Bomberman Project

Alicia Fortes Machado Aloysio Galvão Lopes Iago Martinelli Lopes Igor Albuquerque Silva

***Abstract*—In this work, our main objective was to train an autonomous agent to play the game *Bomberman*. We decided to build our own environment, as, this way, we would have full control over the rules of the game, which allowed us to make simplifications. To tackle this problem, we decided to, in a first moment, define a simplification of the problem that would allow the use of some tabular method. In our case, we used Q-learning to enable our agent to learn how to break blocks in a smaller version of the environment. After that, we implemented an agent based on Double Deep Q-Learning (DDQN), that is better suited for larger environments. Furthermore, we explored Policy Gradient Methods such as REINFORCE [17] and an Actor Critic [8] method, as well as techniques such as using a frame stack, Prioritised Experience Replay and Never Give Up [5]. We compared the performances of our agents on the environment variations that we have created. In the end, our Q-learning agent successfully mastered the task in small environments, and the DDQN was able to learn to destroy up to 20 blocks. Our other agents were unable to break blocks properly.**

I. Introduction

Bomberman is a maze-based video game developed by Hudson Soft and released in 1985 on NES platforms (1987 in US) [1]. After that, it had many remakes and improvements. One of the possible game modes is as it follows: the Bomberman must destroy all soft blocks (breakable blocks) using bombs within limited time (clean the map). Another way to play it is against other Bombermans, where the goal is to kill other players and be the last one to survive.

Previous works such as [13] have already focused on creating a Bomberman environment. Nevertheless, we decided to create our own environment, as, this way, we'd be able to create a more customizable framework. Following that approach, we had more flexibility to adapt the problem to our computational resources and to the state-space representations we wished to use. We also had more knowledge about the code which allowed us to quickly make changes to the environment when needed.

We knew in advance that the Bomberman game can have an enormous number of states. Therefore, in order to solve the game, we would need to be able to compute policy, value or action-value functions via neural networks. At the same time, a tabular method can be much easier to implement and debug. So, throughout our project, we decided to develop four main agents. The first two of them used Q-learning: one tabular based and one neural network based. More precisely a Q-learning agent, that fills a table with action-values in a smaller Bomberman environment, and a DDQN (Double Deep Q-Learning)-based agent that approximates the action-value function with a neural network and is able to play in a standard bigger environment. This approach also allowed us to make a direct comparison between Q-learning and Deep Q-learning. The other two are based on Policy Gradient: Advantage Actor Critic (A2C) [8] and REINFORCE [17]. Those were used mainly as comparison, as, in this task, they were expected to perform worse.

Even though we found some works that explore Q-learning in the Bomberman game [2], we haven't found any work comparing tabular Q-learning, DDQN, A2C and REINFORCE on Bomberman. Furthermore, in our setting, we compared the methods in fair conditions (the same environment, number of roll-outs etc). Of course, no other work has been done so far using particularly our environment.

The core challenge of the work presented here is to make a DDQN agent learn to play well Bomberman regarding the "cleaning the map" setting. This means that the agent should learn how to break blocks in a given map. The main problem that the game poses is that the agent places bombs that will explode after some time, but the environment does not provide the time until the explosion. With this idea in mind, we had to figure out how to introduce this information on our agents, which led us to try two ideas: using frame stacking and saving an internal timer. Other than that, we implemented some extensions on the DDQN method, which will be further explained in the next sections, as well as the the rules of the game.

We've prepared some demos of our code and we strongly encourage you to try to run them. The link to our code is presented below:

https://github.com/aliciafmachado/bomberman-agent

To run the demos, simply follow the instructions in the *README.md* to install your code and then check the section *Demo* of our *README.md* file.

II. Background and Related Work

Reinforcement learning (RL) has seen a huge expansion in recent years, and it is still a very active area of research. Lots of recent works succeeded to overcome human benchmarks in areas where it was previously believed to be impossible. Notably, we see examples in games like Go [15], DOTA 2 [7] and many famous Atari games [4]. Inspired by these projects, we discuss a few methods that have been proven useful for these problems and that were used to train agents in our simplified version of Bomberman (detailed in Section III). We took special inspiration from Agent57 [4] which is able to overcome humans in 57 Atari games.

The first one is the Q-Learning algorithm, better explained in [10]. The basic idea is to create a table of state-action values as we explore the environment and fill it using Equation 1.

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha(r_t + \gamma \max_{a \in \mathcal{A}} Q(S_{t+1}, a) - Q(S_t, A_t)) \tag{1}$$

During the training step, an $\epsilon-$greedy policy can be used in order to encourage exploration.

As we need to map all state-action pairs to values, the memory required is proportional to the complexity of the environment in terms of possible states and actions. Thus, it is not feasible in problems where there are too many possible state-action pairs.

Considering that, a new method was proposed: Deep Q-Learning [11], which, given a state as input, uses a neural network to estimate the value for each possible action. The weights are updated as in Equation 2. Originally, MSE Loss is used, but Smooth L1 Loss [3] is less sensitive to outliers and prevents exploding gradients. Thus, we chose to use the Smooth L1 Loss.

$$\begin{aligned} w = & w + \alpha * [\underbrace{(R_t + \gamma \max_{a \in \mathcal{A}} [q_w(S_{t+1})]_a)}_{target} - \underbrace{q_w(S_t)_{A_t}}_{predict}] * \\ & \nabla_w [q_w(S_t)_{A_t}] \end{aligned} \tag{2}$$

After further work, a new method was developed, Double Deep Q-Learning [16], which is of particular interest to us. This method improved the previous one by using two neural networks instead of only one: the *q-net* and the *target-net*. The *q-net* gets its weights updated, while the *target-net* chooses the action. After a certain number of steps, the weights of the *q-net* are copied over to the *target-net*. Thus, the weights are updated as it follows in Equation 3.

$$\begin{aligned} w^{qnet} = & w^{qnet} + \alpha * [(R_t + \gamma \max_{a \in \mathcal{A}} [q_w^{qnet}(S_{t+1})]_a) - \\ & q_w^{targetnet}(S_t)_{A_t}] * \nabla_{w^{qnet}} [q_w^{qnet}(S_t)_{A_t}] \end{aligned} \tag{3}$$

Policy Gradient methods can also be of particular interest as they've already been used in the Bomberman game as in [19]. Using a stochastic policy also allows us to encode exploration naturally in our agent. Two methods are of interest to us: REINFORCE [17] and actor critic methods [19]. In REINFORCE [17] (Monte-Carlo Policy Gradient) the idea is to roll-out an episode following a stochastic policy π_θ parameterized by θ then use the update rule defined in Equation 4, based in the policy gradient theorem [17].

$$\theta_{t+1} \leftarrow \theta_t + \alpha \nabla_\theta log \pi_\theta(s_t, a_t) G_t \tag{4}$$

Please note that G_t is the discounted future reward and α is the learning rate. This method suffers from the high variance of the gradient.

Actor critic methods [8] were introduced to try to mitigate this problem, in our case we opted to use the advantage actor critic (A2C). The idea is to subtract a baseline from the discounted future reward. In this case, we just have to change G_t in Equation 4 for the advantage defined as $A(s_t, a_t) = Q(s_t, a_t) - V(s_t)$. We finally get the following update rule for the A2C method in Equation 5.

$$\theta_{t+1} \leftarrow \theta_t + \alpha \nabla_\theta log \pi_\theta(s_t, a_t) A_t \tag{5}$$

III. The Environment

Bomberman is a 2D-game game where each player can place bombs that explode on all four sides. If the bomb's explosion hits a hard block nothing happens, but if it hits a soft block, the block will disappear after a few seconds. The player also dies if it touches explosions, so it must be careful to put bombs and then take cover behind hard blocks.

We developed a new environment completely from scratch. We decided to do this for two reasons: to have a better understanding of how to develop a gym environment, and to have more control over what was being executed. We made it configurable in size, events, reward values and how it is rendered, to allow us to test our agents in multiple scenarios.

We did a quick analysis of existing environments, most notably *Pommerman* [13] and *bomberman-rl-keras* [12]. The former is very complete in terms of the code, but it is not the same game (has no fixed blocks) and has a rather simple interface. The latter also has a very simplistic interface, and provides only a partially observed environment, which is not our objective.

We decided on having a discrete state implementation, and a fully observed environment. Therefore, we represent the state as a three dimensional matrix of shape $(h, w, 5)$, in which h and w are the height and width of the field (the number of blocks in each dimension), and 5 is the number of layers that represents the objects in the field, which are fixed blocks, destroyable blocks, bombs, fires and characters. We also made a variation of the environment, which centralises its states on the agent (and becomes partially observable) to test our models in this variation.

The rendering is implemented using Pygame, and we used open source sprites for the game's art. To keep the animations running smoothly and the agent steps discrete, the environment renders a few times every step. This way, for example, the character will seem to walk between two positions on the screen while for the observation matrix only the two discrete positions will be found. Figure 3 contains an illustration of the rendered environment.

The agent has 6 possible actions in each step: doing nothing, moving left, moving right, moving up, moving down and placing a bomb.

Our main goal is to have an agent that destroys blocks as quickly as possible without dying. In this task the environment is deterministic, since all of the events times are fixed, depending only on the agent's actions. The environment also allows for a more ambitious goal, which is playing against up to three other agents. In these settings, the environment is stochastic for each agent, because now the next states depend also on the other agent's actions.

We defined seven types of rewards, given each time the agent completes an action. Table I describes each event and the associated rewards.

Since it is a game, the environment is for research and educational purposes only. The nature of the game makes this a fairly complex task for a reinforcement learning agent, since the state space is too large. In a 11x13 field, for example, the number of states are in the order of $2^{13 \cdot 11 \cdot 5}$. The task

TABLE I
LIST OF REWARDS DEFINED IN THE ENVIRONMENT.

Event	Reward
Destroy a block	1
Bomb destroys no block	-0.3
Place bomb	0.2
Death	-1
Kill	1
Visit new cell	0.04
Illegal movement	-0.2

it has to do is also complex, because the agent has to make a sequence of correct actions to avoid dying and receive a reward (it has to place a bomb near a block, walk twice in perpendicular directions, then wait for it to explode). This temporal dependency can be tricky for agents to learn, and it must keep repeating this pattern throughout the whole episode.

IV. THE AGENT

We chose to implement four different agents. The first two were off-policy, Q-learning methods: one tabular based (Q-learning) and the other using two neural networks (Double Deep Q-Learning). The other two were on-policy, policy gradient methods, mainly used for comparison purposes: Advantage Actor critic and REINFORCE. It's important to observe that we expected the first two methods to perform better as they are generally more sample efficient, and because they are also the basis of the Agent57 agent [4], which is a highly advanced state-of-the-art agent that outperforms the average human in 57 Atari games.

A. Q-Learning

This agent is simply based on the update rule from classical Q-learning, shown in more detail in Section II. The main reason for implementing this agent was to create a baseline. As this agent was simpler to implement and test, it helped us to integrate our agents' code with the environment as well as give us some insight about the expected performance of Q-learning. Before implementing it, we concluded theoretically that the states of an environment of size 5×5 (the smallest we could do) would fit in a Q-table, this way, we expected this agent to be able to learn in this small environment.

We implemented the Q-table as a dictionary, for this reason, the table only grows when the environment is explored. Therefore, we're allowed to run this agent in bigger environments as well, but it's not expected to be able to learn values for all of the states.

To fill in the Q-table, we experience the environment following an ϵ-greedy strategy, we tested several ϵ values and we concluded that we obtained better performance taking random actions with a probability of 0.4 and taking the action with best Q-value with probability 0.6.

In the beginning, just taking into account the observation given by the environment as state, we were unable to make this agent learn. The fact that a bomb near the moment of explosion and a bomb far from this moment were represented in the same way in the given observation made it impossible for the update algorithm to converge. This meant that the observation didn't describe completely the state. To solve this problem, we added a timer to the state. This way, every entry in the Q-table is represented as the concatenation of the observation and a timer. This timer start counting when a bomb is placed and resets some steps after it's explosion. With this adjustment, the algorithms started to converge and this agent was able to solve the smaller environments (5×7 and 5×5). This also inspired us to add a timer and, later, a frame buffer to our Double Deep Q-learning agent.

We decided to initialise the table (every new dictionary entry) with all zeros, this way, the agent is not naturally driven to explore and it relies on the ϵ-greedy strategy to explore. This was our option because, even though we could get faster learning with an exploration-driven initialisation (Q-table initialised with positive values), the agent would very fast kill itself in new states. This behaviour is undesirable when playing, possibly, against other players in a big environment, this way, we opted to have a slightly slower learning time to keep a more conservative strategy. The idea behind this is that this agent won't kill itself soon.

In the end, this implementation was simple to understand and gave us the principle that could be extended to our Double Deep Q-learning agent. However, this implementation had lots of disadvantages, the two main ones are: the method can't generalise for large environments and the fact that, for a never seen state, the agent wouldn't be able to make any kind of guess about what action to take and would act randomly. This means that this agent is only able to take reasonable actions with states that it has already experienced several times.

B. Double Deep Q-Learning

This agent was used for the largest environment that created, which was composed of a grid 11×13. We chose to use two neural networks for the agent (*target-net* and *q-net*) since it increases stability as stated in [16] and explained in Section II. We based our implementation on the Pytorch tutorial on Deep Q-Learning [9].

The architecture used for both neural networks in our agent is further detailed in Appendix VI-A, on Table II. We first pass our data through two residual blocks composed by two convolutional layers in each, following these layers we have linear layers in order to get the expected values for each action. We decided to use this architecture because 2D convolutions make sense in our scenario: our grid representation can be viewed as a simplified version of the game's image. As we know that convolutional networks tend to deal well with images this was a natural choice.

Temporal information

In order to get a sense of when the bomb would explode, we implemented an internal counter for the agent that counts from 9 to 0, starting at 9 when a bomb was dropped. This was done as the state we pass to the agent doesn't give any temporal information. Aside from that, instead of simply passing the time as part of the state to the model, we transformed it into an one hot vector.

An alternative of using an internal counter is to pass a frame stack instead of only one state. This gives a temporal sense

for the agent and is a common practice among reinforcement learning agents. We also implemented it and compared to the previous idea, which is further discussed on Section V.

Exploration decay

Instead of just using an ϵ-greedy exploration, we implemented an exploration factor for the training that was reduced linearly, which improved our results.

After a bit of manual tuning, we trained our agents with the exploration factor starting at 0.9 and decreasing linearly to 0.1 after 500 episodes.

Prioritised Experience Replay (PER)

Having a replay buffer is a common technique to improve the performance of the model. The simplest buffer is one that randomly selects a batch for training, but this can be inefficient if the environment has very sparse rewards for example, or if some states are more important than others.

The prioritised memory is a technique introduced on [14] which tackles this problem. One way of doing it is giving weights to each state that is stores in the buffer, and when sampling a batch it uses these weights as the chosen distribution. These weights are updated at each training step, with the difference between the current state's value and the expected state's value (TD error), "which indicates how 'surprising' or unexpected the transition is" [14]. We wrote this algorithm based the implementation contained in [18].

Never Give Up (NGU)

In 2019 Deep mind introduced another breakthrough in AI using an agent to play all 57 Atari games. They introduced the NGU [5] agent. The main new idea of the paper was to introduce a new exploration strategy. As, according to our qualitative analysis, our agent was having trouble to explore the environment, it was natural to try to integrate the main ideas of the NGU agent. We took inspiration on the implementation provided in [6] to develop our code.

With this in mind, we added an episodic memory to our DDQN agent that recorded every state within an episode. The idea of this new strategy is to add an intrinsic reward based on the distance between the current state and the k states closer to it that are present in the memory. This KNN-based approach would then encourage, in a given episode, the agent to try to visit states that are not well visited.

C. REINFORCE

This agent is based on a stochastic policy π_θ where θ represents the parameters of our neural network. The updates to the network are made every time an episode is finished following the policy π_θ given in Equation 4. To be able to calculate the expected future reward G_t, we recorded the actions and rewards for a given roll-out (until the agent dies). In this case, the agent performed very poorly so we only used a very simple neural network that used only five linear fully connected layers with ReLU activations. The first layer took as input a flattened version of the Bomberman grid and the subsequent layers had, respectively 300+10, 150+10, 60+10, 30 inputs. The final layer had 6 outputs representing the six possible actions the Bomberman can take (5 movements and placing the bomb), a final Softmax made sure that the output summed to one and consisted only of non-negative values. The +10 in each layer represents that it took also as input a one hot encoded timer. The timer was reset each time the Bomberman placed a bomb.

D. Advantage Actor critic

This agent also takes actions based on a stochastic policy π_θ where θ represents the parameters of a neural network. The main difference compared to our REINFORCE agent is that, now, our agent is not only going to output a policy but also a value function, therefore the network has two output heads. The value is going to be used to calculate the advantage $A(s_t, a_t)$ defined in Section II. Now, we run roll-outs and record not only the rewards and the actions, but also the value predicted by the network. The policy head is trained following Equation 5 and the value head is trained based on the Smooth L1 Loss [3].

For our network, as this agent showed promising results with simple linear dense layers, we used the same first four convolutional layers as our Double Deep Q Learning Agent. Then we use two linear layers with inputs of size 200+10 and 50+10. We replicated the same architecture in separate networks, one to calculate the value and another to calculate the policy. This way, policy and value don't share parameters, thus both heads are independent. Finally we highlight that, as for our REINFORCE approach, the +10 in the input of the linear layers represents the bomb timer encoded as one-hot.

V. Results and Discussion

Now, we discuss the results of the agents and extensions referred above.

A. Q-Learning

As discussed, we trained this agent in smaller fields (at most 5×7 grids) and it learned to destroy all the blocks and not kill itself with very high stability. In the end, the Q-table had saved 2200 different states, showing that the number of states is much larger than the size of the map, making it impossible to use this model on larger maps. This limitation was observed when we tried to apply this model to the map 11×13, where the memory required increases constantly. Even though this agent is able to play in the 11×13 setting, it won't be able to play for any given state because of memory limitations.

B. Double Deep Q-Learning

By using the methodology explained in the previous section, we trained multiple agents using different configurations. The results can be seen in Figure 1 and we can observe that the DDQN surpassed all other models, since it's able to achieve better rewards with the same amount of exploration than other methods.

By putting the best trained agent into practice, we observe that it is able to follow an optimal path until it destroys more or less 10 blocks, then it starts to do movements that he clearly

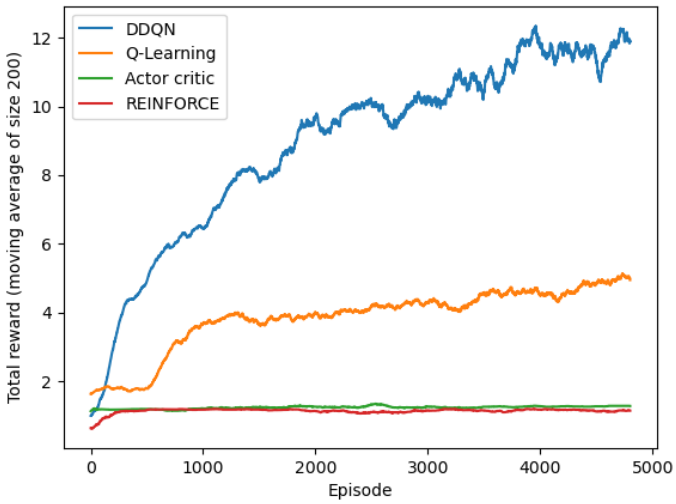

Fig. 1. Rewards evolution of the four reinforcement learning agents we implemented, on the 11×13 environment. The y-axis represents the sum of rewards obtained in the episode, as defined in Table I.

should not, until it dies. So it is possible to deduce that he is having difficulty to train states that takes longer to arrive.

In the sense of fine tuning our hyperparameters, we tested different learning rates and exploration factors, which made us arrive to the conclusion that the hyperparameters presented in Appendix VI-A, on Table III, are the best ones we found for training the agent.

We also did a lot of analysis when adding more techniques into the agent. Figure 2 shows a comparison of the method with and without some of these additions, as well as some failed ideas such as a centralised environment and the frame stack.

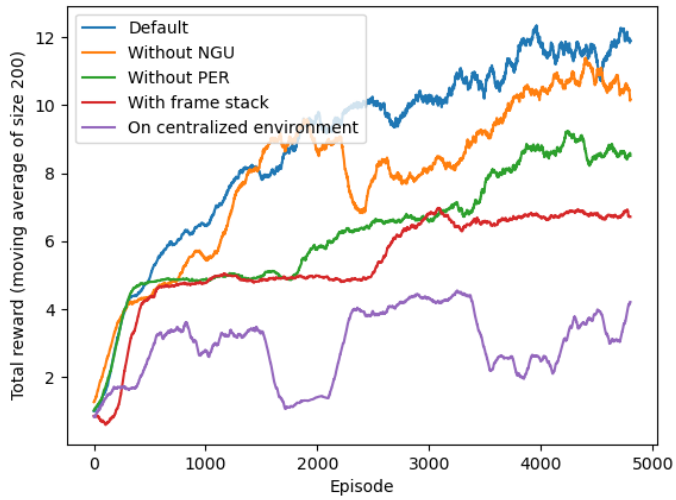

Fig. 2. Comparison of the DDQN default agent (with timer, NGU, PER, on fixed environment) with its variations. It shows rewards improvement over training on a 11×13 environment.

The centralised environment was an attempt to allow the agent to perform better on new states, because with the map centralised on him the beginning states look more similar to the end ones than otherwise. However, it performed much worse than without this variation. One reason for this could be because there are too many values changing on the state transitions, making the task too difficult to learn.

The frame stack approach also performed worse than expected. We believe this could be caused by two reasons: the first is that this method is usually used when taking the actual frames rendered by a game, and in our case it wouldn't bring too much benefits. Another possible reason is that we should have increased the size of the network to accommodate a larger input size, which we didn't have time to do.

Both the PER and NGU were beneficial for the agent. Exploring is one of the hardest challenges of our environment, so these methods did in fact help the agent learn faster.

Finally, both Policy Gradient methods performed very poorly on our tests, they tended to be stuck in the first action that gave a positive reward. In our case, this means that they'd place a bomb and kill themselves most of the time. The actor critic agent (A2C) performed slightly better than the REINFORCE one and it's rewards were constantly evolving, which suggests that the use of a deeper network with much more training would allow it to learn how to break blocks. As expected, those methods need much more training time to achieve good results.

VI. Conclusion and Future Work

We successfully developed a brand new discrete and fully observed Bomberman environment, which we used as a basis to implement four different reinforcement learning algorithms: Tabular Q-Learning, Double Deep Q-Learning, REINFORCE and Advantage Actor Critic. The first two algorithms were able to solve the small environment, but in the default environment DDQN worked best for a same number of episodes used to train.

We believe that the DDQN algorithm can still be improved to make the agent learn more and faster. We could reduce the initial number of maximum steps - which corresponds to the number of steps until the environment is reset - and increase it linearly in relation to the episodes, so that our agent would learn better the beginning and reach distant spaces more easily. We could also start our agent from a random state of the memory, or in a random position of the map.

We had promising results regarding the evolution of the reward, but the movements are still too complex for the model to learn with the state representation and techniques we used. Some ideas are to train the agent in a simpler environment (for example make bombs and fires explode faster, remove fixed blocks, etc), make extensions to the Double Deep Q-Learning method and study changes in the network. Some possible extensions to the DDQN, based on the path followed to get Agent57 [4], are the usage of short term memory (LSTMs and GRUs), episodic memory and more complex exploration methods.

Finally, a future work that can be done is to use the agents and adapt the"kill" reward in order to make them fight each other in the multi-agent environment mode, which is already implemented. Other one is to support "power ups", which are improvements that are dropped from some soft blocks (breakable blocks) when broken. When a Bomberman takes it, he gets more bombs, range or speed.

REFERENCES

[1] Bomberman (nes). https://bomberman.fandom.com/wiki/Bomberman_(NES).

[2] Simulation of the classic game bomberman using q-learning. https://github.com/InduManimaran/Bomberman.

[3] Smoothl1loss. https://pytorch.org/docs/stable/generated/torch.nn.SmoothL1Loss.html, biburl = https://pytorch.org/docs/stable/generated/torch.nn.SmoothL1Loss.html.

[4] Adrià Puigdomènech Badia, Bilal Piot, Steven Kapturowski, Pablo Sprechmann, Alex Vitvitskyi, Daniel Guo, and Charles Blundell. Agent57: Outperforming the atari human benchmark, 2020.

[5] Adrià Puigdomènech Badia, Pablo Sprechmann, Alex Vitvitskyi, Daniel Guo, Bilal Piot, Steven Kapturowski, Olivier Tieleman, Martín Arjovsky, Alexander Pritzel, Andew Bolt, and Charles Blundell. Never give up: Learning directed exploration strategies, 2020.

[6] Adrià Puigdomènech Badia, Pablo Sprechmann, Alex Vitvitskyi, Daniel Guo, Bilal Piot, Steven Kapturowski, Olivier Tieleman, Martín Arjovsky, Alexander Pritzel, Andew Bolt, and Charles Blundell. Never give up: Learning directed exploration strategies, 2020.

[7] Christopher Berner, Greg Brockman, Brooke Chan, Vicki Cheung, Przemyslaw Debiak, Christy Dennison, David Farhi, Quirin Fischer, Shariq Hashme, Christopher Hesse, Rafal Józefowicz, Scott Gray, Catherine Olsson, Jakub Pachocki, Michael Petrov, Henrique Pondé de Oliveira Pinto, Jonathan Raiman, Tim Salimans, Jeremy Schlatter, Jonas Schneider, Szymon Sidor, Ilya Sutskever, Jie Tang, Filip Wolski, and Susan Zhang. Dota 2 with large scale deep reinforcement learning. *CoRR*, abs/1912.06680, 2019.

[8] Vijay R Konda and John N Tsitsiklis. Actor-Critic Algorithms. page 7.

[9] Adam Paszke. Reinforcement learning (dqn) tutorial. https://pytorch.org/tutorials/intermediate/reinforcement_q_learning.html, 2017.

[10] Jesse Read. Reinforcement learning ii, 2021.

[11] Jesse Read. Reinforcement learning iii, 2021.

[12] Henrik Reinstädtler. Bomberman rl keras. https://github.com/henrixapp/bomberman-rl-keras/tree/master/gym-bomberman.

[13] Cinjon Resnick, Wes Eldridge, David Ha, Denny Britz, Jakob Foerster, Julian Togelius, Kyunghyun Cho, and Joan Bruna. Pommerman: A multi-agent playground. *CoRR*, abs/1809.07124, 2018.

[14] Tom Schaul, John Quan, Ioannis Antonoglou, and David Silver. Prioritized experience replay, 2016.

[15] David Silver, Aja Huang, Christopher Maddison, Arthur Guez, Laurent Sifre, George 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. Mastering the game of go with deep neural networks and tree search. *Nature*, 529:484–489, 01 2016.

[16] Hado van Hasselt, Arthur Guez, and David Silver. Deep reinforcement learning with double q-learning. *CoRR*, abs/1509.06461, 2015.

[17] Ronald J Williams. Simple statistical gradient-following algorithms for connectionist reinforcement learning. page 28.

[18] Dulat Yerzat. Dqn adventure: from zero to state of the art. https://github.com/higgsfield/RL-Adventure/.

[19] Eugene Zainchkovskyy. eugene/pommerman, March 2021. original-date: 2019-01-01T15:49:22Z.

APPENDIX

A. Final parameters of DDQN agent

After manually tuning the agent, we decided on using a mixture of convolutional layers with residual components and finishing with linear layers on the model. Table II details each of these layers and their parameters.

TABLE II
DEEP Q-LEARNING MODEL LAYERS

residual block	layer	# of filters	filter size	stride	paddings	batchnorm	activation function
yes	conv1	16	3	1	0	yes	ReLU
	conv2	32	3	1	1	yes	ReLU
yes	conv3	64	3	1	1	yes	ReLU
	conv4	64	3	1	1	yes	ReLU
	layer	# of out features				batchnorm	activation function
	linear1	200				no	ReLU
	linear2	50				no	ReLU
	linear3	6				no	None

Table III details hyperparameters we used to train our best DDQN agent, which was able to break consistently 12 blocks after 5000 episodes.

TABLE III
TABLE OF HYPERPARAMETERS OF OUR MODEL

Learning rate	0.001
γ	0.9
Batch size	32
# of episodes to update the target net	10
# of episodes	5000
ϵ-greedy policy	ϵ varies linearly from 0.9 to 0.1

B. Environment snapshot

Figure 3 contains snapshots of our environment when rendered, including a shot with multiple agents.

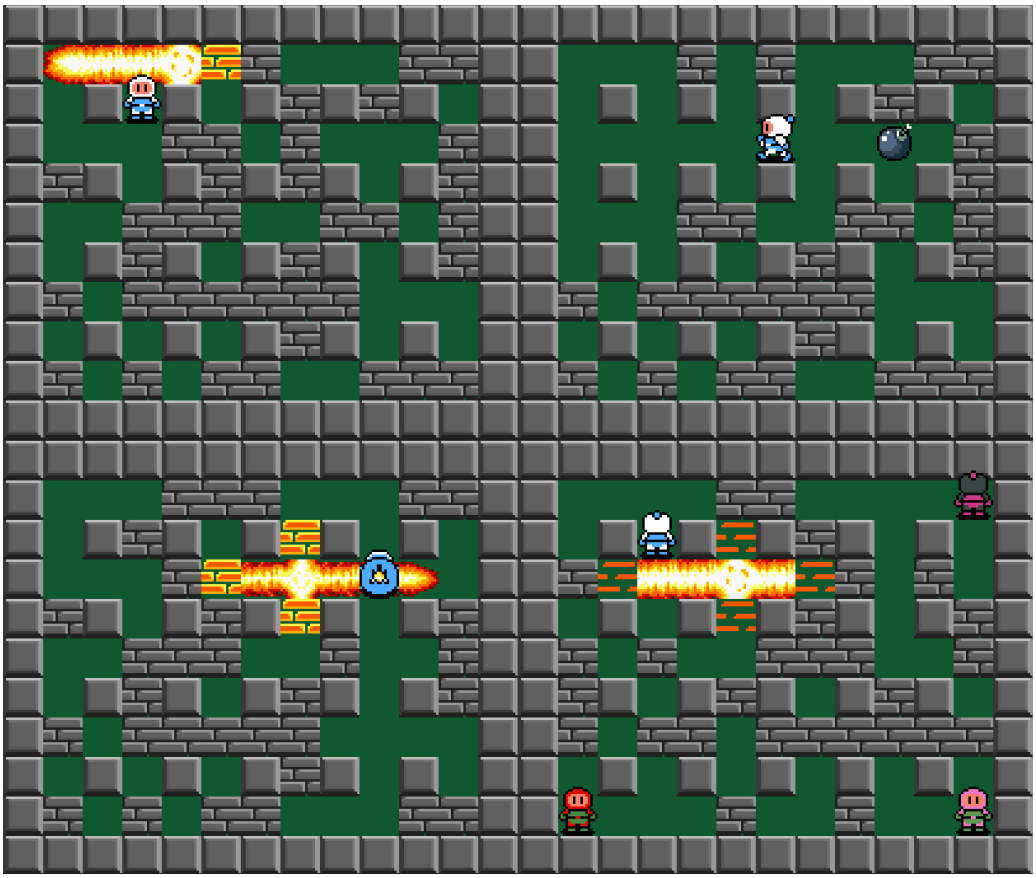

Fig. 3. Four example frames rendered by our environment. The explosions are generated by the bombs after a certain time. They destroy the wall blocks and can kill the agents if they touch them. The fourth frame contains the more ambitious goal, of competing against other agents.