Learning How to Play Bomberman with Deep Reinforcement and Imitation Learning

Chapter · November 2019

DOI: 10.1007/978-3-030-34644-7_10

3 authors:

Ícaro Goulart Faria Motta França
Universidade Federal Fluminense
1 PUBLICATION **0** CITATIONS
SEE PROFILE

Aline Paes
Universidade Federal Fluminense
56 PUBLICATIONS **129** CITATIONS
SEE PROFILE

Esteban Clua
Universidade Federal Fluminense
251 PUBLICATIONS **1,106** CITATIONS
SEE PROFILE

Learning how to play Bomberman with Deep Reinforcement and Imitation Learning *

Ícaro Goulart, Aline Paes, and Esteban Clua

Institute of Computing, Universidade Federal Fluminense
Niterói, RJ, Brazil.
igoulart@id.uff.br {alinepaes,esteban}@ic.uff.br

Abstract. Making artificial agents that learn how to play is a long-standing goal in the area of Game AI. Recently, several successful cases have emerged driven by Reinforcement Learning (RL) and neural network-based approaches. However, in most of the cases, the results have been achieved by training directly from pixel frames with valuable computational resources. In this paper, we devise agents that learn how to play the popular game of Bomberman by relying on state representations and RL-based algorithms without looking at the pixel level. To that, we designed five vector-based state representations and implemented Bomberman on the top of the Unity game engine through the ML-agents toolkit. We enhance the ML-agents algorithms by developing an Imitation-based learner (IL) that improves its model with the Actor-Critic Proximal-Policy Optimization (PPO) method. We compared this approach with a PPO-only learner that uses either a Multi-Layer Perceptron or a Long-Short Term-Memory network (LSTM). We conducted several pieces of training and tournament experiments by making the agents play against each other. The hybrid state representation and our IL followed by PPO learning algorithm achieve the best overall quantitative results, and we also observed that their agents learn a correct Bomberman behavior.

Keywords: bomberman, proximal policy optimization, reinforcement learning, lstm, imitation learning

1 Introduction

Building games with agents that *learn* how to play is a long-standing goal in Game-AI. So far, this has been mainly addressed by Reinforcement Learning (RL) algorithms [1] and, more recently, by combining RL with deep neural networks in the area of Deep Reinforcement Learning (DRL) [2–4]. However, the most remarkable results achieved so far have counted with valuable computational resources to deal with large pixel level-based search spaces.

* We would like to thank the Brazilian research agencies CAPES and CNPq, and NVidia.

In this work we take a step in another direction: we focus on the game of Bomberman represented in a grid scenario. Bomberman is a popular and universally pleasurable maze-based strategy video game that requires intelligence. Although at first sight it looks simple, designing an intelligent Bomberman agent faces a number of challenges: a vast search space due to numerous components possibilities (other agents, bombs, blocks, *etc*), multiplayer mode, strategic reasoning, delayed reward due to the time that the bombs take to explode, and a dynamic environment [5].

Most of the available work addressing Bomberman *learning*-agents has focused on table-based RL without exploring the more general function approximation power of neural networks [6, 7]. Only recently, Komerlink *et al.* [8] have used Q-learning coupled with a multi-layer perceptron neural network (MLP), focusing on comparing exploration strategies. Here, we tackle the complexity of the Bomberman game with a combination of Imitation Learning (IL) [2] and the recently developed Actor-Critic Proximal Policy Optimization (PPO) strategy [9], developed within the area of DRL. In IL, an apprentice agent learns to perform a task observing an expert. Arguably, this strategy produces a good starting point for the PPO algorithm. We devised the Bomberman game using the Unity game engine and implemented the combined approach on the top of the ML-Agents toolkit [10], an open-source project to create simulation environments based on ML and using the Unity Editor. Also, we implemented five vector-based state representation, namely: 1) Hybrid One-hot, 2)ZeroOrOne, 3)Binary Flag, 4)Normalized Binary Flag e 5)ICAART.

The Bomberman game we developed holds the capability of representing cells with more than one element, allowing for more than one agent to be in the same cell, as in the original game. Additionaly, the only information the agents have from the game is the observation of the environment, in the form of the vector-based state representations and the rewards.

We conducted a series of training sequences and tournament among the agents to compare (i.) the five ways of representing the space state, (ii.) PPO trained with MLP or Long-Short-Term memory [11] (LSTM), and (iii.)the Behavioral Cloning (BC) IL algorithm followed by PPO compared to training these algorithms singly from each other. The results have pointed out that by using the hybrid state representation and aggregating the experiences of several learners in a PPO+LSTM at once is the best general way of reaching an effective Bomberman agent. Our solution is publicly available [1] [2], allowing for further improvement and usage.

2 Background

Bomberman[3] is a maze-based strategy game franchise, developed by Hudson Soft Company in 1985. The game is based on placing bombs strategically within

[1] `https://github.com/lorel-uff/pip`
[2] `https://github.com/icaro56/ml-agents`
[3] `https://en.wikipedia.org/wiki/Bomberman`

the scene, to kill the enemies, and to destroy blocks in the scenario, aiming at opening paths or finding items. In a multiplayer mode, the player's goal is to be the last player alive by killing all its opponents. The explosion of a bomb propagates vertically and horizontally to the neighbor cells, respecting possible obstructions along the way.

Here, we address the Bomberman game with RL algorithms. RL aims at teaching agents by trial and error, mapping states into actions through maximizing a numeric reward. The reward can be positive, when the chosen action leads to the goal, or negative, in the opposite case [1]. To define the interaction between the learner agent and its environment, an RL problem is specified as a Markov Decision Problem (MDP) with a space state S, a set of actions A, a transition function $Tr(s, a) \rightarrow s'$, a reward function $R(s, a) \rightarrow \mathbb{R}$, and a discount factor γ. The goal is to find a policy function that maps states to actions.

An RL problem can be modeled as a regression function by mapping the space of states and actions to the reward [12]. The goal becomes approximating such a function using, for example, a neural network [13]. Neural networks can focus on optimizing the policies directly or on learning the value functions from them to infer policies. When the underlying neural network used in an RL problem is deep, we have a Deep Reinforcement Learning (DRL) algorithm [4].

In this work, we rely on a DRL actor-critic method, using either an MLP or an LSTM network and on Imitation Learning. We briefly describe them as follows.

Proximal Policy Optimization (PPO) is a DRL policy optimization method that uses a stochastic ascent gradient to update the policy function. PPO has the stability and reliability of trust-region methods [14], it is a model-free, on-policy RL algorithm, it can deal with a continuous space of observation and actions, and it relies on the Advantage operator instead of Q-values. It is based on the *actor-critic* framework that, roughly speaking, encompasses two networks, one to act as an estimator of the function value (the critic) and the other to determine the policy itself (the actor) [15]. Accurately, PPO follows the actor-critic A3C [16] technique allowing that multiple asynchronous agents can be trained simultaneously in a global supervisor network.

Long Short Term Memory (LSTM) is a recurrent network (RNN) designed to avoid the problem of long-term dependency [11] encountered by RNNs. The input information that enters an LSTM network may undergo some small linear interactions or move on to the subsequent iterations. LSTM also can add or remove cell status information through gates, composed of Sigmoid layers and multiplication operations.

Imitation Learning RL algorithms follow a series of interactions with the environment to achieve the goal, which can make the learning process very slow. To speed up the training time, IL techniques learn from expert demonstrations. Here, we use the Behavioral Cloning (BC), a supervised imitation learning method that maps state/action pairs from expert trajectories to policies without learning the reward function [2]. The BC method works as follows: i) observe the current state s and the action chosen a by the expert; ii) choose

an action a' based on the current policy P (initially this policy is selected at random); iii) compare the chosen action a' with the expert demonstrated action a using an error function; iv) optimize the policy P using the error function, yielding a policy P'; v) repeats the whole process, now using P' as the current policy.

3 Related Work

Previous work has also focused on Bomberman to implement and experiment with ML techniques. Bomberman as an Artificial Intelligence Platform (BAIP) [6] is a Bomberman-based graphical, open-source, agnostic language platform, able to assist in the study and development of new AI techniques. It includes agents based on heuristics, searching, planning, and RL methods. Agents that use RL techniques were not able to play the same version of the game as heuristic and search agents.

In [7] a Reactive, MiniMax, and Q-Learning agents were compared in a discrete-style Bomberman game. The MiniMax-based agent routinely defeats a human player, but it is computationally expensive and unable to run in real-time. The Reactive agent was able to defeat human players most of the time, but cannot beat the MiniMax agent. The best result achieved by their Q-Learning agent was in a static scenario without destructible blocks and with only one enemy standing still.

A Q-learning with MLP strategy was implemented in [8]. An Error-Driven-ϵ and Interval-Q exploration strategies were compared to five other techniques. As there, we also represent the state of the environment with a vector to feed a neural network. However, we also include other state representations and rely on more sophisticated learning techniques, trying to address the complexity of the Bomberman game.

Pommerman [5] is a multi-agent environment based on the game Bomberman, consisting of a set of scenarios, each one with at least four players and containing cooperative and competitive aspects. Also, it can be used to create methods such as planning, opponent/teammate modeling, game theory, and communication. A competition track at NIPS 2018[4] used Pommerman as a framework. The Pommerman scenarios always have 11 rows by 11 columns and, different from the original game, Pommerman agents cannot occupy the same cell. The environment developed in our work allows cells with more than one state, and agents can occupy the same cell as in the original game. Besides that, we developed using a commercial game engine with a large community of developers.

4 State Representation and Imitation-Reinforcement Learning in Bomberman

In this paper, we investigate the influence of five vector-based state representation and four learning algorithms to make an agent learn how to play Bomber-

[4] https://nips.cc/Conferences/2018/CompetitionTrack

man. The Bomberman game implemented here resembles the original gameplay. However, we focus on the agent behavior regarding the other agents, the scenario, and its wish to remain alive, disregarding power abilities and not allowing more than one bomb per agent. With that, the agent aims at being the last living agent on the scene to win the match. The game was developed with the Unity game engine, and the algorithms were built on the top of ML-Agents Toolkit.

4.1 Dynamic of the Game

The Bomberman scenario has nine rows and nine columns within a 2D grid board. The game computes the time according to the number of interactions between the agent and the environment. Each game may have two, three, or four players battling against each other. The entities in the game are (i.) indestructible blocks, (ii.) destructible blocks, (iii.) agents, (iv.) bombs, (v.) bomb's fires, and (vi.) danger zones. The danger zones are invisible entities that inform the extension of the bomb's fires when it explodes.

At the start of a match, the agents are randomly positioned in one of the four corners, each one in a different position. At each iteration, the agent receives the observation from the environment (detailed next) and takes one action. The agents can put one bomb at a time on the scenario in their current position. After N iterations that the bomb has been activated, it explodes with a range of two cells to each linear direction (up, down, left, and right). It is required to wait for the bomb explosion before putting another one. The fire of a bomb blast is configured to last only one iteration.

Some entities within the range of an explosion obstruct its effect beyond them. Thus, the bomb explosion does not pass through the blocks or other bombs. Thus, in the presence of such entities, the explosion of a bomb is limited to one cell in the direction of the block (or another bomb) if it is on the same side of the bomb. However, the fire of a bomb causes the explosion of other bombs inside the two-cells range.

A match ends when there is only one living agent remaining on the scenario, or when they are all dead (draw). The last existing agent in the scenario is the winner of the game. If we do not have a winner after 300 iterations, then we generate a rain of bombs to force the end of the game.

4.2 Bomberman training as a Markov Decision Process (MDP)

As usual in RL problems, we design the Bomberman problem as a finite and episodic MDP. The MDP problem is composed of (1) *the state set*: the states of all the grid cells in the scenario and the position (x, y) of the agent. The state is encoded as a vector to feed the agent's observation. The vector representation can vary according to the type of representation. (2) *Actions*: the agent can (i.) standstill, (ii.) go up, (iii.) go down, (iv.) go left, (v.) go right, all of these last four moving a single cell, and (vi.) put a bomb. (3) *Transitions*: the transition from one state to another guided by action is determined by the previous state and the action themselves, with no uncertainty caused by the environment. Thus, the

agent tries to execute an action and, if it is allowed, the environment perceives its effect. Other entities may also cause a transition from one state to another, such as a bomb that explodes. (4) *Rewards* The rewards are distributed according to Table 1. After performing one action, the agent receives a reward according to the new state of the game. One or more type of rewards can be applied combined in the same iteration. For example, the agent can kill an enemy *and* destroy a block, or the agent can only stay still and suffers the iteration penalty. Every time an agent gets closer to an enemy, it receives a reward for approaching an opponent (the fifth line in the table). For example, if the distance between the two agents in question is x cells, then the reward will only be given to the agent if he gets closer than that to the opponent, *i.e.*, at least $x - 1$ cells. Rewards for approximating and distancing agents are only given after a successful walking action when the observed state has a new position of the agent. The distances are computed as Manhattan distance.

Table 1. Rewards given to the Agent

State	Reward
The agent is dead	-0.5
An opponent is dead by the agent earning the reward	1.0
The agent is the last man alive	1.0
A block has been destroyed	0.1
The agent is in the closest position so far to an enemy	0.1
The agent is closer to an opponent than before	0.002
The agent is farther to an opponent than before	-0.002
Penalty per iteration	-0.01
The agent is in a cell within reach of a bomb	-0.000666
The agent is in a safe cell when there is a bomb nearby	0.002

4.3 State Representations

In RL algorithms, the choice of representation for the observed states may influence the agent's learning ability as, if the state is misrepresented, the agent will probably be unable to find patterns and perform coherent actions. Here, the observation of the current state is represented as a vector. The size of this vector depends on the type of the state representation: if we use only the numerical value to represent a cell, then the observation vector has a numerical value to represent each cell of the grid. If we use an array, then the observation vector has the number of elements in this array to represent each cell in the grid.

We devised four possible representations to the environment: Binary Flag, Normalized Binary Flag, Hybrid, ZeroOrOne. Furthermore, we implemented the state representation devised in [8] and named it as ICAART, yielding a total of five state representations. In all of these cases, every cell is first encoded as bit flags. Bit flags are used to store more than one Boolean value in a whole set of bytes, representing the existence of one or more objects of that state type in the cell. Thus, we can inform, for example, that an agent *and* a bomb are in the same grid cell at the same moment. In this case, if the bit flag representing the agent and the bomb are 01 and 10, respectively, then the bit flag representing

the grid cell is 11. Eight possible state types are represented by bit flags: empty cell, indestructible block, destructible block, current agent, enemy, bomb, fire, and danger. The five state representations are as follows:

1. Binary Flag In this case, for each cell in the grid, there is a bit flag to represent what is there, plus the position (x,y) of the agent, making the size of the observation vector as the size of the grid plus 2. After composing the observation vector with the bit flags, it is converted to decimal to provide the agent with this information finally.

2.Normalized Binary Flag This representation only normalizes the previous one to stay between maximum and minimum values. Values are in [0.0, 1.0] range.

3.Hybrid The hybrid representation combines the Binary Flag representation with a One-Hot Vector representation. A One-Hot Vector is a 1 x C matrix, where C is the number of possible situations, which consists of 0s in all dimensions except for a single 1 in a dimension used uniquely to identify the class or state type. To represent a cell that has an agent ([1,0]) and a bomb ([0,1]) with a One-Hot Vector, the vector dimension must be increased by one, generating a new One-Hot Vector that represents the agent plus the bomb ([0,0,1]). This feature makes One-Hot vectors very costly. Thus, we propose a hybrid representation that allows for using a One-Hot vector similar to the Binary Flag representation. For example, to represent bomb and agent entities occupying the same grid cell, we have the Hybrid vector [1,1] instead of a new vector [0,0,1]. Each cell in the grid is represented using a Hybrid vector. Thus, if there are 4 types of state (plus empty), then the size of each Hybrid vector will be 4 (e. g. [1,0,0,1]). Consequently, the size of the observation vector will be the number of cells multiplied by the size of the Hybrid Vector plus position x, y (e.g. $4 \times 4 + 2 = 18$, in a 2x2 grid). More accurately, a Hybrid vector is a matrix $1 \times H$, where H is the number of possible situations, which consists of numbers 0 in all dimensions, except for numbers 1 in each dimension when their respective state is active.

4. ICAART This is the representation used in [8]. For each cell in the grid, the number 1 is used to represent an empty cell, 0 if it is destructible or -1 if it is obstructed (indestructible or bomb). After this step, we have a vector of the same number of cells that we have in the grid. Next, for each cell in the grid, the number 1 is included if the agent is in that position or 0, otherwise. Then, we add 1 to the cell if there is an opponent there, or 0, otherwise. Finally, for each cell in the grid, a float value between -1.0 and 1.0 is included to represent the level of danger of a cell, computed according to the time that the bomb has left to explode (less time to explode means more dangerous). The danger value is negative if the bomb was placed by the agent and positive if it was placed by an opponent or by the environment itself. In this representation, the size of the observation vector is four times larger than the size of the grid.

5.ZeroOrOne This representation closely resembles the Hybrid representation, but, instead of adding one hybrid vector to each grid cell, each observation is added to the observation vector separately, as in the ICAART representation. It increases the size of the agent observation vector by seven times the grid size. The order of addition, considering the value of 1 or 0, respectively, is as follows:

(1), free or obstructed cell; (2) destructible or indestructible; (3) the cell contains an agent or not; (4) the cell has an opponent or not; (5) the cell has a bomb or not; (6) the cell is dangerous or not; (7) the cell is on fire or not.

4.4 Reinforcement Learning Algorithms

We include four RL algorithms to train Bomberman agents, as follows: (i) the PPO [9] method integrated with a Multi-Layer Perceptron (MLP); (ii)the PPO [9] method integrated with a Long Short-Term Memory (LSTM) network; (iii) the Behaviour Cloning (BC) [2] imitation learning method; and (iv) a novel implementation that runs the PPO after learning with BC, *i.e.,* first, we run the BC, save its model, and start PPO with such a model.

The PPO algorithm approximates the policy function using a neural network. To that, the implementation provided by the ML Agents toolkit allows for using either an MLP or an LSTM as such a neural network. In this last case, we benefit from the memory component of the LSTM to allow for the agent to remember its relevant past experiences. The method keeps saving the iterations of the agent with the environment together with the received rewards to compose the batch of examples. More specifically, during each iteration of the game, the method accumulates the experience of each agent composed of the current state, the executed action, the received reward, and the next state. After a certain number of collected experiences, the network (MLP or LSTM) do its job, by executing the value and the advantage function, computing the errors, and updating the policy.

To train the agent using the BC algorithm, we provide a replay file with the expert interactions. The number of opponents in each match may change according to the replay file. The match in which the expert agent has played is exactly reproduced in the replay file so that the expert can demonstrate to the student how to play the game. When the student is learning, all the opponents' moves and matches are synchronized with the replay file. In this case, its opponents react according to the replay, but the apprentices have made their own decisions. After it dies or the number of iterations per match reaches a maximum limit, the next match in the file is loaded.

We also developed a novel combination of the BC and PPO training, by first training with the BC approach, followed by training with PPO. We save the model trained with BC and load this model to initialize the weights of the PPO network. We use a function that forces the network to load even though they are slightly different. All the nodes in the neural network that have the same name are loaded with the values from the previous training, while all nodes that were not present in the last training are initialized with their default values.

4.5 Training Process Loop

The entire training process is composed of several matches, corresponding to the episodes of a traditional RL algorithm. At the beginning of each match, the agents are created and randomly positioned in the grid. Since there is more than

one agent per scenario, it is necessary to synchronize their observations, actions, and rewards. In this way, one agent does not have advantage over another one due to getting some information first. At the beginning of each iteration in a match, the state of the bombs is updated, and, if the time has come, they explode. It is also necessary to update the blocks since some of them may have been hit by an explosion. Next, each agent receives the state of the environment and the reward corresponding to the observed state. The learning algorithm also receives state observation and the given rewards to update the policy function. Finally, the living agents act in the environment, according to the action computed by the current policy function. At the end of an iteration, we apply the time penalty reward.

5 Experimental Results

In this section, we present experiments and results related to our proposal and conducted to investigate how Bomberman agents learn under (1) the five state representation, (2) MLP versus LSTM, and (3) the learning algorithms.
General Experimental Setting: At the beginning of a match, we create at random 2 to 4 agents per scenario. To train the agents, we created and configured ten scenarios that run in parallel in the same 2D environment, simulating different episodes of the RL algorithm. Among them, five are static, meaning that their destructible blocks are recreated in the same configuration at the beginning of the episode, and the other five are configured at random. Table 2 shows the learning algorithms' hyperparameters. After the training phase, we conducted tournaments with the agents composed of a 100 matches (T100) whenever we want to compare agents learning within the same environment and tournaments with 1000 matches (T1000) to compare distinct agents.

To select the most appropriate way of representing states, we train PPO with only one ML-Agents toolkit's brain learning from the experience of all the agents in the scenario. We repeat the training five times for 2M iterations for each one of the five types of state representation (Binary Flag, Normalized Binary Flag, Hybrid, ICAART, and ZeroOrOne). Thus, we have a set of *homogeneous_brain* = {***BF***, ***NBF***, ***H***, ***I***, ***ZO***}, with each element in the set corresponding to one of the five state representations. Each element in *homogeneous_agents* set unfolds into five training sets, *i.e.*, $\boldsymbol{BF} = \{BF_1, \ldots, BF_5\}$, $\boldsymbol{NBF} = \{NBF_1, \ldots, NBF_5\}$, where the first element corresponds to a whole training execution, the second element corresponds to another independent training execution, and so on. Thus, we have a total of 25 training executions, taking, on average, 20 hours of training. The larger is the observation vector, the more time the training takes to finish.

Fig. 1 shows that the cumulative rewards of the Binary Flag and Normalized Binary Flag representations are lower than the other for almost the entire training phase, achieving the average performance of the others only over the end of the training. Meanwhile, the other agents keep all taking the lead until the step 651K, when the ICAART representation gets the best average cumulative reward and surpasses the rest of them until the end of the training. The average

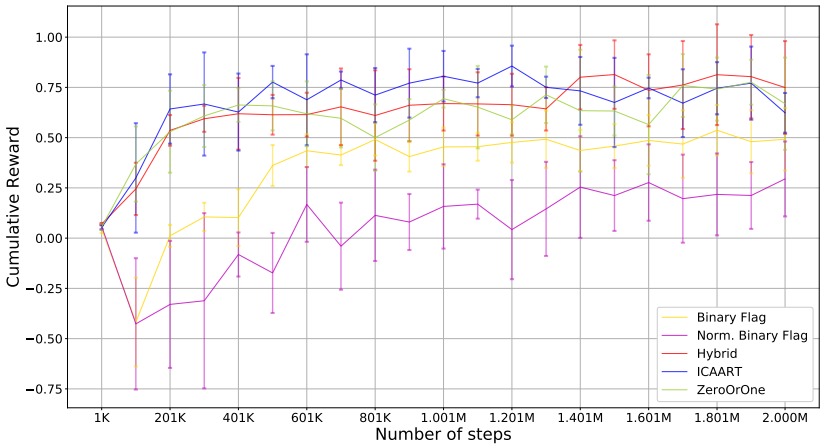

Fig. 1. Cumulative rewards in the Bomberman training sessions with the 5 types of state representation. Vertical lines stand for standard deviation.

cumulative reward of both ZeroOrOne and Hybrid Brains were similar. Note that standard deviation behaves similarly for all Brains.

The behavior of the cumulative rewards of the PPO with LSTM and the BC agents are very similar to the one presented in Fig. 1. Thus, due to lack of space, we do not show them here.

Table 2. Hyperparameters used in the learning process

Name	Value	Name	Value	Name	Value
batch size	128	# hidden layers (MLP)	3	# hidden units (MLP)	128
learning rate (MLP)	0,0003	time horizon	128	beta	0,005
buffer size (MLP)	2048	epsilon	0,2	gamma	0,99
lambd	0,95	max steps	2M	normalize	false
# epoch (MLP)	3	sequence length (LSTM)	32	memory size (LSTM)	256
# hidden layers (LSTM)	1	time horizon (LSTM)	64	buffer size (LSTM)	1024

Next, we select four out of the five agents of each representation according to their cumulative rewards (we keep the four agents with the largest accumulated rewards). Then, the selected agents battle against the other agents that learned with the same representation as in a T100. The tournaments are accomplished with four agents because this is the maximum number of agents allowed in the same scenario.

The percentage of wins considering each battle in the T100 tournaments are: BF_2 (28.5% of victories), NBF_1 (28.2%), H_1 (38.5%), I_4 (27.2%), ZO_1 (29.7%). Next, we would like to make these winners fight against each other in a new tournament. Again, as we have at most four agents in a scenario, we must discard the worst of these five. For so, we rely again on their training accumulated reward. Then, we have a new set composed of the winner agents $W = \{H_1, I_4, ZO_1, NBF_1\}$ as BF_2 was the worst of them in the training phase. By analyzing the behavior of the agents in W when battling, we observed that they were able to learn how to play the Bomberman game correctly. They usually

put bombs and escape from them, except for NBF_1, which only sometimes acts as expected.

Finally, we yield a $T1000$ tournament to make the agents in the set W to battle against each other, aiming at verifying the best state representation for the Bomberman environment. The results are listed in Table 3. We can see that H_1 was the one reaching the most significant number of victories (4442), pointing out that this should be the best way of representing the environment.

Next, we experiment with LSTM to approximate the policy function within the PPO algorithm and verify if this would improve the performance of an agent that uses the Hybrid representation (the winner of the previous tournament). As LSTM requires a lot more parameters than MLP, we change some hyperparameters to alleviate the runtime. We trained five models with LSTM and did a T100 with the four models that obtained the best cumulative rewards. The winner of them was the second model generated in the training, achieving 31.4% of wins.

Then, we created another tournament called T1000-LSTM to compare two LSTM agents with two hybrid-MLP agents, the winner of the T1000-representation battle. As can be seen in Table 4, an agent trained with LSTM won the tournament with 61.02%. These results indicate that by activating LSTM in the PPO algorithm, we can induce agents that learn better from their environment to the point of winning other agents. This is due to the memory capacity of LSTMs that allows for the agent learn what to do, even when the reward is in a long way ahead in the future.

Table 3. Results of a T1000 tournament that compares agents with different state representations

Results	No. of Wins
Draw	263 (2,63%)
NBF_1	319 (3,19%)
H_1	**4442 (44,42%)**
I_4	1782 (17,82%)
Z_1	3194 (31,94%)
Total	10000 (100%)

Table 4. Results of a T1000-LSTM Versus Hybrid tournament

Results	No. of Wins
Draw	395 (3,95%)
LSTM 2	**6102 (61,02%)**
H_1	3503 (35,03%)
Total	10000 (100%)

Finally, we trained an agent with the BC algorithm and the Hybrid representation and tried to continue its training with PPO. Before training, we recorded 40 matches in a replay file where we play at least three matches in static scenarios and 20 matches in random ones, fighting against others that use LSTM. This replay file acts during the BC training as the expert so that the student agent can learn to play from it. In the BC training, we use the same hyperparameters used in Table 2 except that we changed **max steps** to 100K and set **batches per epoch** to 5. If the learner dies or if the game is taking more than 400 iterations to finish, we reset the scenario and load the next match of the replay file.

After finishing the BC training, we may still improve the policy function of the agent following two ways: **i) (BC+PPO_Only1):** we run PPO starting from the policy learned by BC to train one agent against enemies already trained with the PPO-LSTM-Hybrid-state setting; **ii) (BC+PPO_All):** In the second way of continued training, PPO runs after BC as before, but all the agents in the scenario are still learning. We create a series of T100 and T1000 tournaments to observe the BC agents behavior and found out the following: **(A) (T1000 BC-vs-LSTM):** The BC agent plays reasonably only when it is in the exact scenarios he has trained before, and when his enemies behave in the same way as in the replay file. When the BC agent battles against the LSTM in a T100 one-vs-one agent tournament, it wins only 9.7% of the matches. **(B) (T1000 BC+PPO_Only1 vs LSTM):** In this case, the BC-PPO_Only1 wins the LSTM in 91.3% of the matches. This result shows that it is useful to start from a trained BC agent and refine it with PPO. **(C) (T1000 BC+PPO_All vs. LSTM):** Here, the winning agent was the BC+PPO_All with 56.69% of wins against 39.98% of the LSTM agent. These results further confirm that BC provides a good starting point to PPO and that the knowledge acquired in the previous training with BC has not been forgotten. **(D) (T1000 BC+PPO_Only1 vs. BC+PPO_All):** The BC+PPO_All won 49.5% of the matches whereas the BC+PPO_Only1 won 42.3% of the matches. This result shows that the agents may learn better when facing other agents that are learning as well. However, this was the most closely results from all the others. Note that the LSTM opponents used in BC pre-training are also used in the BC+PPO_Only1 training. However, in BC+PPO_All training, LSTM opponents are not used, as all the agents are using the same controller and, therefore, the same policy function. This explains why BC+PPO_All wins fewer matches against LSTM opponents than BC+PPO_Only1 as this later becomes an expert at beating LSTM because it trained only with these opponents.

6 Conclusions

In this work, we investigated five state representations and four learning algorithms to build Bomberman agents that learn how to play. Regarding the state representations, the results pointed out that our proposed hybrid representation achieves the best results in test time, experimented with tournaments conducted after the learning phase. Regarding the learning algorithms, we experimented with the actor-critic-based PPO algorithm using either an MLP or an LSTM, the BC imitation learning, and a novel approach developed by us that starts with BC and continues the training with PPO from the function learned with BC. The results pointed out that by coupling LSTM within the PPO algorithm produces smarter agents and training beforehand with the BC algorithm can influence subsequent training with PPO. As future work, we would like to enhance the Bomberman agents by giving them the ability to put several bombs at once and acquiring power, variable-size scenarios, multi-players and testing other RL and IL algorithms.

References

1. R. Sutton and A. Barto, "Reinforcement learning: An introduction (in preparation)," 2017.
2. Y. Li, "Deep reinforcement learning: An overview," *CoRR*, vol. abs/1701.07274, 2017.
3. D. Silver, T. Hubert, J. Schrittwieser, I. Antonoglou, M. Lai, A. Guez, M. Lanctot, L. Sifre, D. Kumaran, T. Graepel, T. Lillicrap, K. Simonyan, and D. Hassabis, "A general reinforcement learning algorithm that masters chess, shogi, and go through self-play," *Science*, vol. 362, no. 6419, pp. 1140–1144, 2018.
4. V. Mnih, K. Kavukcuoglu, D. Silver, A. Graves, I. Antonoglou, D. Wierstra, and M. A. Riedmiller, "Playing atari with deep reinforcement learning," *CoRR*, vol. abs/1312.5602, 2013.
5. C. Resnick, W. Eldridge, D. Ha, D. Britz, J. Foerster, J. Togelius, K. Cho, and J. Bruna, "Pommerman: A multi-agent playground," in *Joint Proc. of the AIIDE 2018 Workshops co-located with 14th AAAI Conf. on AI and Interactive Digital Entertainment (AIIDE 2018)*, 2018.
6. M. A. da Cruz Lopes, "Bomberman as an artificial intelligence platform," Master's thesis, Universidade do Porto, 2016.
7. E. Karasik and A. Hemed, "Intro to ai bomberman." `http://www.cs.huji.ac.il/~ai/projects/2012/Bomberman/`, 2013. Accessed in 2018-11-25.
8. J. G. Kormelink, M. M. Drugan, and M. A. Wiering, "Exploration methods for connectionist q-learning in bomberman," in *Proceedings of the 10th Int. Conf. on Agents and Artificial Intelligence, ICAART 2018*, pp. 355–362, 2018.
9. J. Schulman, F. Wolski, P. Dhariwal, A. Radford, and O. Klimov, "Proximal policy optimization algorithms," *arXiv preprint arXiv:1707.06347*, 2017.
10. A. Juliani, V. Berges, E. Vckay, Y. Gao, H. Henry, M. Mattar, and D. Lange, "Unity: A general platform for intelligent agents," *CoRR*, vol. abs/1809.02627, 2018.
11. S. Hochreiter and J. Schmidhuber, "Long short-term memory," *Neural computation*, vol. 9, no. 8, pp. 1735–1780, 1997.
12. M. Lapan, *Deep Reinforcement Learning Hands-On: Apply modern RL methods, with deep Q-networks, value iteration, policy gradients, TRPO, AlphaGo Zero and more*. Packt Publishing Ltd, 2018.
13. K. Hornik, M. B. Stinchcombe, and H. White, "Multilayer feedforward networks are universal approximators," *Neural Networks*, vol. 2, no. 5, pp. 359–366, 1989.
14. J. Schulman, S. Levine, P. Abbeel, M. Jordan, and P. Moritz, "Trust region policy optimization," in *Int. Conf. on Machine Learning*, pp. 1889–1897, 2015.
15. M. Lanham, *Learn Unity ML-Agents–Fundamentals of Unity Machine Learning: Incorporate new powerful ML algorithms such as Deep Reinforcement Learning for games*. Packt Publishing Ltd, 2018.
16. V. Mnih, A. P. Badia, M. Mirza, A. Graves, T. Lillicrap, T. Harley, D. Silver, and K. Kavukcuoglu, "Asynchronous methods for deep reinforcement learning," in *Proc. of the 33rd Int. conf. on Machine Learning*, pp. 1928–1937, 2016.