectiveness of Reader on eight variants of RTFM, showing better performance and sample efficiency than the SOTA model-free agent

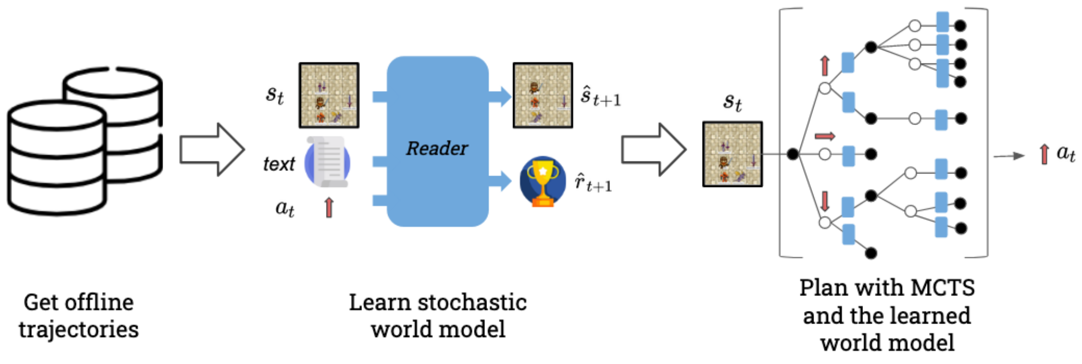


Figure 1: High-level view of the proposed method. We collect trajectories in the environment with behavioural policies, then use them to learn Reader, a discrete stochastic world model, and finally deploy Reader at test time to plan with Monte Carlo Tree Search (MCTS).

txt2π from Zhong et al. (2019): Reader surpasses txt2π on all the eight RTFM variants when both methods are trained on 1M frames of experience and is better on five and on par on the remaining three even when txt2π is trained on 100 times more frames[1]. Furthermore, Reader has comparable performance with a strong model-based Transformer baseline, while being more than eight times faster during planning. Finally, we show how Reader's plans can be visualised and how those plans change accordingly to the written instructions, making the language-instructed model-based approach more interpretable and trustworthy than the model-free counterpart.

2 Related Work

Language Grounding and Understanding

Chevalier-Boisvert et al. (2018) proposes BabyAI, a benchmark for studying the sample efficiency of Imitation Learning and RL methods in tasks where the goal is specified in natural language. Ruis et al. (2020) instead studies the problem of compositional generalization in situated Language Understanding in a Supervised Learning setup with the gSCAN benchmark, where agents have to map language instructions to corresponding action sequences. Narasimhan et al. (2018) considers a transfer learning setup between pairs of grid-world environments, where entities are annotated with language information about their role and behaviour. Bahdanau et al. (2018) learns how to train reward models from language specifications and expert trajectories and shows the usefulness of such reward models in training RL agents to accomplish language specified tasks.

Our work builds on the environment RTFM, introduced in Zhong et al. (2019), with the main target of solving such environment with a model-based approach instead of a model-free one. Similar work on grounding language can be found in Hanjie et al. (2021), which introduces the MESSENGER environment; a notable difference between RTFM and MESSENGER is that in the latter the co-reference of the entities and their names is harder to learn, but the reasoning steps to perform are easier. Zhong et al. (2021) proposes SILG, a unified interface for RTFM, MESSENGER, NetHack (Küttler et al., 2020) and symbolic abstractions of ALFRED (Shridhar et al., 2020) and Touchdown (Chen et al., 2019); each environment poses its own unique challenges, like learning multi-hop reasoning or grounding co-references, dealing with partial observability, large action spaces or rich natural language instructions and annotations. Both the baseline in Zhong et al. (2021) and the following work on SILG in Zhong et al. (2022) include in the benchmark only the simplest, stationary variation of RTFM and focus instead on finding model-free algorithms that are able to deal with all 5 SILG environments.

In this work, we focus only on RTFM, we consider pre-existing and new dynamical levels of the game and we propose the first model-based approach for this environment.

Model-based Reinforcement Learning

AlphaGo (Silver et al., 2016) is the first work demonstrating SOTA performance of MBRL with a MCTS-based agent which has access to the true

[1]For a fair comparison of the methods, we do not train txt2π with curriculum learning.

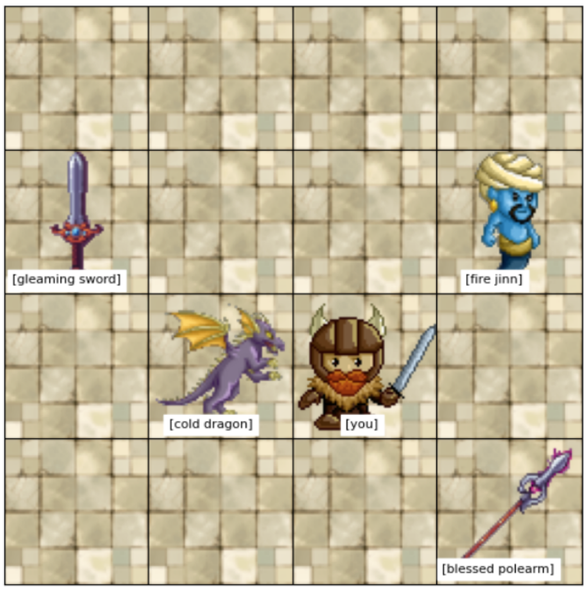
simulator of the game of Go and learns with neural networks both a prior over promising actions and an evaluation function to estimate the values of game configurations. MuZero (Schrittwieser et al., 2019) lifts the constraint of having access to a simulator of the environment, by learning a latent model of it and using it to perform a variant of MCTS in the latent space with the aid of a value function and a policy.

In this work, for simplicity we do not use policy and value functions as it is not our focus, but they could be beneficial to reduce the simulation budget of our MCTS agent further and they would certainly be necessary to scale up this approach to higher dimensional action-spaces and longer-horizon tasks. Overall our contribution is orthogonal to the learning of policy and value networks for MCTS algorithms, as we aim to learn a good model of a stochastic environment and a complex language-dependent reward function that is able to generalize to new environments.

Most works in MBRL assume a deterministic environment (as it is the case for example in chess and Go) or weakly stochastic (as Atari) and show dramatic drops in performance when applied to stochastic ones. Ozair et al. (2021) demonstrates how MuZero performance deteriorates when playing chess if the opponent is considered part of the environment (version of chess denoted *single player*) and the algorithm cannot enumerate its actions, but has to learn to model them as possible stochastic outcomes.

The Vector Quantized Model (VQM) in Ozair et al. (2021) probably has the most similar approach to ours, learning a "State VQVAE" to extract discrete latent codes and then learning a "Transition model" which, given a latent state-action pair and a discrete latent code, produces the next latent state.

Another notable line of work capable of dealing with stochastic environments can be found in Hafner et al. (2018), Hafner et al. (2019) and Hafner et al. (2020) . These works are based on the Recurrent State Space Model (Hafner et al., 2018) and of particular interest is Hafner et al. (2020), as it also uses discrete latent variables to capture the stochasticity in the environment dynamics. The discrete variables are trained with straight-through gradients and the obtained model is used to produce synthetic data in the latent space to train a model-free algorithm instead of being used for planning. However, none of these models involves language.



Goal: `Fight the order of the forest.`
Manual: `Fire monsters are weak against gleaming items. Lightning monsters are defeated by grandmasters items. Use shimmering items for poison monsters. Rebel enclave has the following members: demon. Dragon are star alliance. Jinn are on the order of the forest team. Cold monsters are weak against blessed items.`
Inventory: `empty.`

Figure 2: Example of a frame from the RTFM environment with two monsters in the natural language version. Together with the grid observation (above), the agent is provided with the goal, manual and the inventory (below).

3 Read To Fight Monsters

Read To Fight Monsters (RTFM) is a challenging benchmark proposed by Zhong et al. (2019) for testing grounded language understanding in the context of RL. RTFM tests the acquisition of complex reading skills in RL agents in order to solve completely new tasks based on written descriptions of the task dynamics (also called manual m) and goal g. Crucially, it is not enough to consult the written information in order to obtain an optimal policy, but the agent needs to perform a multi-step reasoning between such information and the current state of the environment in order to figure out a plan of action.

To elucidate the reasoning steps and reading skills needed to win an episode, we go through the concrete example reported in Fig. 2.

1. From the goal extract which team to defeat (`order of the forest`).
2. Search in the manual which monster is assigned to that team (`jinn`).
3. Find in the map the element type of the target monster (`fire`).

4. Search in the manual which modifier beats the target monster's type (`gleaming`).
5. Find in the map the item with the correct modifiers (`gleaming sword`).
6. Pick up the correct item (`gleaming sword`).
7. Engage the correct monster (`fire jinn`) in combat with the correct item (`gleaming sword`).

The agent is given a reward of $+1$ if it engages the correct monster in combat while carrying the correct item, -1 in any other encounter with a monster and a reward of 0 for all intermediate steps.

As every episode contains a procedurally generated set of (monster, element) pairs, (item, modifier) pairs, goal and manual entries, the agent cannot solve new episodes memorizing what is the right pair of item to take and monster to fight, but it has to learn to read the goal and the manual and cross-reference them with the environment observation. The agent's performance is tested on episodes generated in such a way that no assignments of monster-team-modifier-element are ever seen during training, to test whether the agent is able to generalize via reading to new environments with unseen dynamics. For more in depth description of the environment and how it is generated the reader can refer to Zhong et al. (2019).

We start considering two variants of the original RTFM, the dynamical version with simple language and the dynamical version with natural language, denoted respectively `dyna` and `dyna+nl` in Zhong et al. (2019).

First, we notice that the SOTA agent txt2π (Zhong et al., 2019) cannot solve even the simpler `dyna` from scratch; instead it has to resort to curriculum learning. Second we notice that `dyna` and `dyna+nl` differ in two aspects, both concerning the way in which the manual and the goal are expressed.

The first difference is that `dyna` uses fixed language templates like `"gleaming beats fire"` instead of one of multiple crowd-sourced natural language reformulations, like `"fire monsters are weak against gleaming items"`; we denote them simple language (`sl`) and natural language (`nl`) respectively.

The second difference is that in `dyna` the sentences of the manual are always ordered in a specific way (e.g. the first sentence always refers to monsters of the `cold` element and the last sentence to the `star alliance` team), whereas in the `dyna+nl` task the order of the sentences is always shuffled; we call these factors `no_shuffle` and `shuffle` respectively.

Taking all of this into consideration, we propose to systematically analyse the RL methods' performances under the following factors of complexity: number of monsters (`one` or `two`), kind of language (`sl` or `nl`) and sentence ordering (`no_shuffle` or `shuffle`). Considering all possible combinations, we end up with eight RTFM variants; in particular we introduce the single monster variants to provide more gradual degrees of complexity to the RTFM challenge.

In the rest of the paper we will use the following notation: the original variant with two monsters, simple language and no shuffling, `dyna`, is named `(two, sl)`; similarly the original variant with two monsters, natural language and shuffling, `dyna+nl`, is named `(two, nl, shuffle)`. A consistent notation is adopted for the remaining six variants, which do not have a corresponding counterpart in the original RTFM.

4 Language-conditioned world model

The goal of any RL agent is to find a policy $\pi(a|s)$ that maximizes the expected cumulative reward $\mathbf{E}_\pi[\sum_{t=0}^{T} R_t]$ received from the environment in an episode if all actions are taken according to such a policy. In this work, we take the model-based approach to RL: we learn a language-conditioned stochastic model of the environment and use it to plan with a stochastic version of vanilla MCTS.

We name our method **Reader** (for REinforcement learning Agent for Discrete Environments with wRitten instructions). It is composed of a world model and a MCTS planning module. The world model consists of three components (see Fig. 3).

1. the **representation model**, an autoencoder which encodes the grid-world observations s_t into discrete codes z_t:

$$z_t = f(s_t, s_{t-1}, a_{t-1}),$$

2. the **transition model**, which predicts the discrete codes z_{t+1}:

$$p(z_{t+1} \mid s_t, a_t),$$

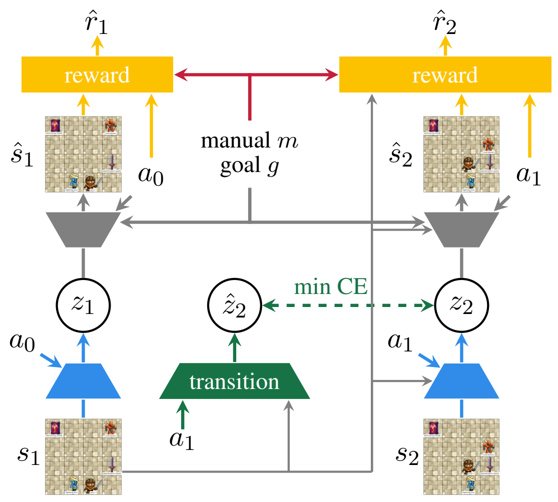


Figure 3: Components of language-conditioned world model: in blue and grey the encoder and decoder of the representation model, in green the transition model and in yellow the reward model. During training the three models are trained separately (in terms of computational graphs). At planning time, we use the transition model, followed by the decoder and by the reward model to predict respectively $\hat{z}_{t+1}, \hat{s}_{t+1}, , \hat{r}_{t+1}$.

3. the **reward model**, which predicts the next-step reward:

$$p(r_{t+1} \mid s_t, s_{t+1}, a_t, m, g).$$

Note that in contrast to existing model-based agents, our world model is conditioned on the episode-specific textual descriptions: the manual m which describes the roles of the monsters and the rules of the episode and the sentence g describing the agent's goal.

4.1 Representation model

The purpose of the representation model is to encode in a latent variable z_t the knowledge gained in observing the current state s_t, while knowing already the previous state s_{t-1} and action a_{t-1}; the gain in knowledge comes from the fact that the environment is stochastic. Once we have the representation model, we can train a transition model to learn the distribution of z_t conditional on s_{t-1} and a_{t-1}, as explained in the next section, and use it to simulate possible next states. Our representation model is based on vector-quantized variational autoencoders (VQVAE, van den Oord et al., 2017), a latent variable model with discrete latent codes. The VQVAE architecture is composed of an encoder, a decoder and a vector quantization layer in between. Similarly to (Ozair et al., 2021), we condition the representation z_t of the current state s_t on the previous state s_{t-1}:

$$z_t = f(s_t, s_{t-1}, a_{t-1}),$$

where z_t is a discrete code produced by the encoder f. The decoder d is trained to reconstruct the original state from the discrete code z_t given the previous state and action[2]:

$$\hat{s}_t = d(z_t, s_{t-1}, a_{t-1}, m).$$

The discrete codes are produced by the encoder in the same way it was done in the original VQ-VAE paper (van den Oord et al., 2017) by keeping a trained codebook of prototype vectors and selecting as a representation the prototype with the smallest distance to a continuous encoder output. We train the model end-to-end using the straight-through approximation for the vector quantization function when back-propagating through it. We use the three losses that were proposed by van den Oord et al. (2017) and implement the encoder and the decoder using a Transformer architecture (Vaswani et al., 2017). Since the RTFM environments have at most two sources of stochasticity (which correspond to two monsters), we modify the quantization layer of our representation model to produce two codes (z_a, z_b), such that the continuous output of the encoder is split into parts and the quantization is performed for the two parts independently. This choice gives a combinatorial inductive bias to the representations we are learning.

4.2 Transition model

The purpose of the transition model is to predict possible values of the next state s_{t+1} given the current state and the taken action. We implement the model by using an additional block (the green block in Fig. 3), which predicts the distribution $p(z_{t+1} \mid s_t, a_t)$ of the discrete representation z_{t+1} of the next state s_{t+1}. To simulate the next state s_{t+1} at the planning stage, the output of this transition block is passed though the decoder of the representation model.

We train the transition model concurrently with training the representation model, using the codes produced by the representation model as the transition model targets. We stop the gradients such that

[2]We denote as $\hat{s}$ and $\hat{r}$ the variables that are predicted by the model, in contrast with the variables s and r that are used as targets for training. For the discrete codes, we use z for the encoder's output and $\hat{z}$ for the samples drawn from the transition model.

the existence of the transition model does not affect the learned representations. We use a Transformer architecture for the transition model.

4.3 Reward model

A key challenge of RTFM is to model correctly the language-instructed reward function. Since the environment is stochastic, a natural choice for the reward function is $p(r_{t+1} \mid s_t, s_{t+1}, a_t, m, g)$, as including the next state s_{t+1} makes the function deterministic and thus easier to learn. This is only made possible by the ability of predicting the next state s_{t+1} with our stochastic transition model.

We train the reward model concurrently with the other two models and during training we use the true next state for s_{t+1}. The main neural architecture that we consider in Reader is one closely based on the txt2π actor-critic architecture proposed in Zhong et al. (2019); see appendix B for more details. For additional studies we also consider as an ablation a Transformer architecture.

Furthermore, since RTFM gives non-zero rewards only for terminal transitions, empirically we find beneficial in terms of sample efficiency and performance to train the reward function only on those transitions. For planning, we first predict if a transition is terminal or not; if the transition is non-terminal, we assign a reward of 0, if it is terminal, we predict the reward +1 or -1 with the reward model.

4.4 Transformer baseline world model

A possible alternative to our representation and transition models over the latent space is to have a model learn directly the distribution over the next state s_{t+1} auto-regressively:

$$p(s_{t+1}|c_t) = \prod_{ij=1}^{H*W} p(s_{t+1,ij}|s_{t+1,<ij}, c_t),$$

where $c_t = (s_t, a_t, m, g)$ is the conditional information available, using for example a single Transformer model (Vaswani et al., 2017). While this model is possibly more general and it can be trained in teacher-forcing mode to capture correctly the stochasticity of the environment, its auto-regressive modeling over the entire next state makes it impractical for planning, where a model can be used hundreds if not thousands of times during planning for every single time-step. We nonetheless consider this model as a baseline, dubbed "Transformer baseline"; to keep the comparison with Reader fair, in both cases we use the same reward model based on txt2π.

5 MCTS planning with the learned world model

To use the model for planning, we need to predict the next state $\hat{s}_{t+1}$ and reward $\hat{r}_{t+1}$ given the information available at the current step, that is s_t, a_t, m and g. We do that by first using the transition model to sample the next discrete latent code $\hat{z}_{t+1}$, then using the representation model decoder to get $\hat{s}_{t+1}$ and finally the reward model to get $\hat{r}_{t+1}$.

To extend MCTS to stochastic transitions, we use two types of nodes in the tree: state nodes and state-action nodes. The tree branches at state nodes by considering different actions that can be taken in the state and it branches at state-action nodes by looking at different stochastic outcomes. While we have control over the actions that we choose, the outcomes and corresponding rewards are always sampled by using the learned model. We index the possible outcomes from a state-action node based on the indices of the pair of discrete codes (z_a, z_b) sampled by the transition model; if the codes have been already sampled, we continue traversing the already-expanded tree, otherwise we expand the newly-sampled state node.

6 Experiments

Training

To ease the computational demands of the full RL pipeline, we consider an offline RL setup. We first collect a dataset of trajectories for each task, then we train the model on it without ever directly interacting with the real environment and finally we evaluate the MCTS agent equipped with the learned model by playing 1000 episodes in the test environments. To collect the datasets we use a random policy for 50% of the episodes and a vanilla MCTS policy[3] with access to a simulator of the real environment for the remaining 50% of the trajectories. We opted for this distribution of policies to show that our method does not rely on high-quality expert trajectories to work.

Each dataset is composed of 200k episodes, or roughly 1M frames; full details are reported in Table 3 in the appendix. We train the model with mini-batches of 1-timestep transitions sampled from

[3]We select the number of simulations per action of the MCTS policy in such a way that its performance is sub optimal.

Methods	Win rate				Training Frames
	`one sl`	`one sl shuffle`	`one nl`	`one nl shuffle`	
txt2π	$0.38_{.04}$	$0.41_{.02}$	$0.39_{.03}$	$0.42_{.01}$	1M
	$\mathbf{0.99}_{.01}$	$0.68_{.22}$	$0.50_{.01}$	$0.50_{.01}$	100M
Reader (ours)	$\mathbf{0.99}_{.01}$	$0.85_{.17}$	$\mathbf{0.79}_{.19}$	$\mathbf{0.63}_{.18}$	1M
Transformer baseline	$0.97_{.01}$	$\mathbf{0.90}_{.15}$	$0.75_{.19}$	$0.59_{.17}$	1M
Oracle MCTS	$0.97_{.01}$	$0.97_{.01}$	$0.97_{.01}$	$0.97_{.01}$	N/A

Table 1: **Comparison of methods (single monster)**. We evaluate every method with 5 independent training runs for every single-monster RTFM variant (with simple language, `sl`, or natural language, `nl`, and with or without shuffling). We report the average and standard deviation (over 5 runs) of the win rate over 1000 episodes. We also report the number of frames used for training each method.

Methods	Win rate				Training Frames
	`two sl`	`two sl shuffle`	`two nl`	`two nl shuffle`	
txt2π	$0.09_{.02}$	$0.09_{.02}$	$0.07_{.01}$	$0.08_{.02}$	1M
	$0.24_{.00}$	$0.24_{.00}$	$\mathbf{0.24}_{.01}$	$\mathbf{0.24}_{.01}$	100M
Reader (ours)	$\mathbf{0.50}_{.07}$	$\mathbf{0.47}_{.16}$	$\mathbf{0.24}_{.01}$	$\mathbf{0.24}_{.01}$	1M
Transformer baseline	$\mathbf{0.50}_{.07}$	$0.39_{.10}$	$0.23_{.03}$	$0.23_{.01}$	1M
Oracle MCTS	$0.83_{.01}$	$0.83_{.01}$	$0.83_{.01}$	$0.83_{.01}$	N/A

Table 2: **Comparison of methods (two monsters)**. We evaluate every method with 5 independent training runs for every two-monsters RTFM variant (with simple language, `sl`, or natural language, `nl`, and with or without shuffling). We report the average and standard deviation (over 5 runs) of the win rate over 1000 episodes. We also report the number of frames used for training each method.

the dataset and train the model components as described in Sec 4.

Results

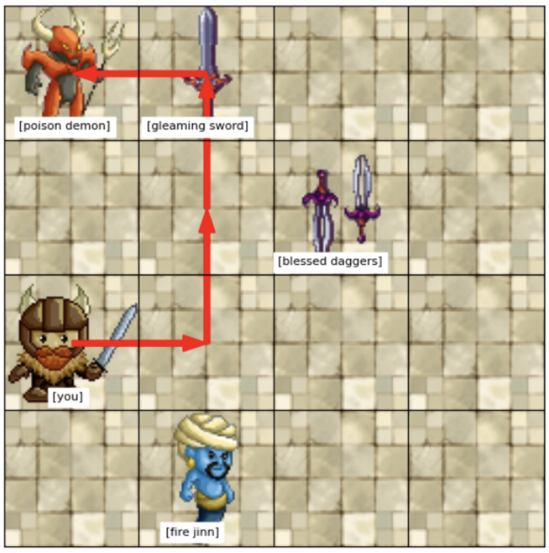
We train our Reader model on the eight RTFM tasks proposed in Sec. 3 and evaluate its performance with MCTS, using 400 simulations per time step.

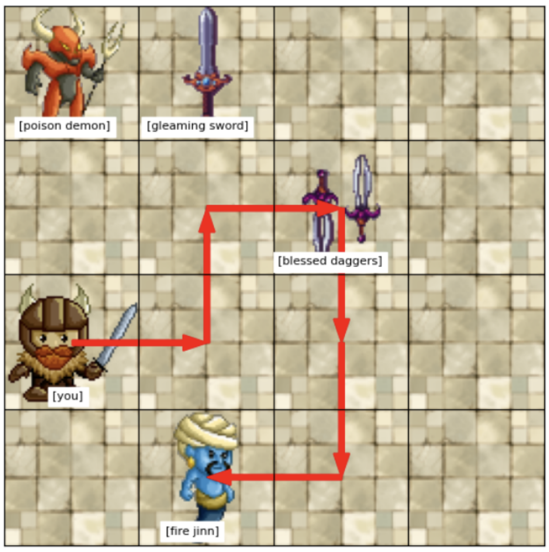
We compare against two strong agents: the first, which we call **Oracle MCTS**, uses MCTS with the ground-truth simulator of the environment, instead of using a learned world model; this represents a sort of upper bound for the performance of our agent. The second is the model-free agent **txt2π** proposed in (Zhong et al., 2019), which at the time of writing detains the state-of-the-art in RTFM environment. In contrast with the training procedure used in Zhong et al. (2019) to obtain the SOTA, we bar all the methods from resorting to curriculum learning, because it is expensive to study and to ablate systematically; furthermore, while certainly useful, it is not needed to evaluate the merits of RL algorithms and architectures. We also compare to our model-based **Transformer baseline** described in Sec. 4.4; this model is also evaluated with MCTS, but we use only 100 simulations per time step, as planning with a fully auto-regressive Transformer model is much slower than planning with Reader.

We report the results for the single monster variants in Table 1 and the ones for the two monsters variants in Table 2. Reader is superior in all eight tasks to the txt2π agent trained on 1M frames. Furthermore, when compared with the txt2π agent trained on 100M frames, Reader is still better on five out of the eight tasks and achieves equal performance on the remaining three.

Reader performance is also on par with that Transformer baseline, but it is much faster at planning: a prediction step with Reader takes approximately 12 ms, whereas one with the Transformer baseline takes about 102 ms (more than eight times slower). We provide further ablations of Reader's representation and reward model in Sec. D.

These results thus validate our method and show how Reader is a strong, sample-efficient and fast language-instructed model-based method. However RTFM still remains an open challenge, as none of the methods closes the gap with the Or-





(a) Goal: `defeat the order of the forest` (b) Goal: `defeat the star alliance`

Manual: `Grandmasters beat cold. Blessed beat fire. Shimmering beat lightning. Gleaming beat poison. Demon are order of the forest. Dragon are rebel enclave. Jinn are star alliance.`

Figure 4: **Interpretable plans**. Action sequences planned with MCTS and Reader for different possible goals and the same manual (reported above). Planned action sequences can be visually inspected to interpret the agent's decisions. Furthermore, given the same state, we can prompt the agent with different goals (or manuals) and see how the agent's plan changes accordingly, e.g. from (a) to (b). For ease of visualization in the environment considered the monsters are stationary.

acle MCTS in the harder variants[4].

Interpretability

Finally, in this section we show how we can answer questions like *"Why does the agent act in this way?"* and *"How would have the agent behaved if the goal were different?"*.

To do that, for a given state, manual and goal, we build a stochastic tree with MCTS and Reader. We then select the path from the root of the tree which at every state node selects the best action and at every state-action node selects the most likely outcome. Finally, we render with arrows the agent's movements corresponding to the action-sequence obtained above. With the same procedure, we can also prompt the agent with alternative language instructions and compare how the plans change accordingly; this lets us answer counterfactual questions about the agent's behavior. An example of this procedure can be seen in Fig. 4 and more are presented in Sec. F of the appendix.

7 Discussion and Conclusions

RL environments such as RTFM are great for developing models capable of natural language understanding. Prior work shows that the model-free approach might work well in those benchmarks, but it requires a large number of samples and it is not insightful with regard to the agent's understanding capabilities.

In this work we show that the model-based approach can improve both the performance and the sample-efficiency of RL agents. While the sample-efficiency gains were expected, we speculate that the performance gains in RTFM are mostly driven from improvements in estimating the complex language-dependent reward function; we think that disentangling the reward prediction from the environment stochasticity by conditioning the reward prediction on the next predicted frame is a key element in this regard. Moreover we provide concrete examples of how the agent's plans can be inspected and we consider this a step in the direction of more interpretable RL agents.

Finally, our experiments highlight that RTFM remains a challenging benchmark and that existing methods need a large number of samples to learn the RTFM reward. Therefore, more research is needed to improve the sample efficiency of models for grounded language learning; leveraging the out-of-the-box representation learning capabilities of pre-trained large language models seems a promising direction to explore.

[4]Interestingly enough, Reader surpasses the Oracle in the simplest variant. We explore this result in more depth in Sec. E of the appendix.

Limitations

We did not study the effect of the curriculum learning used in Zhong et al. (2019), nor compared with the results of txt2π trained with such protocol; we leave such study for future work. We did not test the approach on the more complex variants of RTFM presented in the original work by Zhong et al. (2019), as SOTA results on those variants are obtained through curriculum learning. We did not try our approach on other environments than RTFM, as it would be too large a step for a single paper. Language-instructed environments are quite different from one another as they test for different reasoning properties. As such, it would require some tailoring and considerable computational resources to extend our method to other environments like Alfred (Shridhar et al., 2020).

References

Dzmitry Bahdanau, Felix Hill, Jan Leike, Edward Hughes, Arian Hosseini, Pushmeet Kohli, and Edward Grefenstette. 2018. Learning to understand goal specifications by modelling reward. *arXiv preprint arXiv:1806.01946*.

Howard Chen, Alane Suhr, Dipendra Misra, Noah Snavely, and Yoav Artzi. 2019. Touchdown: Natural language navigation and spatial reasoning in visual street environments. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 12538–12547.

Maxime Chevalier-Boisvert, Dzmitry Bahdanau, Salem Lahlou, Lucas Willems, Chitwan Saharia, Thien Huu Nguyen, and Yoshua Bengio. 2018. Babyai: A platform to study the sample efficiency of grounded language learning.

Danijar Hafner, Timothy Lillicrap, Jimmy Ba, and Mohammad Norouzi. 2019. Dream to control: Learning behaviors by latent imagination.

Danijar Hafner, Timothy Lillicrap, Ian Fischer, Ruben Villegas, David Ha, Honglak Lee, and James Davidson. 2018. Learning latent dynamics for planning from pixels.

Danijar Hafner, Timothy Lillicrap, Mohammad Norouzi, and Jimmy Ba. 2020. Mastering atari with discrete world models.

Austin W Hanjie, Victor Zhong, and Karthik Narasimhan. 2021. Grounding language to entities and dynamics for generalization in reinforcement learning.

Levente Kocsis and Csaba Szepesvári. 2006. Bandit based monte-carlo planning. volume 4212 LNAI, pages 282–293. Springer Verlag.

Heinrich Küttler, Nantas Nardelli, Alexander Miller, Roberta Raileanu, Marco Selvatici, Edward Grefenstette, and Tim Rocktäschel. 2020. The nethack learning environment. *Advances in Neural Information Processing Systems*, 33:7671–7684.

Adrian Lancucki, Jan Chorowski, Guillaume Sanchez, Ricard Marxer, Nanxin Chen, Hans J. G. A. Dolfing, Sameer Khurana, Tanel Alumäe, and Antoine Laurent. 2020. Robust training of vector quantized bottleneck models. *CoRR*, abs/2005.08520.

Karthik Narasimhan, Regina Barzilay, and Tommi Jaakkola. 2018. Grounding language for transfer in deep reinforcement learning. *Journal of Artificial Intelligence Research*, 63:849–874.

Sherjil Ozair, Yazhe Li, Ali Razavi, Ioannis Antonoglou, Aäron Van Den Oord, and Oriol Vinyals. 2021. Vector quantized models for planning.

Laura Ruis, Jacob Andreas, Marco Baroni, Diane Bouchacourt, and Brenden M. Lake. 2020. A benchmark for systematic generalization in grounded language understanding.

Julian Schrittwieser, Ioannis Antonoglou, Thomas Hubert, Karen Simonyan, Laurent Sifre, Simon Schmitt, Arthur Guez, Edward Lockhart, Demis Hassabis, Thore Graepel, Timothy Lillicrap, and David Silver. 2019. Mastering atari, go, chess and shogi by planning with a learned model.

Mohit Shridhar, Jesse Thomason, Daniel Gordon, Yonatan Bisk, Winson Han, Roozbeh Mottaghi, Luke Zettlemoyer, and Dieter Fox. 2020. Alfred: A benchmark for interpreting grounded instructions for everyday tasks. In *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*, pages 10740–10749.

David Silver, Aja Huang, Chris J. Maddison, Arthur Guez, Laurent Sifre, George Van Den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Veda Panneershelvam, Marc Lanctot, Sander Dieleman, Dominik Grewe, John Nham, Nal Kalchbrenner, Ilya Sutskever, Timothy Lillicrap, Madeleine Leach, Koray Kavukcuoglu, Thore Graepel, and Demis Hassabis. 2016. Mastering the game of go with deep neural networks and tree search. *Nature*, 529:484–489.

Aäron van den Oord, Oriol Vinyals, and Koray Kavukcuoglu. 2017. Neural discrete representation learning. *CoRR*, abs/1711.00937.

Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Ł ukasz Kaiser, and Illia Polosukhin. 2017. Attention is all you need. In *Advances in Neural Information Processing Systems*, volume 30. Curran Associates, Inc.

Victor Zhong, Austin W Hanjie, Sida Wang, Karthik Narasimhan, and Luke Zettlemoyer. 2021. Silg: The multi-domain symbolic interactive language grounding benchmark. *Advances in Neural Information Processing Systems*, 34:21505–21519.

Victor Zhong, Jesse Mu, Luke Zettlemoyer, Edward Grefenstette, and Tim Rocktäschel. 2022. Improving policy learning via language dynamics distillation. *arXiv preprint arXiv:2210.00066*.

Victor Zhong, Tim Rocktäschel, and Edward Grefenstette. 2019. RTFM: generalising to novel environment dynamics via reading. *CoRR*, abs/1910.08210.

A Offline trajectories datasets

To collect the datasets we use a random policy for 50% of the episodes and a vanilla MCTS policy with access to a simulator of the real environment for the remaining 50% of the trajectories.

The MCTS policy used to collect data specifically uses 30 internal simulations per time-step, maximum rollout length of 10, a discount factor of 0.9 and the Upper Confidence Bound constant $c = 1$.

We report in Table 3 the dataset statistics for the single monster (`one`) and two monsters (`two`) variants of RTFM. Since all dynamical variants of RTFM have the same underlying mechanics (e.g. either one or two monsters and two items placed randomly in a grid world, the monsters move stochastically according to an unknown policy which is always the same in all the variants), all the dataset statistics are identical, up to stochastic fluctuations, for the `sl` / `nl` factors and the `no_shuffle` / `shuffle` factors.

Dataset	**one**	**two**
Tot. frames	1120k	971k
Non-terminal	920k	771k
Successes	116k (10%)	77k (8%)
Failures	88k (8%)	123k (13%)
Win rate (random)	17.3%	4.6%
Win rate (MCTS)	94.7%	71.0%

Table 3: Datasets collected for offline training of the model and relative statistics.

B Training details

We report in the following section the summary details about the architectures and training procedures used in this work.

Both the Representation model and the Transition model use the Transformer architecture. The Representation model comprises of the Encoder, the Decoder and the Vector Quantization layer; both the Encoder and the Decoder are implemented as two-layers Transformer encoder. The Transition model comprises a two-layer Transformer encoder (to encode s_t and a_t) and a two-layer Transformer decoder (to auto-regressively predict the indices of $(z_{a,t+1}, z_{b,t+1})$). All hyper-parameters of the Transformer layers are reported in Table 4, following PyTorch naming convention.

Hyper-parameters	**Values**
d_model	128
nhead	4
dim_feedforward	256
dropout	0.1
activation	gelu

Table 4: Transformer hyper-parameters for the Representation and Transition models.

The Vector Quantization layer is made of 2 codebooks of 32 codes with feature dimension 64 (half of the model dimension of the Encoder and Decoder model dimension). We use a commitment loss coefficient $\beta = 0.25$ and a codebook learning rate multiplier $\lambda = 5$. Following (Lancucki et al., 2020), we use KMeans ++ to reinitialize the codes during the first 6 epochs, once every 50 forward passes.

For the Reward model, we modify txt2π architecture as follows:

1. We remove the concatenation of 2D positional embeddings to the state;
2. We encode the next state and inventory in the same way as the current state and inventory are encoded in txt2π;
3. We concatenate along the channel dimension the embedded current state and next state;
4. We concatenate along the feature dimension the embedded current inventory and next inventory;
5. We adapt the txt2π layers to receive variables with double the amount of channels or features (because of the concatenation above);
6. We replace the linear layers for the policy and the value with linear layers to predict the reward class, the termination signal and the valid moves in the next state.

We use the same identical hyper-parameters as the ones used in txt2π: embedding dimension of 30, dimension of small RNN of 10, dimension of

Bi-LSTM of 100 and dimension of final representations of 400. For more details about the architecture, please refer to (Zhong et al., 2019).

For the Transformer baseline model, we use a encoder-decoder architecture, plus the Reward model from Reader. Encoder and decoder use two layers; all layers use model dimension of 256, feed-forward dimension of 1024, gelu activation function, dropout of 0.1. The choice of a bigger model was made to offset the fact that there is no Transition model, thus there are less layers in total (4 instead of 6).

We report the other hyper-parameters used during training in Table 5.

Hyper-parameters	Values
Batch size	512
Optimizer	Adam
Learning rate	10^{-4}
Lr warm-up steps	400
Epochs	114

Table 5: Hyper-parameters used.

Finally we divide the collected transitions according to their reward (0, +1, -1) and form batches with an equal number of samples from each class, as the 0 class represents roughly 79% of all transitions, while the +1 and -1 represent respectively 8% and 13% of the transitions[5].

C Computational resources used

GPU resources used for Table 1 and Table 2 are reported in Table 6, whereas the CPU resources for the MCTS evaluation of the agents are reported in Tab 7.

Model	single (h)	total (h)
Reader	13.5	540
Transformer baseline	10	400
txt2π	23	920
Total		1840

Table 6: GPU resources used for Table 1 and Table 2. We use 5 random seeds and train on 8 tasks.

All experiments use either the Tesla V100 or the AMD MI250X GPU models and the total amount of resources used to obtain the reported results is approximately 1840 GPU hours and 38720 CPU hours.

[5]This in the two-monsters case. In the single monster case, the percentages are slightly different, respectively 82%, 10% and 8%.

Model	single (h)	total (h)
Reader (64)	112	4480
Transformer baseline (64)	304	12160
txt2π (24)	552	22080
Total		38720

Table 7: CPU resources used for Table 1 and Table 2. We use 5 random seeds and evaluate on 8 tasks. Number of CPU cores used for every run is reported between parenthesis after the name of the model.

D Ablation studies

Codebook ablation

Ablation study with an earlier version of Reader shows on (`two`, `s1`) RTFM variant that using two codebooks instead of a single one gives a slight advantage (see Table 8).

Codes	Embedding dim	Win rate
32x32	64	**0.79 (0.02)**
32	128	0.67 (0.13)
512	128	0.75 (0.01)
1024	128	0.75 (0.03)

Table 8: **Reader codebook ablation**.

Reward model ablation

We also tried to substitute the txt2π Reward model with a Transformer-based one. What we found is that the Transformer architecture is superior in the environments without manual shuffling, but is clearly inferior when the shuffling happens (see Table 9). The overall performance averaged over the 8 RTFM variants is roughly equal, but while the Transformer architecture is promising for the Reward model, there is also some a clear barrier in that the 1D positional encodings for the manual do not seem to suffice for it to learn the reward function in the shuffle case; thus we chose txt2π architecture for our Reward model.

E Reader and Oracle MCTS performance

In Table 1 we show that the win rate of Reader is 2% higher than the win rate of Oracle MCTS for the most simple RTFM environment (`one`,`s1`). This is surprising at first sight, as the underlying planning algorithm is the same and, assuming that Reader learned the optimal reward function, the difference

Architecture	Win rate no shuffle	Win rate shuffle
txt2π	0.631	**0.547**
Transformer	**0.790**	0.371

Table 9: **Reward model ablation**. While the Transformer architecture is superior in the environments without manual shuffling, it is clearly inferior when the shuffling happens. Thus our main choice for the Reward model architecture is txt2π.

in performance must come from differences in the transition probabilities, as that is the only remaining difference. In the following of this section we explain how is it possible that planning with vanilla MCTS and a learned world model can yield better results than planning with the same MCTS algorithm but with the ground-truth simulator instead of the world model.

First, we need to keep in mind that vanilla MCTS is not necessarily an optimal planner, as it relies on a random policy to estimate the values of the leaf nodes. While using a random policy greatly simplifies the algorithm (which for example can be used out-of-the-box without any policy and value learning), it also implies that value estimates have high variance and are usually greatly underestimated.

Second, in the single monster case, for every state there always exists an action that is 100% safe to take, thus an optimal agent should be able to always avoid losing. By inspecting the failure cases of the Oracle MCTS we can see that the agent sometimes takes risky actions, as the ones reported in Fig. 5 and Fig. 6, where the agent decides to go straight for the item it needs, at the risk of encountering the monster.

These two points together imply that if the optimal action's Q-value is greatly underestimated due to high variance and the usage of the random rollouts, then it is possible that riskier actions' Q-values might seem higher. This in turn makes the MCTS sample more and more the risky action, leaving the optimal action under-explored and thus underestimated.

Why then using Reader's world model is able to improve the performance? It turns out that Reader's learned world model is biased towards sampling transitions where the monster moves towards the agent. This penalizes what we called risky actions, as the estimated risk is higher in Reader, and leads to safer behaviors which explain the increase in win rate from 97% to 99% in the (`one,s1`) RTFM variant. One of the possible ways in which we can show this bias is the following: we simulate episodes with a random policy until termination, we then consider the second to last state s and the last action a and we simulate with the ground-truth simulator n possible outcomes $\{s'^{(i)}\}_{i=1,\dots,n}$. We then create an identifier (e.g. a hash) for each unique outcome (in the single monster case, there are at most five, corresponding to all the possible movements the monster can do) and compute the total probability of reaching an outcome being a terminal state. We then estimate the total probability of reaching a terminal state with Reader's world model. The transition probabilities $p(s'|s,a)$ are given by the total sum of the probabilities of all the latent codes z that together with s and a are decoded into s':

$$p(s'|s,a) = \sum_{z \in Z} p(z|s,a)\delta(d(z,s,a,m),s'),$$

where $p(z|s,a)$ is Reader's transition model, $d(z,s,a,m)$ is the Reader's representation decoder and $\delta(\cdot,\cdot)$ is Dirac's delta distribution. We get a 78% probability of reaching a terminal state for the ground-truth simulator and 95% probability in the case of Reader's world model[6]. The bias found is coherent in terms of magnitude and sign with the explanation given above.

Next, in Fig. 5 and Fig. 6 we inspect a typical failure case, where we show examples of risky behavior that emerges from the Oracle MCTS, the transition probabilities for all the possible outcomes estimated from the ground truth simulator and Reader's world model and finally the Q-values of the top two actions when running MCTS with either method. We can see how Reader's bias in sampling the worst-case scenario helps in ranking the safer action higher than the risky one.

Finally, we can see in Fig. 6 how Reader's bias is dependent on the action. This can be explained by our training procedure, where in forming the mini-batches we up-sample the terminal transitions to balance the reward classes. As the training of the reward model is independent from the training of the representation and transition models, we can fix this issue by sampling separate mini-batches for the two parts of the model and re-balancing the classes only in the case of the reward model.

[6]Note that these are conditional probabilities on the fact that at least one outcome is a terminal state. The conditioning is coming from the procedure with which we pick (s,a).

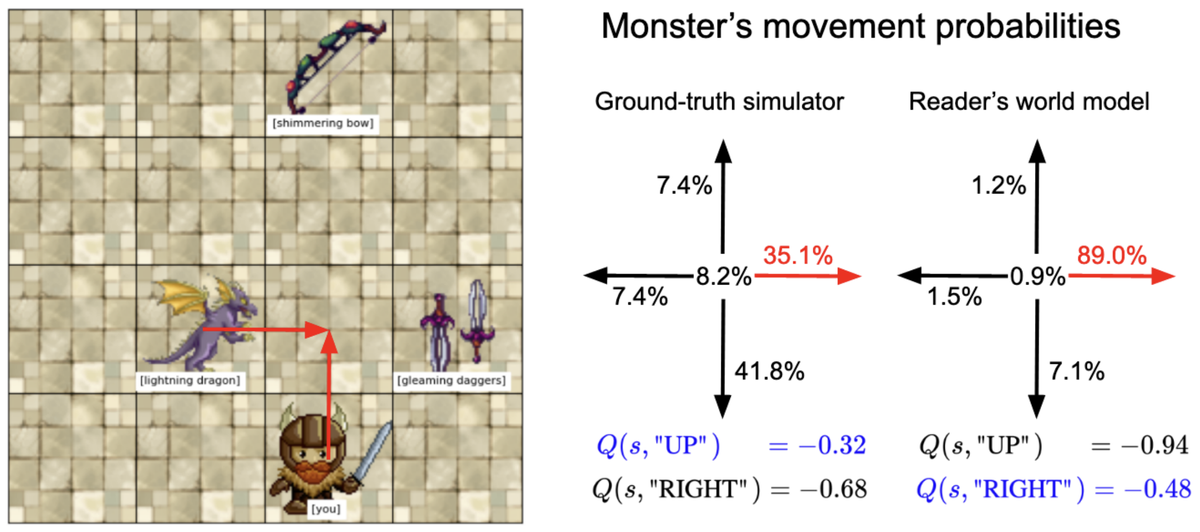

[shimmering bow]
[lightning dragon]
[gleaming daggers]
[you]
Monster's movement probabilities
Ground-truth simulator
7.4%
7.4%
8.2%
35.1%
41.8%
$Q(s, \text{"UP"}) = -0.32$
$Q(s, \text{"RIGHT"}) = -0.68$
Reader's world model
1.2%
1.5%
0.9%
89.0%
7.1%
$Q(s, \text{"UP"}) = -0.94$
$Q(s, \text{"RIGHT"}) = -0.48$

Figure 5: **Oracle MCTS failure case (first example).** On the left is represented a state, with red arrows highlighting the action that the Oracle MCTS decides to execute and the movement of the monster that could lead to failing the episode. On the right are reported the probabilities for the different movements of the monster (in red the one that would make the agent lose) and Q-values estimated via MCTS for the best two actions (in blue the one for the highest Q-value). We report these values both for the ground-truth simulator and for Reader's world model. What emerges is that Reader's world model is over-sampling the outcomes that lead to terminal states, resulting in a more conservative behavior (i.e. in a different choice of actions).

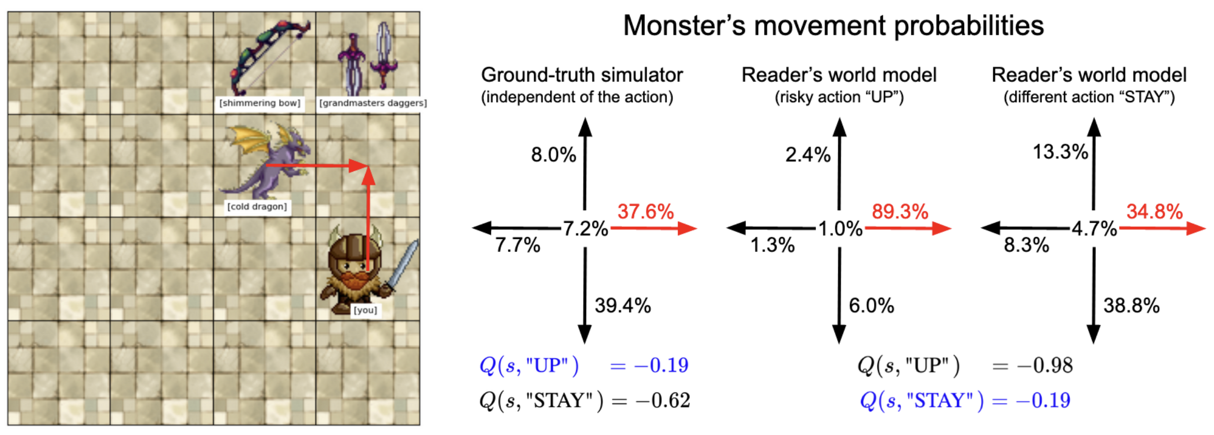


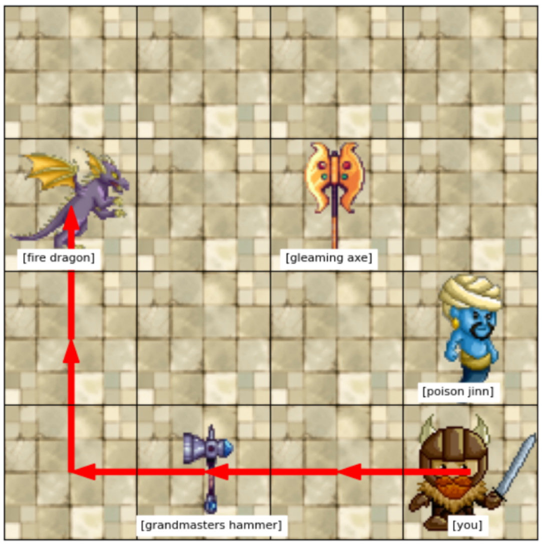
Figure 6: **Oracle MCTS failure case (second example).** On the left is represented a state, with red arrows highlighting the action that the Oracle MCTS decides to execute and the movement of the monster that could lead to failing the episode. On the right are reported the probabilities for the different movements of the monster (in red the one that would make the agent lose) and Q-values estimated via MCTS for the best two actions (in blue the one for the highest Q-value). We report these values both for the ground-truth simulator and for Reader's world model conditioned on two different actions. What emerges is that Reader's world model is biased only when an action can lead to a terminal state (action UP in this case) and it estimates quite accurately the probabilities when conditioned on other actions (e.g. action STAY in the example). This bias is most likely an artifact of the training procedure, where the terminal transitions are up-sampled to balance the reward classes.

F More interpretability results

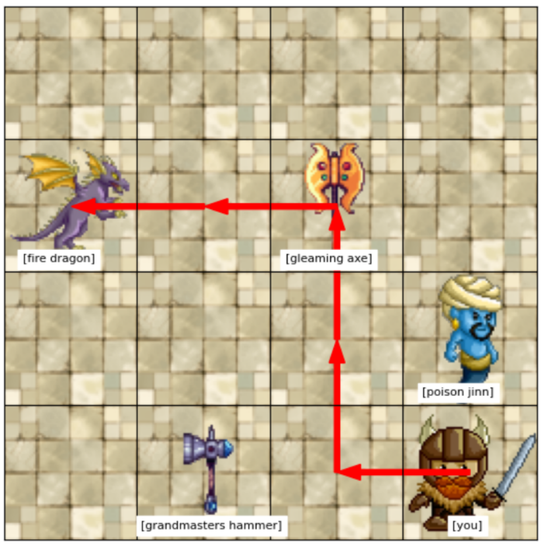
Similarly to Fig. 4, we can show how the agent's plans change if we change the manual instead of the goal.

In Fig. 7 we effectively change the (item, monster) assignment by swapping the modifiers of the two items present in the state. This means that to achieve the same goal (e.g. killing the `fire dragon`) the agent has to collect a different item in the two scenarios and it does that correctly (on the left it plans to use the `grandmasters hammer` as `grandmasters beat fire`, whereas on the right it plans to use the `gleaming axe` as `gleaming beat fire`).

In Fig. 8 instead we change the (monster, team) assignment, such that the same goal (`defeat the star alliance`) effectively asks to defeat a different monster in the two cases (`dragon` on the left, `jinn` on the right). The agent identifies correclty the target monster in the two cases and the correct item to defeat such monster (`grandmasters hammer` on the left, as `grandmasters beat fire` and the `dragon` is of the `fire` type, `gleaming axe` on the right, as `gleaming beat poison` and the `jinn` is of the `poison` type).

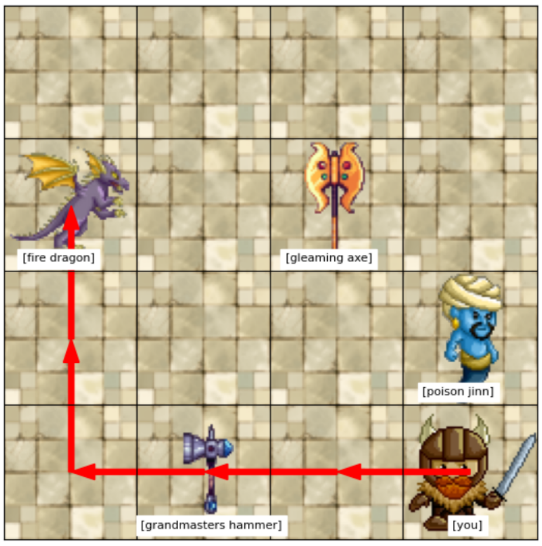


(a) Goal: Defeat the star alliance.
Manual: Blessed beat cold. Grandmasters beat fire.
Shimmering beat lightning. Gleaming beat poison.
Demon are order of the forest. Jinn are rebel enclave.
Dragon are star alliance.

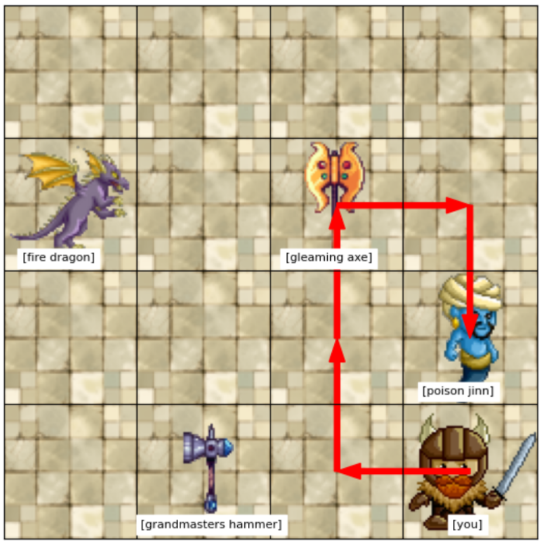


(b) Goal: Defeat the star alliance.
Manual: Blessed beat cold. Gleaming beat fire.
Shimmering beat lightning. Grandmasters beat poison.
Demon are order of the forest. Jinn are rebel enclave.
Dragon are star alliance.

Figure 7: **Interpretable plans for (item modifiers, monster elements) assignments**. Action sequences planned with MCTS and the learned world model for different possible (item modifiers, monster elements) assignments in the manual (changes highlighted in the two captions). Planned action sequences can be visually inspected to interpret the agent's decisions. Furthermore, given the same state, we can prompt the agent with different (item modifiers, monster elements) assignments and see how the agent's plan changes accordingly, e.g. from (a) to (b). For ease of visualization the environment is deterministic.



(a) Goal: Defeat the star alliance.
Manual: Blessed beat cold. Grandmasters beat fire. Shimmering beat lightning. Gleaming beat poison. Demon are order of the forest. Jinn are rebel enclave. Dragon are star alliance.



(b) Goal: Defeat the star alliance.
Manual: Blessed beat cold. Grandmasters beat fire. Shimmering beat lightning. Gleaming beat poison. Demon are order of the forest. Dragon are rebel enclave. Jinn are star alliance.

Figure 8: **Interpretable plans for (monster, team) assignments**. Action sequences planned with MCTS and the learned world model for different possible (monster, team) assignments in the manual (changes highlighted in the two captions). Planned action sequences can be visually inspected to interpret the agent's decisions. Furthermore, given the same state, we can prompt the agent with different (monster, team) assignments and see how the agent's plan changes accordingly, e.g. from (a) to (b). For ease of visualization the environment is deterministic.