Multi-agent Reinforcement Learning with ‘Pommerman’

Marcel Banser, Bhaskar Majumder, Kamran Vatankhah-Barazandeh
Deep Reinforcement Learning - Osnabrück University - WiSe 20/21

Introduction

In this project, we try to implement an agent for the Pommerman multi-agent environment [1], [2] which should learn to play the game and ideally outperform the baseline “simple_agent” that is included in the environment.

Related Work

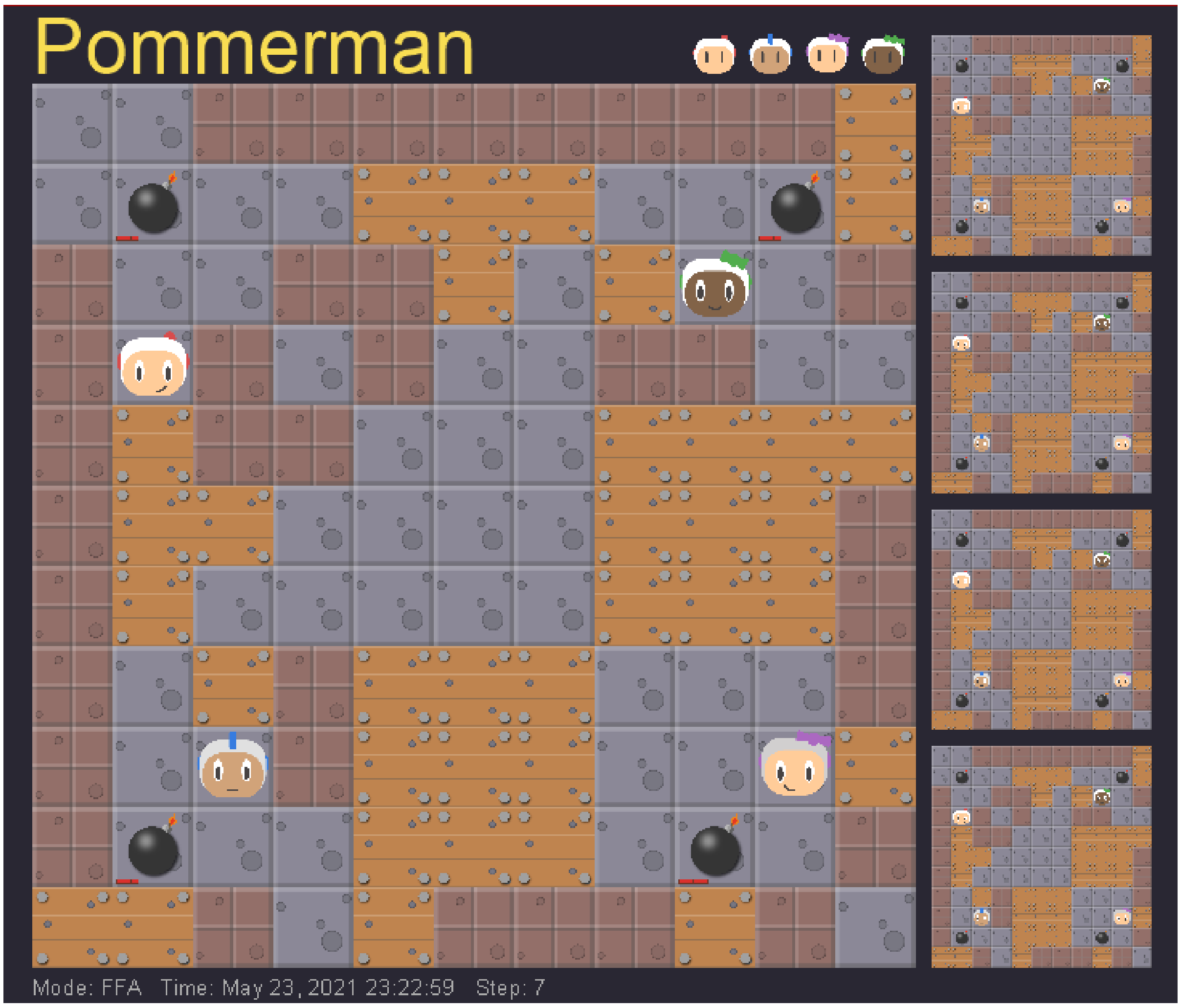
There are many attempts at solving this problem using deep reinforcement learning, and there was even a tournament-like competition organized by NeurIPS that has different teams trying to build a team of two agents that outperforms the others. The top 2 teams using reinforcement learning were ‘*navocado’* [3] & ‘*skynet955’* [4] placing at 4th and 5th place respectively. Some of the other top teams used tree search approaches in pessimistic scenarios [5].
‘Skynet955’ uses a mixture of action-filter (which restricts actions that the agent is allowed to take in certain scenarios), PPO algorithm, reward shaping (to cope with sparse reward problem) & Curriculum learning (learning slowly in/against increasingly complex environments/opponents).
Our own project does not aim to build a working team of two agents, but only one agent that performs well against three others.

Problem description

Pommerman

Pommerman is an implementation of the game Bomberman, designed to be used for Multi-Agent Reinforcement Learning projects. The game is a Battle-Royale type scenario, in which one of the four agents in a symmetrical 11x11 grid world wins by outlasting the others. At each step, each agent can choose one of six actions: move in one of the four directions, stay in place, or place a bomb. The board has three different tile types; allowing passage, blocked, and blocked by a wood plank, and is procedurally generated (guaranteeing that the agents can meet in the middle). A placed bomb will explode after a short while, killing agents that are within the blast and converting wood plank tiles into free passage. When wood planks are destroyed, they have a chance to drop one of three power-up items: an extra bomb, a larger blast radius for the bombs, or the ability to kick a bomb away. When an agent uses a move action in a non permitted direction (onto a blocked tile or another agent), it will stay in its current position.

Due to the Battle Royale type game mode, a reward of 1 for winning (or -1 for losing) is only dealt out once the game ends.



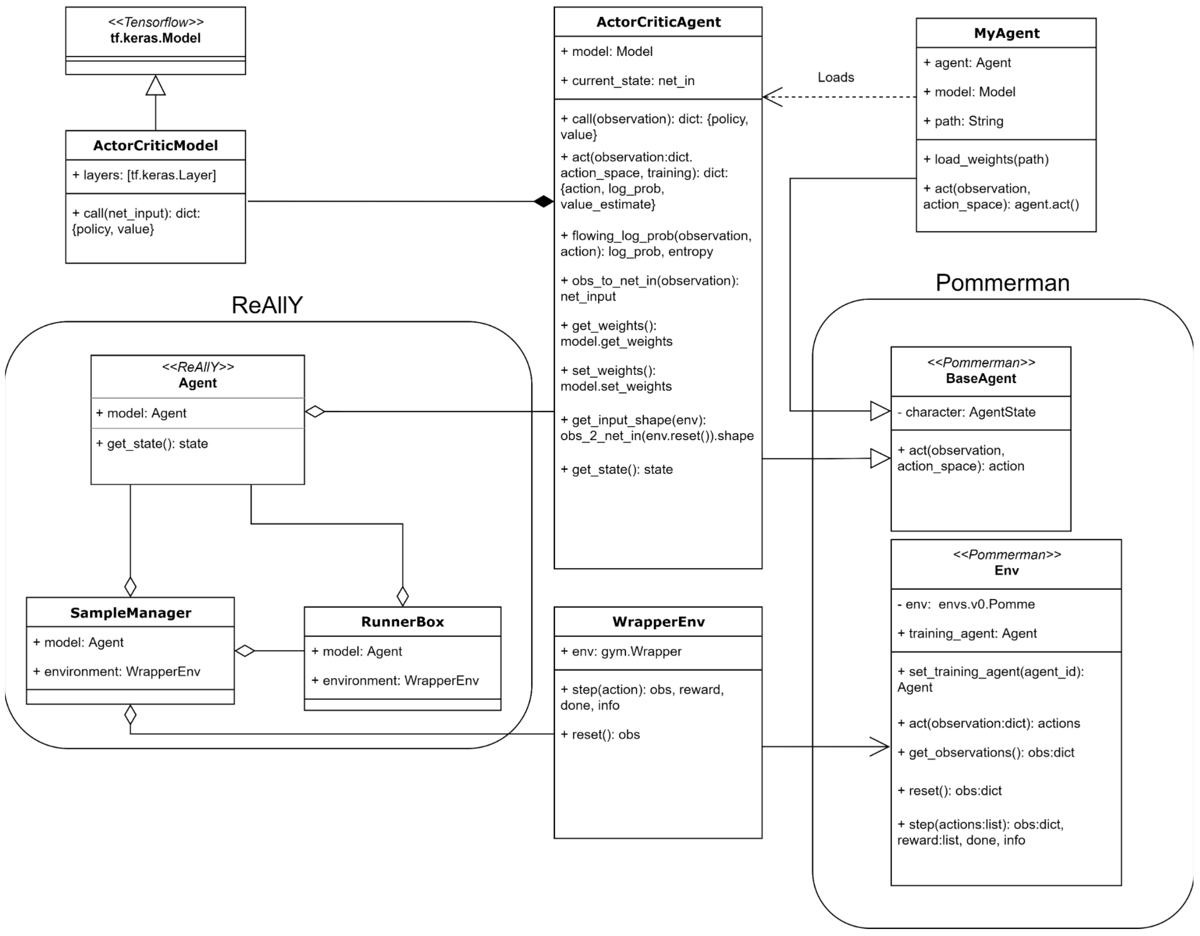
Challenges

The level being a gridworld might make the problem seem easy to solve at first, but there are multiple difficulties when trying to build an agent that performs well using reinforcement learning.
One is the sparse, noisy reward space. Since only one of the four agents wins, only a sequence of actions that wins the entire game will lead to a positive reward, which is very unlikely given the amount of actions that requires. Additionally, the other agents' behaviour can cause unpredictable results, like winning by camping when all others take each other and/or themselves out.
Another one is that the agent not only needs to learn a comparatively complex strategy, they must also overcome obstacles when exploring to acquire the basic skills, like learning to place a bomb even if this will lead to many suicides at first. [6]
In summary, despite taking place in a gridworld with discrete values, this problem is absolutely not trivial to solve.

Architecture

Software

We wanted to use the ReAllY framework provided in this course and combine it with the Pommerman environment. This required many adaptations as the Pommerman environment expects a list of actions for each agent and also returns a list of observations while the ReAllY framework works only with actions and observations of one agent. You can find a class-diagram of our code below.



<<Tensorflow>>
tf.keras.Model
ActorCriticModel
+ layers: [tf.keras.Layer]
+ call(net_input): dict: {policy, value}
ActorCriticAgent
+ model: Model
+ current_state: net_in
+ call(observation): dict: {policy, value}
+ act(observation:dict. action_space, training): dict: {action, log_prob, value_estimate}
+ flowing_log_prob(observation, action): log_prob, entropy
+ obs_to_net_in(observation): net_input
+ get_weights(): model.get_weights
+ set_weights(): model.set_weights
+ get_input_shape(env): obs_2_net_in(env.reset()).shape
+ get_state(): state
Loads
MyAgent
+ agent: Agent
+ model: Model
+ path: String
+ load_weights(path)
+ act(observation, action_space): agent.act()
ReAllY
<<ReAllY>>
Agent
+ model: Agent
+ get_state(): state
SampleManager
+ model: Agent
+ environment: WrapperEnv
RunnerBox
+ model: Agent
+ environment: WrapperEnv
WrapperEnv
+ env: gym.Wrapper
+ step(action): obs, reward, done, info
+ reset(): obs
Pommerman
<<Pommerman>>
BaseAgent
- character: AgentState
+ act(observation, action_space): action
<<Pommerman>>
Env
- env: envs.v0.Pomme
+ training_agent: Agent
+ set_training_agent(agent_id): Agent
+ act(observation:dict): actions
+ get_observations(): obs:dict
+ reset(): obs:dict
+ step(actions:list): obs:dict, reward:list, done, info

You can see that we used a Wrapper Environment in order to pass the required parameters into the Pommerman environment. We also had to make our ActorCriticAgent, which is required to inherit Pommermans BaseAgent, compatible with ReAllYs Agent.

Agent

An agent that should be able to act in the Pommerman environment has to inherit the BaseAgent class of Pommerman. It needs to have an act() method that takes the observation of the agent and returns an action. The agent we build should feed it’s observation into an Actor-Critic Network that then outputs a probability for taking each action. During training the action would then be sampled from this probability distribution while during inference the actions would be selected greedily.

The observation an agent gets from the environment comes in the form of a dictionary. This contains board-sized arrays (eg. the representation of the board and different attributes of placed bombs), as well as single values (eg. amount of own ammo, ability to kick) and iterables (eg. own position, list of enemies).
A slightly shortened printout would look like this:

```
'alive': [10, 11],
'board':
array([[ 0,  1,  1,  0,  2,  1,  2,  2],
       [ 1,  0,  0, 10,  0,  2,  2,  1],
       [ 1,  0,  0,  1,  2,  0,  0,  2],
       [ 0,  0,  1,  0,  1,  0,  0,  0],
       [ 2,  0,  2,  1,  0,  0,  0,  2],
       [ 1,  2,  0,  0,  0,  0,  0,  1],
       [ 2,  2,  0,  0, 11,  0,  0,  1],
       [ 2,  1,  2,  0,  2,  1,  1,  0]]),
'bomb_blast_strength':
array([[0., ..., 0.],
       [0., ..., 0.],
       [0., ..., 0.]]),
'bomb_life':
array([[0., ..., 0.],
       [0., ..., 0.],
       [0., ..., 0.]]),
'bomb_moving_direction':
array([[0., ..., 0.],
       [0., ..., 0.],
       [0., ..., 0.]]),
'flame_life':
array([[0., ..., 0.],
       [0., ..., 0.],
       [0., ..., 0.]]),
'game_type': 4,
'game_env': 'pommerman.envs.v0:Pomme',
'position': (1, 3),
'blast_strength': 2,
'can_kick': False,
'teammate': <Item.AgentDummy: 9>,
'ammo': 1,
'enemies': [<Item.Agent1: 11>],
'step_count': 1
```

In order to be able to feed this data into our tensorflow model, we built a function that translates the parts we want to make use of into the desired data structure. We tried two approaches to this problem:

- Flatten the arrays and appending the different values to form one big feature vector
- Creating board-sized arrays out of the single values and iterables and stack those together to form a 3 dimensional array containing feature planes

Model

We built two models, one purely building upon fully-connected layers and another one which used convolutional layers.

Dense

The Dense-Model gets as input the feature vector we created. It has three fully-connected layers as read-in layers and a policy- and a value-head. The policy-head consists of another two fully-connected layers where the last layer has six units to output the probabilities for each action. The value-head also consists of two fully-connected layers but has only one unit in its last layer to estimate the state-value.

CNN

The CNN-Model gets the stacked feature planes as its input. Its read-in block is made up of four convolutional layers. Its output is then fed into a policy- and a value-head. These heads have an additional convolutional layer whose output is then flattened and fed into the output layers which have again 6 units in the policy head and one unit in the value head.

Training

Naive Approach

After getting Pommerman and ReAlly to work together, we first tried out a regular naive implementation. We quickly found that the reward space as it was made learning difficult to impossible. Our agent became stuck in a local optimum of not moving at all, given that this would occasionally lead to a win. A more complex sequence of placing bombs near opponents was already impossible since our agent never learned how to place a bomb without getting blown up.

Reward Shaping

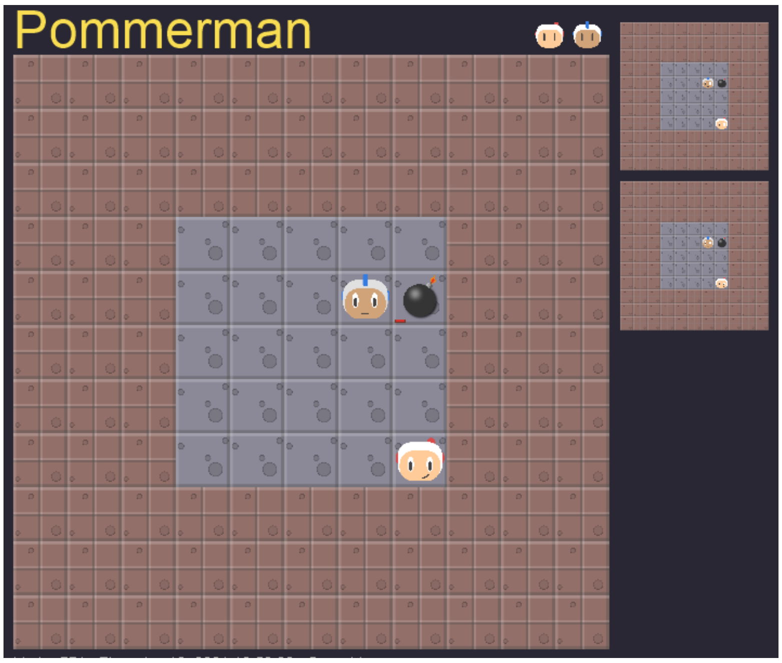
We then tried to solve this by shaping the rewards, for example by introducing rewards for exploring new positions, placing bombs, destroying wood blocks, collecting power-ups and blasting enemies. However, no combination of rewards and penalties we tried out convinced the agent to learn to place a bomb and avoid the blast, because in the small free space, the agents are initially spawned in, it’s still very unlikely to randomly place a bomb *and* avoid the blast during exploration, so the risk when placing a bomb was still to high.

Curriculum learning

To make the agent learn these basics that were causing so much trouble, we decided to use a curriculum approach. This was also based on the fact that we wanted to avoid hardcoded decision making, which other solutions employed.
We created increasingly more challenging environments two slowly teach our agent how to play the game.

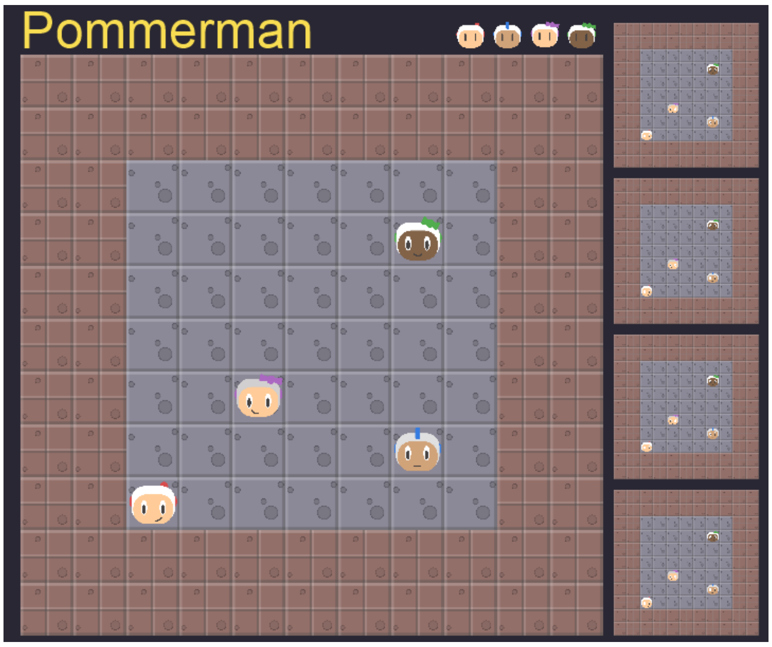
Train_1

At first we decreased the available space on the board to a 5x5 obstacle free space and placed just one non-moving enemy(blue) in it with different starting positions. In this environment our agent(red) consistently wins the game after 65 epochs.



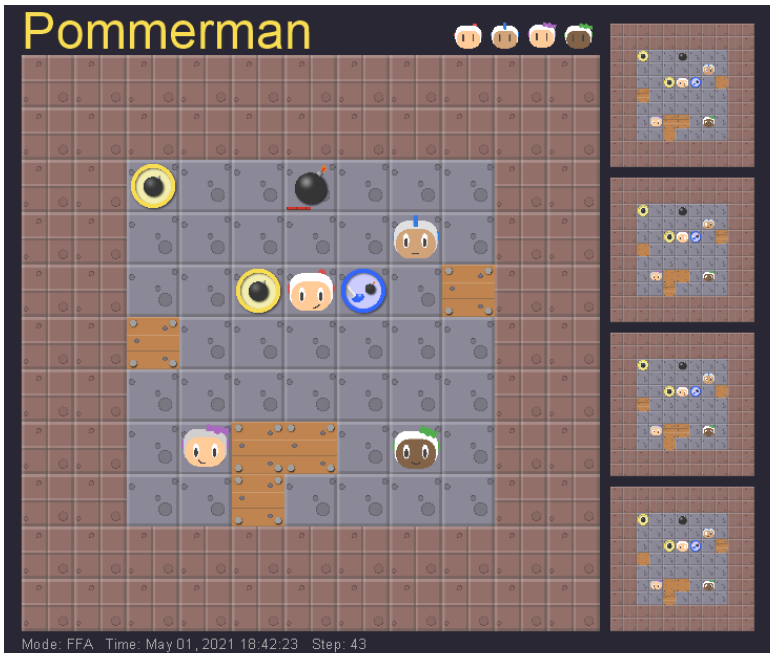
Train_2

The second environment is a bigger empty grid (7x7) where we placed 3 enemies. To increase the stochasticity slightly and to encourage our agent to learn to place bombs depending on the enemies positions, those enemies randomly move for the first 5 steps.



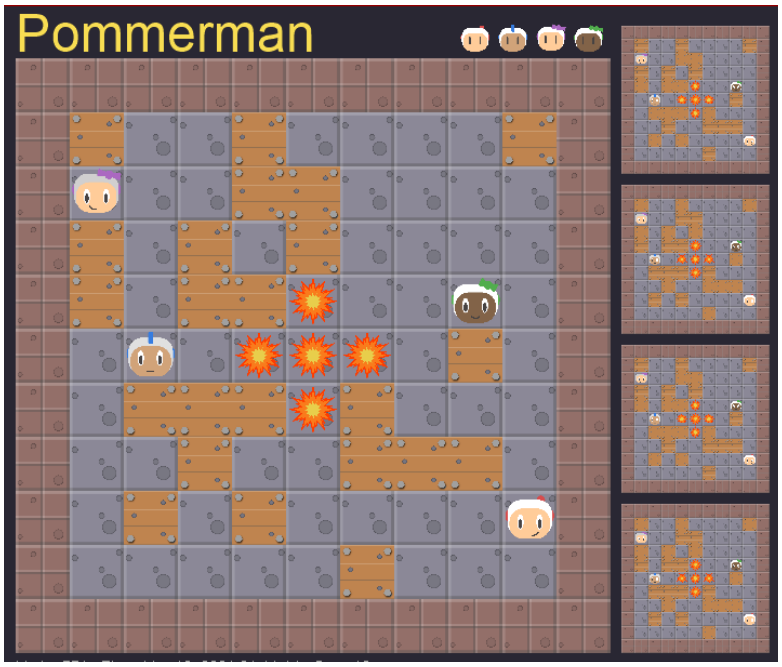
Train_3

In the third environment we introduced destroyable wood blocks that are guaranteed to drop items when getting destroyed. In combination with the shaped rewards our agent should learn to destroy wood blocks and pick up items. Again the enemies are moving in the beginning of the game.



Train_4

In this environment we increase the size again to a 9x9 grid and add more wood blocks. This time not every wood block will drop an item.



After placing the agent we trained in the Train_1 environment in another environment it turned out it simply learned in which positions it should place a bomb to guarantee a win but these positions were not related to the enemy's position.

Action Filter

After seeing that our curriculum learning approach didn't yield the desired results, we included an action filter after all. We used the action pruning module that BorealisAI provides as a baseline after using it for their skynet team. [7] The action filter constrains the action space significantly & thus also making exploration faster & more efficient.
There are two main components to this filter:

Avoiding Suicide:

1. Not going to positions that are flames on the next step.
2. Not going to doomed positions, i.e., positions where if the agent were to go there, the agent would have no way to escape. For any bomb, doomed positions can be computed by referring to its *bomb_blast_strength*, *bomb_life*, *bomb_moving_direction* and *flame_life*, together with the local terrain.

Bomb Placement:

1. Not place bombs when a teammate is close, i.e., when their Manhattan distance is less than their combined blast strength.
2. Not place bombs when the agent's position is covered by the blast of any previously placed bomb

Evaluation

To evaluate the performance of an agent we wrapped our ActorCriticAgent in another MyAgent class which loads the latest model saved under /pommerman/trained_model. We expanded the run_battle function that the pommerman environment provided to run multiple battles and summarize the overall results. It can be called from the terminal (after installing the pommerman environment) as follows:

```
pom_battle
--agents=test::agents.MyAgent,simple::null,simple::null,simple::nu
ll --config=PommeFFACompetition-v0 --render --num_times=5
```

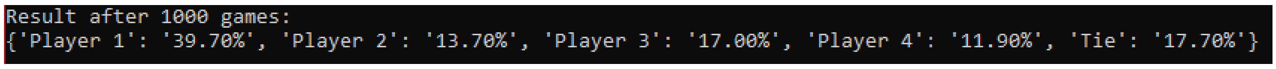
This starts 5 games in which the trained agent competes against 3 SimpleAgents.

Conclusion

Performance

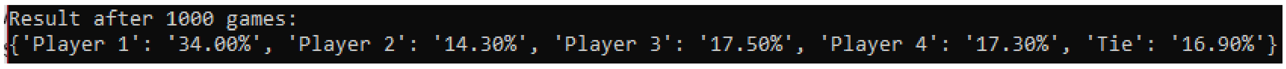
After our attempts at training our agent and looking at the learned action probabilities, we suspected that an agent, that utilized the same action filter and would place bombs whenever possible and otherwise randomly choose an action the filter allows, wouldn't perform significantly worse than the trained agent. To test this we let both, our agent and the described 'RandomBomber', compete against 3 SimpleAgents (Players 2 - 4) in 1000 games.

RandomBomber (Player 1):



```
Result after 1000 games:
{'Player 1': '39.70%', 'Player 2': '13.70%', 'Player 3': '17.00%', 'Player 4': '11.90%', 'Tie': '17.70%'}
```

Our Agent (Player 1):



```
Result after 1000 games:
{'Player 1': '34.00%', 'Player 2': '14.30%', 'Player 3': '17.50%', 'Player 4': '17.30%', 'Tie': '16.90%'}
```

The results show the RandomBomber performs in fact slightly better than our trained Agent.

In Comparison

Looking at other attempts at solving this problem, our agent does not deliver any outstanding results. When equipped with the action filter it manages to perform at a similar level as other solutions. However, even a random agent equipped with the action filter from Borealis AI outperforms the pommerman simple agent. A simple agent without placing bombs also scored 7th place at NIPS 2018 (where Borealis AI scored 5th). In other words, pommerman is a problem where even the best Reinforcement Learning solutions have not overcome the inherent difficulties yet.

Adopting the action filter brought our solution to a competitive level but it also means it is no longer a model free reinforcement learning system, and most(if not all) of its behaviour will be decided by predefined rules.

References

[1] C. Resnick, W. Eldridge, D. Ha, D. Britz, J. Foerster, J. Togelius, K. Cho, and J. Bruna. (2018). Pommerman: A multi-agent playground. arXiv preprint arXiv:1809.07124.

[2] Pommerman environment on GitHub:
https://github.com/MultiAgentLearning/playground/tree/master/pommerman

[3] Peng Peng, Liang Pang, Yufeng Yuan, and Chao Gao. (2018). Continual Match Based Training in Pommerman: Technical Report. arXiv preprint arXiv:1812.07297.

[4] Gao, C., Hernandez-Leal, P., Kartal, B., & Taylor, M. E. (2019). Skynet: A top deep RL agent in the inaugural pommerman team competition. arXiv preprint arXiv:1905.01360.

[5] Osogami, T. & Takahashi, T.. (2019). Real-time tree search with pessimistic scenarios: Winning the NeurIPS 2018 Pommerman Competition. Proceedings of The Eleventh Asian Conference on Machine Learning, in PMLR 101:583-598
http://proceedings.mlr.press/v101/osogami19a/osogami19a.pdf

[6] Gao, C., Kartal, B., Hernandez-Leal, P., & Taylor, M. E. (2019, October). On hard exploration for reinforcement learning: A case study in pommerman. In Proceedings of the AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment (Vol. 15, No. 1, pp. 24-30). https://ojs.aaai.org/index.php/AIIDE/article/view/5220/5076

[7] Skynet action filter on GitHub: https://github.com/BorealisAI/pommerman-baseline/