AdventumRL: A Quest Based Reinforcement Learning API

Kyle Xiao, Kushagr Singh, Mark Riedl

October 2019

Abstract

We propose AdventumRL, an API framework for complex mission-based reinforcement learning in a Minecraft setting. In this research, a set of encodings are defined on top of tools in Malmo, an open source Minecraft reinforcement learning framework. We propose a framework grammar for encoding states, actions, transitions, and goals that can represent these challenges for a hierarchical RL scenario, The described methods define a logical grammar and interaction encoding schema that can support quests with logical dependencies. Proof of concept reinforcement learning agents are constructed using a tabular agent and deep Q learner.

1 Introduction

Minecraft is an open-world sandbox construction game that provides users with the capability to explore and interact with entities. It provides a continuous state representation as well as a sufficiently expansive world where we can define complex hierarchical constructions and objectives [2]. Minecraft has proven to be a popular platform to teach agents via reinforcement learning. Reinforcement learning is the problem of learning a control policy $\pi(s_t, a_{t-1}) = a_t$ that provides an action a_t that, if taken and the policy is followed henceforth, results in the maximum expected future reward. The learner is referred to as the agent, which selects actions in the environment and possibly receives a reward while a transition is made into a new state. Example of reinforcment learning agents in Minecraft include navigation-based tasks [6] [7] [4] and block placing [1], which for human players are typically primitive actions that are performed to achieve a higher goal.

Project Malmo [5] provides an API for reinforcement learning agents to control virtual characters within a Minecraft map. Although Minecraft—and Malmo in particular—offers an expansive environment with many flexible features, the game itself has seen limited use for reinforcement learning involving sequences of subgoals (outside of building). Other computer games—such as role-playing games—and also the real world often involve the interleaving of navigation and interactions with objects and entities in order to execute a plan and achieve a goal. In computer games, these sequences of sub-goals are often referred to as *quests*. Quests can be simple, such as picking up and delivering an object to another entity, or complex, such as locating and using a set of keys to open doors in a particular order. In general, learning to solve quests in games is a precursor for more complex sequential tasks in the real world.

For the scope of this project, a quest is a series of discrete high-level actions the agent must perform to complete a game. A corresponding goal is a configuration of facts or propositions about the world that the agent must achieve through its actions. Quest-based reinforcement learning refers to a scenario of reinforcement learning that structures learning as a quest. This differs from general reinforcement learning in that multiple low level actions may be needed to make a single transition in the propositional configuration space, while general reinforcement learning

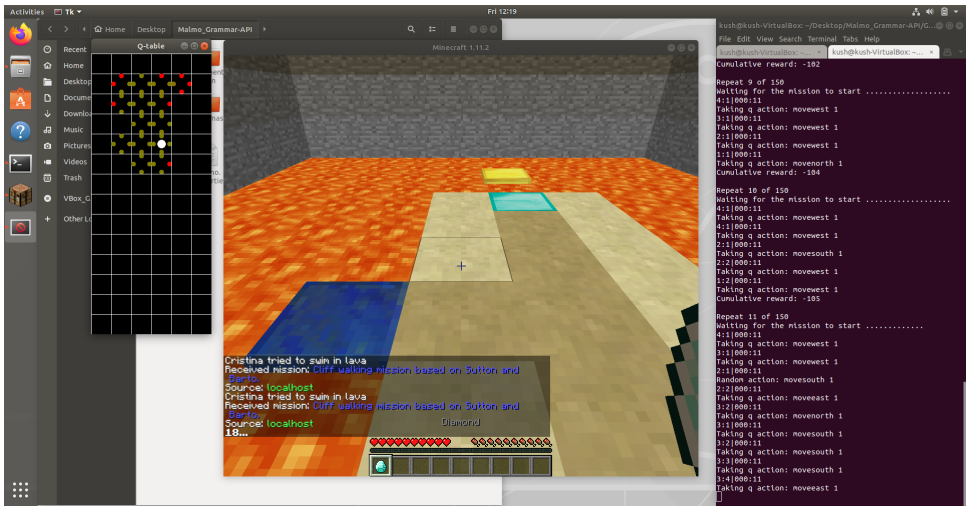

Figure 1: An example of a quest-based reinforcement learning scenario described in the Case Study section, where an agent attempt to escape out of a locked room.

techniques have no inbuilt structure separating nascent, higher level facts and transitions about the world from the general state space.

We present a framework—*AdventumRL*[1]—that extends Project Malmo by providing a grammar for encoding states, actions, transitions, and goals that can be used to express quests in terms of high-level state abstractions (e.g., "the non-player character has the key") that are easy to read and write. This framework can help produce richer reinforcement learning tasks in Minecraft by allowing developers to reduce joint-action spaces by keeping only high-level information, facilitate game theoretic payoff search scenarios through a context-free grammar, grant users the ability to assign individual rewards for accomplishing subgoals in cooperative tasks, and promote an inbuilt method for describing complicated propositional structures. Current state of the art reinforcement learning tools such as Malmo encode this information via MissionSpec, which defines a map, reward signal, consumable goods, and set of observations and actions [5]. However, these tools have difficulty communicating complex signals among multiple agents as well as high level objectives outside of navigation-based goals.

These richer capabilities allow exploration into difficult reinforcement learning tasks. Such challenges include, for instance, reducing joint-action spaces for multi-agent reinforcement learning. Other use cases include modeling game-theoretic events in events and specifying credit-assignment mechanisms for subtasks. These features could be used in hard explorations task scenarios, such as Montezumas Revenge. [10]

In AdventumRL, a set of encodings are defined on top of the current Malmo tools. States are represented via first-order logic on entities relevant to the problem space. The described methods define a logical grammar and encoding schema. A proof of concept reinforcement learning agent is then constructed.

2 Quest Grammar

The questing scenario is a proposed structure to reinforcement learning missions whereby high level representations are formed from more primitive state space information, inspired by Textworld [3]. This high level representation is referred to as the propositional space,

[1] *Adventum* is latin for "adventure".

which contain facts and propositions about the world inspired by first-order logic definitions. Transitions in the propositional space must be achieved through actions satisfying a precondition and transitioning the propositional configuration space into a postconditions. These conditions thus form subgoals out of more primitive actions in the general state space.

The grammar's propositional encoding includes information on objects, relations, and functions. For example, we may want to encode a mission where the objective is to eat an apple and a carrot. The initial world propositional space may be

$$in(apple, chest) \ \wedge \ in(carrot, house)$$

with the action of taking the apple given as:

$$in(apple, chest) \quad \wedge \quad by(agent, chest) \quad \rightarrow \quad in(apple, inventory).$$

Actions apply transitions via a context free grammar on the current propositional space. Goals are represented as a set of facts or propositions in the propositional space (e.g. $\forall food | in(food, belly)$). From the logical encoding, we construct facts, statements, and actions using a grammar derived from first-order logic as well as the themes of the quest.

The construct of the grammar includes facts, propositions, actions, and rules as defined by first-order logic constructs. [9] A fact defines a relation among elements that is true in the context. A proposition defines a relation among subsets that is true in the context. An action defines a transition between a set of precondition facts to a set of postcondition facts. A rule defines a similar transition from a set of precondition propositions to a set of postcondition propositions.

We define several built-in predicates.

- **in**—Whether entity A in the coordinate space is encapsulated by entity B, that is $Coord(A) \subset Coord(B)$ where $Coord()$ is the set of coordinates occupied by a given entity. This can be used, for example, to describe the state of the agent being inside of a building or other region that has been specifically marked out and given an identifier. Also used to express abstract containment, such as being in inventory.
- **at**—Whether entity A is in the coordinate space of another entity $(Coord(A) \cap Coord(B) \neq \varnothing)$.
- **by**—Whether entity A is in within a range to entity B for cases of interacting with an object (e.g. being in range to grab an apple off from a chest) $(\exists \delta < \epsilon, \exists a \in Coord(A) | a + \delta \in Coord(B))$.
- **unlocked**—A theme specific attribute relation for unlockable items.
- **inhand**—Whether an entity is currently at the top of the inventory and is usable in the world.
- **hasMaterials**—Whether an agent A has the materials in inventory to craft item X

AdventumRL uses special routines inspired by [8], that automatically track and compute the truth values of the predicates.

The grammar allows users to specify constructs using these relations in a grammar JSON file. In addition, we also allow users to specify physical entities in the Minecraft world through the quest file, which are then tracked with respect to the previous relations. Entity objects can be given identifiers that correspond to those in the relations. For example, we can identify specific key objects and specific doors. One can also assign identifiers to artificial, invisible bounding boxes in the map to indicate unique locations. There is no notion of specific locations or being inside or outside of structures native to Minecraft, which is just a grid of voxels. Thus we can place an invisible bounding box around a building in order to track when the *in* predicate with respect to this building is true.

Finally, we allow the user to specify *triggers*, which are facts or propositions that are made observable to the general state space. The user can choose what information to share with the agent that may be useful for decisions on low level tasks

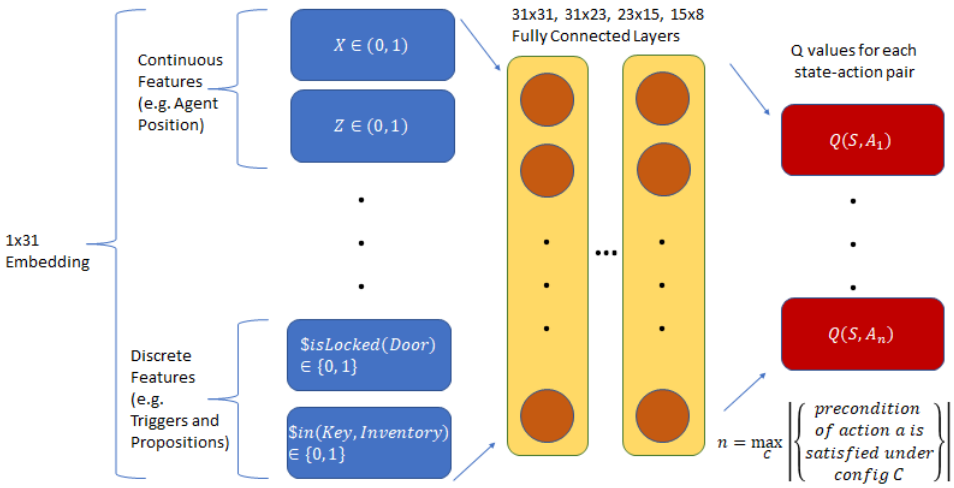

Figure 2: The architecture for the DQN embedding and model

such as navigation. For example, consider the scenario where a task requires you to ride a boat across a river. Since the navigation challenges of navigating a ship are different than walking, it would be useful to have a feature that distinguishes these forms of locomotion. In this case, the trigger would expose state information that the agent could use to determine whether it was walking or steering to facilitate training. However, one may also require the agent to learn the distinction entirely from visual cues, which are always available through the Malmo API—we support the ability to mix visual features and state features as the agent developer deems necessary.

3 Agent Models

For the purpose of this API, we include two premade agent models. These are the tabular (TabQ) agent and Deep Q-Learning (DQN) agent. The former utilizes a traditional q-learning table for discretized state-action spaces and tasks while the latter utilizes a generalized embedding.

Q-learning is an approach to reinforcement learning wherein the agent uses trial and error learn the function $Q(s, a)$, the estimate of expected future discounted rewards for taking action a in state s. Tabuluar Q-learning represents $Q(s, a)$ as a table of discrete states and discrete actions. Our TabQ agent operates in a discretized state space whereby each block the agent can stand on is an unique positional state. In addition, one dimension of the agent's state representation includes triggers for $locked(door)$, $by(player, door)$, $in(key, inventory)$, and $inhand(key)$ such that there are $n \times m$ unique states where n is the number of possible agent positions in the map and m is the number of triggers. The agent can move left, right, backward, and forward, and interact with items (picking up an item is the result of "interacting" with a pickup-able item such as the key). The API user can also decide whether the agent should receive a reward for each proposition change, including changes to triggers or actions on the configuration space. This allows the user to specify intermediate rewards for subtasks or configuration changes.

The second agent uses a Deep Q Network (DQN), which accepts positional and propositional information and approximates $Q(s, a_i)$ for each action a_i available from s. The default embedding for the DQN takes all trig-

gers and action preconditions into a one-hot embedding, along with continuous information like position. The default model includes 4 fully connected layers (31x31, 31x23, 23x15, 15x8) with LeakyRelu activations. The output is each action's corresponding Q value according to $\alpha_t[R(s) + \gamma \max_a Q(s_{t+1}, a)]$

4 Case Study

To demonstrate the AdventumRL framework, we constructed a simple quest requiring an agent to learn to obtain a key and unlock a door to escape a platform surrounded by lava. Figure 1 shows a screenshot; the key is a yellow block and the "door" is a region made of blue blocks. Picking up (i.e. adding to inventory) and putting down of the key is handled by the standard Minecraft game mechanics. The unlocking of the door is a new action defined as:

$$locked(door) \wedge by(player, door) \wedge in(key, inventory) \wedge inhand(key) \rightarrow \neg locked(door) \wedge by(player, door)$$

We include triggers for having the key in inventory and the locked-door state. This is to facilitate training by explosing these propositions so the agent can explore configurations as it handles more primitive navigation and exploration within the room.

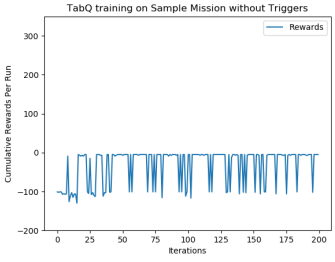

Figure 3: Cumulative rewards for a discrete q-learner attempting the locked room quest

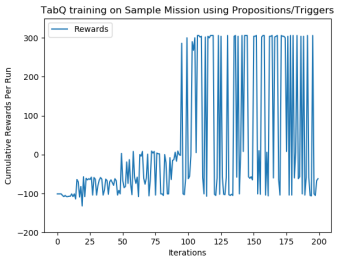

Figure 4: Cumulative reward using our proposition and triggering system for the same task

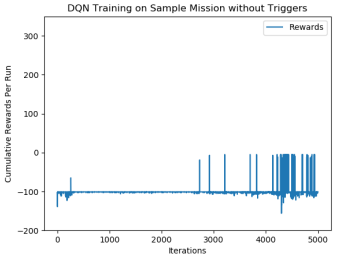

Figure 5: DQN cumulative rewards per run for the locked room mission

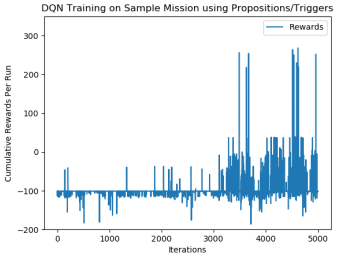

Figure 6: DQN cumulative rewards per run using the proposition and triggering system in the same task

For the regular TabQ agent, the agent is easily caught in a local minima after 25 iterations and finds difficulty exploring the exploding complexity of the state space. In the context of this mission, the agent is able to find a "key" entity, but is unable to explore actions that exploit this entity in order to change the underlying state space. This exploration problem is a common issue in reinforcement learning problems, especially in a questing scenario, which is the impetus for our system.

For the AdventumRL TabQ agent, the model taskes around 100 iterations to converge to π^*. Each average reward increment at 50 and 100 iterations marks indicate a change in the trigger state representation.

The regular DQN agent runs into the same problem as the regular TabQ agent in that the exploration problem is still poignant. The policy stabilizes at around 4500 iterations towards an unoptimal π.

For the AdventumRL DQN agent, the model takes around 3500 iterations to converge to a winning solution. However, it should be noted that the training process for DQNs are in general more unstable than tabular counterparts.

5 Conclusions

In this paper, we described a novel API for the construction of questing-based reinforcement learning. We gave a set of value propositions and use cases for this API, and defined a structure for learning high-level constructs. We provided a set of proof-of-concept models and gave an interface for building quests and agents. We then proposed scenarios for novel exploration into the reinforcement learning space.

References

[1] Stephan Alaniz. Deep reinforcement learning with model learning and monte carlo tree search in minecraft. *CoRR*, abs/1803.08456, 2018.

[2] Marc Brittain and Peng Wei. Hierarchical reinforcement learning with deep nested agents. *CoRR*, abs/1805.07008, 2018.

[3] Marc-Alexandre Côté, Ákos Kádár, Xingdi Yuan, Ben Kybartas, Tavian Barnes, Emery Fine, James Moore, Matthew J. Hausknecht, Layla El Asri, Mahmoud Adada, Wendy Tay, and Adam Trischler. Textworld: A learning environment for text-based games. *CoRR*, abs/1806.11532, 2018.

[4] Spencer Frazier and Mark Riedl. Improving deep reinforcement learning in minecraft with action advice, 2019.

[5] Matthew Johnson, Katja Hofmann, Tim Hutton, and David Bignell. The malmo platform for artificial intelligence experimentation. In *Proceedings of the Twenty-Fifth International Joint Conference on Artificial Intelligence*, IJCAI'16, pages 4246–4247. AAAI Press, 2016.

[6] Junhyuk Oh, Valliappa Chockalingam, Satinder P. Singh, and Honglak Lee. Control of memory, active perception, and action in minecraft. *CoRR*, abs/1605.09128, 2016.

[7] Diego Pérez-Liébana, Katja Hofmann, Sharada Prasanna Mohanty, Noburu Kuno, André Kramer, Sam Devlin, Raluca D. Gaina, and Daniel Ionita. The multi-agent reinforcement learning in malmö (marlö) competition. *CoRR*, abs/1901.08129, 2019.

[8] Mark Branly Arnav Jhala R.J. Martin R. Michael Young, Mark O. Riedl and C.J. Saretto. An architecture for integrating plan-based behavior generation with interactive game environments. *Journal of Game Development*, 2004.

[9] Stuart Russell and Peter Norvig. *Artificial Intelligence: A Modern Approach*. Prentice Hall Press, Upper Saddle River, NJ, USA, 3rd edition, 2009.

[10] Tim Salimans and Richard Chen. Learning montezuma's revenge from a single demonstration. *CoRR*, abs/1812.03381, 2018.