HumbleRL - Straightforward reinforcement learning Python framework

Grzegorz Beringer*, Mateusz Jabłoński*, Piotr Januszewski* and Karol Draszawka†
Faculty of Electronics, Telecommunications and Informatics
Gdańsk University of Technology

***Abstract*—We propose a framework called HumbleRL that unifies code structure of even very different Reinforcement Learning algorithms and allows for mixing them and experimenting with loosely coupled components such as: preprocessing, agent, environment, etc. We show that other solutions, despite their advantages in other applications, don't allow for this level of flexibility. We also provide readers with practical use cases and code taken from our work on Deep Reinforcement Learning.**

***Index Terms*—**

I. Introduction

Reinforcement learning (RL) is learning what to do—how to map situations to actions—so as to maximize a numerical reward signal. [1] This mapping is called a policy. RL consists of an agent that, in order to learn a good policy, acts in an environment. The environment provides a response to each agent's action that is interpreted and fed back to the agent. Reward is used as a reinforcing signal and state is used to condition agent's decisions. See fig. 1 for visual explanation.

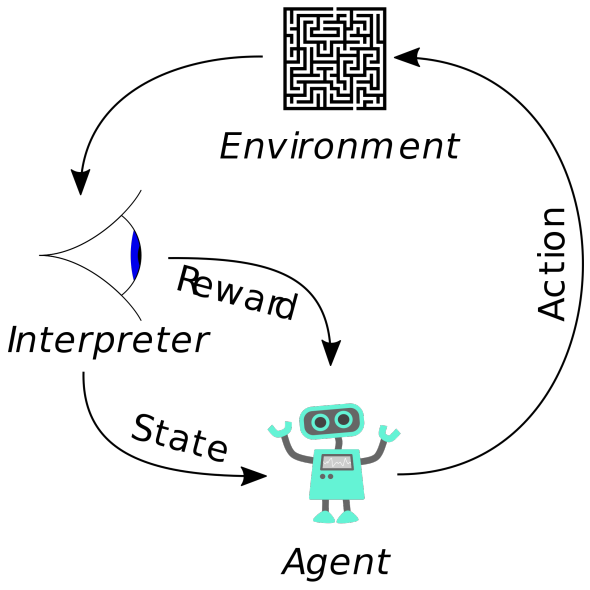

Fig. 1. Reinforcement Learning [2]

Each action-response-interpretation sequence is called a step. Multiple steps form an episode. The episode finishes in a terminal state and the environment is reset in order to start the next episode from scratch. Very often, RL agents need dozens and dozens of episodes to gather enough experience to learn the (near) optimal policy. For simplicity's sake, we call many consecutive episodes of learning a loop.

RL research scientists tend to write the entire RL code by themselves, instead of using existing frameworks. This is justified by the fact, that the commonly available frameworks are not flexible enough for intended experiments or require a specific backend.

HumbleRL was created with this problem in mind. Its simple API allows to perform a variety of RL experiments without any restrictions on the algorithms used. Since the back-end is not tied to any specific technology, it is possible to mix different neural network frameworks or not use any at all - HumbleRL provides the boilerplate code and determines the common interface, the rest is done by the user.

In this paper, we compare HumbleRL to other available solutions and explain the architecture of our framework. In the end, we present some use cases of HumbleRL, which showcase its simplicity and flexibility for RL experiments.

II. Related work

Currently, there are some libraries that facilitate research on deep reinforcement learning. We will review some of them.

A. TensorForce

One of the most popular libraries is TensorForce [3]. The project focuses on a readable and modular API to deploy reinforcement learning solutions both in research and practice. TensorForce is built on top on TensorFlow [4]. All low-level operations related to computation graph are encapsuled in Model component which allows the end user to forget about the intricacies of Tensorflow's low-level programming. The main disadvantage of the library is the lack of callback functions. There can be defined only one callback function which will be called at the end of episode.

B. RLlib

RLlib [5] is a part of bigger project named Ray. RLlib is focused on distributed reinforcement learning. It provides out-of-the-box solutions to run state-of-the-art RL experiments, both in parallel and distributed way, e.g. by using a computational cluster. The difference to other libraries is that there is easy-to-use JSON config which defines all expertiment's

*These authors contributed equally to this work.
†Research advisor.

TABLE I
RL FRAMEWORKS OVERVIEW

Framework	TensorForce	RLlib	KerasRL	Garage	HumbleRL
Available callback functions	only one	many	many	none	many
Available backends	TensorFlow	TensorFlow, PyTorch	Keras, TensorFlow	TensorFlow	Any (provided by user)
Parallel training	Available	Available	Unavailable	Available	Available
Distributed training	Available	Available	Unavailable	Unavailable	Unavailable
Ready solutions	Built-in	Built-in	Built-in	Built-in	Not ready yet
Training coordination	Built-in	Built-in	Built-in	Provided by user	Built-in

properties, e.g. environment name, size of batches, CPU/GPU support. It is also possible to define some callback functions at the start/end of episode/step/sample. Since RLlib is a part of a bigger project, it has some additional advantages, like compatibility with hyperparameter tuning tools (Tune). The main advantage of our solution to RLlib is that RLlib has limited Model class. It's impossible to create model with many heads in a simple way.

C. KerasRL

KerasRL [6] implements some state-of-the-art deep reinforcement learning algorithms and seamlessly integrates with the deep learning library Keras [7]. It focuses on providing out-of-the-box solutions with Keras API but it was not designed for performing complex RL research experiments. What distinguishes this library from others is a very simple API and support for multiple callbacks, also compatible with Keras callback functions. The main disadvantage of the library is that it is integrated only with Keras while HumbleRL has no dependency on the backend.

D. Garage

Garage is based on a predecessor project called rllab [8]. It is a framework for developing and evaluating reinforcement learning algorithms. It includes a wide range of continuous control tasks and implementations of many algorithms. Garage also provides functionality to perform distributed experiments using TensorFlow. The main difference between Garage and HumbleRL is that Garage doesn't coordinate training process - logic of single worker's experiment, especially communication between components, has to be implemented in user-code. It makes it harder to reuse the code between experiments.

Table I shows the comparison of the discussed frameworks and HumbleRL.

III. ARCHITECTURE

We show framework architecture in Fig. 2. An agent is represented by the *Mind* class. *Mind* encapsulates action planning logic and provides it via the *plan* method. In order to learn, the agent acts in the world represented by the *Environment* class. The *Environment* class provides methods for resetting, taking steps, rendering and getting information about the world. The agent isn't usually presented with raw environment observations - instead, it looks at states preprocessed by the

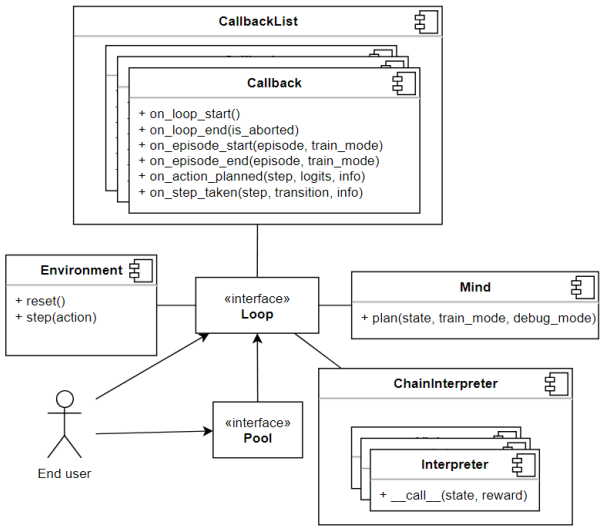

Fig. 2. HumbleRL architecture

Interpreter. Different interpreters can be joined together with the *ChainInterpreter* class. It acts as a preprocessing pipeline, with each subsequent interpreter using the output of a previous one as an input.

Framework user doesn't need to call all of those methods directly, those are utilized by the *loop* function. This function gets an action from the *Mind*, executes it in the *Environment* and then next observation is preprocessed with the *Interpreter* in preparation for the next step. To extend basic *loop* functionality, user can define callbacks that implement the *Callback* interface. Callbacks can react to events:

- at the beginning and ending of the *loop*,
- at the beginning and ending of each episode,
- after action is planned by the *Mind*,
- after step is taken in the *Environment*.

Callbacks are accumulated in the *CallbackList*. The entire *loop* function logic is shown in Fig. 3.

Parallel version of *loop* function is available as the *pool* function. It uses predefined number of workers to execute a pool of *Minds* in their own *Environments* in parallel.

IV. USE CASES

In this section, we present three different use cases of our framework to: a basic TD-learning algorithm, a state-of-the-art model-based planning algorithm and an evolution strategy

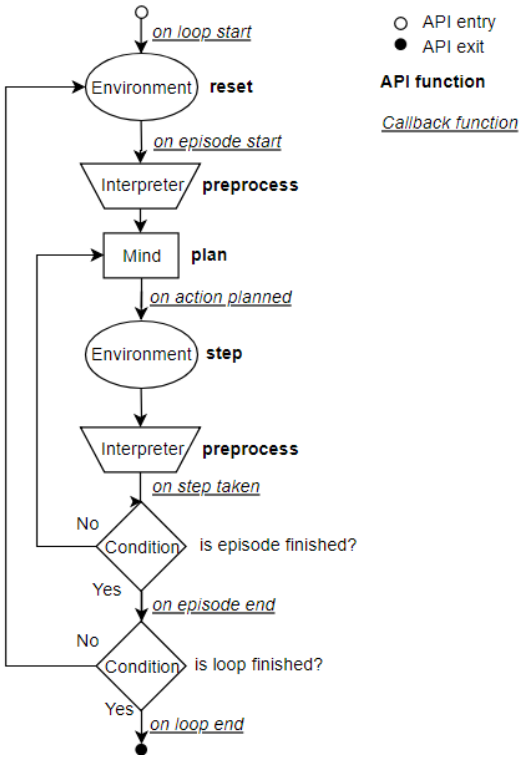

Fig. 3. HumbleRL loop function overview

algorithm that utilizes environment representation learned in unsupervised fashion. What's important, these methods differ substantially in the way they solve RL problems. We show how HumbleRL unifies their implementation structure and therefore allows us to easily mix them together in order to experiment with new methods.

A. Tabular Q-Learning

In this experiment, we show basic application of HumbleRL to a classic RL algorithm. HumbleRL user have to implement only a crucial code of the algorithm and can focus more on the algorithm logic. All the run time is provided for him by the framework.

1) Algorithm: Tabular Q-Learning is a model-free RL algorithm that keeps state-action pairs quality values in a table. Values are learned using temporal-difference learning. Supplied with optimal quality values, we are able to derive the optimal policy by greedily choosing the best action in every state [1].

2) Environment: We used the OpenAI Gym [9] "Frozen Lake" environment in our experiment with Tabular Q-Learning. The agent's objective is to get from the starting position to the target position without sliding into one of the holes with cold water. The agent moves in four directions (North, East, South and West) in a grid that represents ice on a frozen lake. There is some probability that agent's action will fail due to slippery ice. In that case, a random action is performed. The environment's state is the agent's current position in the grid represented by an integer number associated with each grid field. The environment finishes with a negative reward when the agent ends up in a hole and with a positive reward when the agent reaches the target position.

TABLE II
ADAPTATION OF TABULAR Q-LEARNING IN HUMBLERL

HumbleRL's feature	Tabular Q-learning's use case
Environment	OpenAI Gym environment
Mind	Quality value table
Interpreter	Not used
Callbacks	Updating policy

3) Implementation: Our framework provides the factory of OpenAI Gym environments and we use it to create the "Frozen Lake" *Environment* object. Q-Learning policy that chooses the best action according to current state-action quality values is implemented in the *Mind* class. During training, it adds a random noise to quality values to boost exploration that is essential for Q-learning algorithm convergence. The *Interpreter* isn't needed as the environment raw state is used as the quality values table index.

We use a callback to update quality values after each step. *On step taken* callback provides current experience (the current state, the action taken, the reward obtained and the next state) which is used to update the current state-action quality value. We also use *on step taken* and *on episode end* callbacks to track the running average of agent's success rate.

With those components prepared, we run *loop* function to start training and observe progress bar and success rate metric. You can find the code in HumbleRL GitHub repository [10]. Adaptation of Tabular Q-learning in HumbleRL is shown in table II.

B. AlphaZero

This use case shows how HumbleRL can ease implementation of search algorithms for perfect information environments with two adversarial agents.

1) Algorithm: AlphaZero [11] is the state-of-the-art algorithm for playing Shogi, Chess and Go [12] board games. It extends Monte-Carlo Tree Search (MCTS) with a deep neural network (DNN) used for states evaluation and simulations guiding. AlphaZero plays with itself to gather experience stored in a buffer. Its own experience is then used to train the DNN by a novel reinforcement learning algorithm.

2) Environment: Because of a lack of compute power, we used shallower DNN than the original one and chose simpler board games for our experiments: Connect Four and Othello. We successfully trained our AlphaZero implementation to play both games on super-human level. A state is a board configuration and an index of a player that'll take action in this turn. Actions and goals are specific for each game. Both games have a positive reward for a winning player and a negative reward for a losing player. In case of a draw both players get a zero reward.

TABLE III
ADAPTATION OF ALPHAZERO IN HUMBLERL

HumbleRL's feature	AlphaZero's use case
Environment	MDP representation
Mind	MCTS for one player
Interpreter	State normalization
Callbacks	Gathering experience data and score statistics

3) Implementation: We took already implemented board games [13] and wrapped each in the *MDP* class to represent them as Markov Decision Processes [14] (MDP). MDP is used for MCTS simulations, but also as the *Environment*. The *Mind* class implements MCTS algorithm for one player. In AlphaZero, it uses the DNN to evaluate each node and guide search direction on each edge. We compose two *Minds* in *AdversarialMind* class that choose which player should take action based on a current state. In the *Interpreter* class, we take care of states normalization. Each player always sees the board as if he's playing white. This way, we don't have to condition the DNN on a player's pawns' color.

We gather self-play experience and players' score statistics using callbacks. The DNN training phase takes place after a predefined number of self-play games. The training phase is performed using popular Keras [7] framework, that is specialised for neural networks training. HumbleRL is designed not to interfere with other frameworks e.g. for DNN training. Next, the self-play phase takes place once again and the two further interchange. The self-play phase uses the *loop* function to effortlessly clash AlphaZero with itself for given number of games (episodes). Adaptation of AlphaZero algorithm in HumbleRL is shown in table III.

C. World Models

1) Algorithm: World Models [15] is based on generative models that learn a compressed spatial and temporal representation in unsupervised manner. Environment's observation features extracted from these models are then used to train the final model for solving complex environments.

The solution consists of 3 components: Vision for encoding spatial information (not to be confused with HumbleRL's *Interpreter*), Memory for encoding temporal information and Controller, which chooses the actions to take in the environment and is trained with evolution strategy algorithm.

2) Environment: We used OpenAI Gym's CarRacing environment (the same as in original World Models paper), which is a continuous-control, top-down racing environment. The agent receives state pixels as input and controls the break, accelerator and the steering wheel of a car.

3) Implementation: We use HumbleRL in a few stages of training the model. First, we use *loop* together with an agent performing random actions (*Mind*) and a *Callback*, that gathers transitions and saves them to an external storage (HDF5 file in our case). The framework allows us to focus strictly on collecting trajectories and not worry about agent-environment interactions.

TABLE IV
ADAPTATION OF WORLD MODELS IN HUMBLERL

HumbleRL's feature	World Models's use case
Environment	OpenAI Gym environment
ChainInterpreter	Vision and Memory components
Mind	Controller component
Callbacks	Gathering training data
Pool	Evolutional strategy during Controller's training

Transitions are used to train the Vision and Memory components. For Vision, we use Keras and for Memory we use PyTorch [16], since we found it easier to use for this case than Keras. HumbleRL is not constricted to work with any particular deep learning library, so it's not a problem to mix the solutions, as long as we wrap our trained models in proper interfaces.

Once we have these components ready, we use them to train the final piece - the Controller. The great advantage of using an evolution strategy algorithm is that we can easily parallelize the computations related to evaluation of a population of agents, each using Vision and Memory as *ChainInterpreter* and a linear layer of neurons as *Mind*. Parallelization of this process is done using HumbleRL's *pool* method, which distributes the workload to user-defined number of workers, each running their own *loop*. At the end of an evolution epoch, we gather results from workers, use the reward returns to improve our population and then start *pool* again, now with a new batch of agents. Adaptation of World Models in HumbleRL is shown in table IV.

V. CONCLUSIONS

Our framework makes it very easy to implement basic RL algorithms in compact way. User can focus on algorithm logic and all the boilerplate code for environments creation, RL loop execution, etc. is provided for him.

REFERENCES

[1] R. S. Sutton and A. G. Barto, *Reinforcement Learning: An Introduction 2nd*. The MIT Press, 2018.

[2] Megajuice, "Reinforcement learning diagram," 2017, accessed on 28.12.2018. [Online]. Available: https://commons.wikimedia.org/wiki/File:Reinforcement_learning_diagram.svg

[3] M. Schaarschmidt, A. Kuhnle, and K. Fricke, "TensorForce: A TensorFlow library for applied reinforcement learning," Web page, 2017. [Online]. Available: https://github.com/reinforceio/tensorforce

[4] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. S. Corrado, A. Davis, J. Dean, M. Devin, S. Ghemawat, I. Goodfellow, A. Harp, G. Irving, M. Isard, Y. Jia, R. Jozefowicz, L. Kaiser, M. Kudlur, J. Levenberg, D. Mané, R. Monga, S. Moore, D. Murray, C. Olah, M. Schuster, J. Shlens, B. Steiner, I. Sutskever, K. Talwar, P. Tucker, V. Vanhoucke, V. Vasudevan, F. Viégas, O. Vinyals, P. Warden, M. Wattenberg, M. Wicke, Y. Yu, and X. Zheng, "TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems," 2015, software available from tensorflow.org. [Online]. Available: http://tensorflow.org/

[5] E. Liang, R. Liaw, R. Nishihara, P. Moritz, R. Fox, K. Goldberg, J. E. Gonzalez, M. I. Jordan, and I. Stoica, "RLlib: Abstractions for Distributed Reinforcement Learning," 2018.

[6] M. Plappert, "keras-rl," https://github.com/keras-rl/keras-rl, 2016.

[7] F. Chollet *et al.*, "Keras," https://keras.io, 2015.

[8] Y. Duan, X. Chen, R. Houthooft, J. Schulman, and P. Abbeel, "Benchmarking Deep Reinforcement Learning for Continuous Control," *arXiv e-prints*, 2016.

[9] G. Brockman, V. Cheung, L. Pettersson, J. Schneider, J. Schulman, J. Tang, and W. Zaremba, "OpenAI Gym," *arXiv e-prints*, 2016.
[10] P. Januszewski, G. Beringer, and M. Jablonski, "HumbleRL," 2018. [Online]. Available: https://github.com/piojanu/humblerl
[11] D. Silver, T. Hubert, J. Schrittwieser, I. Antonoglou, M. Lai, A. Guez, M. Lanctot, L. Sifre, D. Kumaran, T. Graepel, T. P. Lillicrap, K. Simonyan, and D. Hassabis, "Mastering Chess and Shogi by Self-Play with a General Reinforcement Learning Algorithm," *arXiv e-prints*, 2017.
[12] D. Silver, J. Schrittwieser, K. Simonyan, I. Antonoglou, A. Huang, A. Guez, T. Hubert, L. Baker, M. Lai, A. Bolton, Y. Chen, T. Lillicrap, F. Hui, L. Sifre, G. van den Driessche, T. Graepel, and D. Hassabis, "Mastering the game of Go without human knowledge," *Nature*, vol. 550, 2017.
[13] S. Thakoor, S. Nair, and M. Jhunjhunwala, "Alpha Zero General (any game, any framework!)," https://github.com/suragnair/alpha-zero-general, 2017.
[14] Wikipedia contributors, "Markov decision process," 2018, accessed on 30.12.2018. [Online]. Available: https://en.wikipedia.org/wiki/Markov_decision_process
[15] D. Ha and J. Schmidhuber, "World Models," *CoRR*, vol. abs/1803.10122, 2018. [Online]. Available: http://arxiv.org/abs/1803.10122
[16] A. Paszke, S. Gross, S. Chintala, G. Chanan, E. Yang, Z. DeVito, Z. Lin, A. Desmaison, L. Antiga, and A. Lerer, "Automatic differentiation in PyTorch," in *NIPS-W*, 2017.