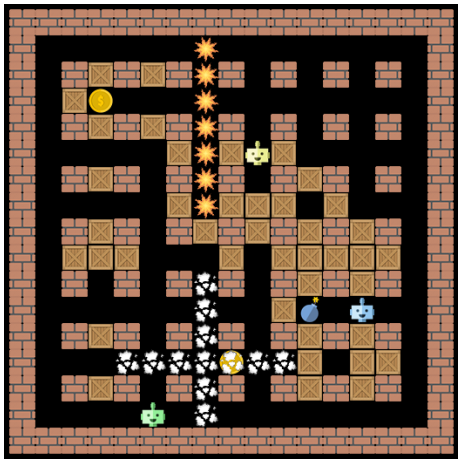
Final Project

Reinforcement Learning for Bomberman

Deadline: 25.3.2019



In this year's final project you will use reinforcement learning techniques to train an agent to play the classical game Bomberman.

In our setting, the game is played by up to four agents in discrete, but simultaneous time steps. Your agent can move around, drop bombs or stand still. Crates can be cleared by well-placed bombs and will sometimes drop coins, which you can collect for points. The deciding factor for the final score, however, is to blow up opposing agents, and to avoid to get blown up yourself. To keep things simple, special items and power-ups are not available in this version of the game.

After the project deadline, we will hold a tournament between all trained agents with real prizes. Tournament performance will be a factor in the final grade as well, although the quality of your approach (as described in the report and code) will carry more weight.

Regulations

Submission

Your submission for the final project should consist of the following:

- The agent code to be used in the competition, including all trained weights, in a subdirectory of `agent_code` (see details below).
- A PDF report of your approach. Aim for a length of about 10 pages per team member and indicate after headings who is responsible for each subsection.
- The URL of a public repository containing your entire code base, which must be mentioned in the report.

Solutions not involving machine learning will be rejected. Zip all files into a single archive with naming convention (sorted alphabetically by last names)

`lastname1-firstname1_lastname2-firstname2_final-project.zip`

or (if you work in a team of three)

`lastname1-firstname1_lastname2-firstname2_lastname3-firstname3_final-project.zip`

and upload it to Moodle before the given deadline.

Development

As will be detailed later on, your agent's code will be run in its own separate process. To not interfere with interprocess communication, and to share resources fairly between competing agents, you are not allowed to use multiprocessing in your final agent. However, multiprocessing during training is perfectly fine.

We will distinguish classical and neural-network solutions. The choice among the two is entirely up to you. Neural networks will be executed on the CPU during official games, but you may use GPUs during training. If agent performance between these approaches differs too much, we will play the tournament in two separate leagues.

Playing around with the provided framework is allowed, and may in fact be necessary to facilitate fast training of your agent. However, be aware that the final agent code will simply be plugged into our *original* version of the framework – any changes you made to other parts will not be present during games.

Discussions with other teams are very much encouraged, and trained agents (*not* training code!) may be exchanged among teams to serve as sparring partners. Just keep in mind that in the tournament you will compete for prizes, so you may want to keep your best ideas to yourself :)

The internet provides plenty of material on all aspects of reinforcement learning. Study some of it to learn more about RL and get inspiration. The use of free software libraries (e.g. `pytoch`) is allowed, if they can be installed from an official repository like `pip` or `conda`, but you must not copy-paste any existing solution in whole or in part. Plagiarism will lead to a "failed" grade.

1 Setup

You can find the framework for this project on `Github`:

```
git clone https://github.com/ukoethe/bomberman_rl
```

It contains the game environment with all APIs you need for reinforcement learning, as well as example code for a simple rule-based agent that does not learn anything. Running `main.py` from the command line should let you watch a set of games between four of these simple agents. Direct any questions about the framework to Jakob Kruse (`jakob.kruse@iwr.uni-heidelberg.de`).

As a game engine we are using the `pygame` package, which you can install into your existing `conda` environment using `pip`:

```
pip install pygame
```

Other than that, we assume a standard Python 3 installation with `numpy`, `scipy` and `sklearn`. Clearly indicate at the beginning of your report which additional libraries we need to install.

2 General setting and rules of the game

The game is played in discrete steps by one to four agents, who are represented by robot heads in different colors. Because of the random elements, multiple episodes of the game will be played to determine a winner by total score. In one step of a game episode, each robot can either move one tile horizontally or vertically, drop a bomb or wait. Movement is restricted to empty (i.e. black) tiles – stone walls, crates, bombs and other agents block movement. Coins are an exception, as they are collected when moving onto their tile. While the placement of stone walls is the same for every round, the distribution of crates and the hiding places of coins differ each time. Agents always start in one of the board's corners, but it is randomly determined which.

Once a bomb is dropped, it will detonate after four steps and create an explosion that extends three tiles up, down, left and right (so you can just outrun the explosion). The explosion destroys crates

and agents, but will stop at stone walls and does not reach around corners. It lingers for two time steps, and agents running into it during that period are still defeated. Agents can only drop a new bomb after their previous one has exploded.

A fixed number of coins is hidden at random positions in each episode. When the crate concealing a coin is blown up, the coin becomes visible and can be collected. Collecting a coin gains the agent one point, while blowing up an opponent is worth five points.

Every episode ends after 400 steps. The agent taking the highest average thinking time during an episode is deducted one point to encourage efficient implementations. In addition, there is a fixed time limit of 0.5 seconds[1] per step for agents to arrive at their decisions. After this time, tardy agent processes are interrupted and whatever actions they have picked by this time are performed.

The exact numbers used for all these rules can be found in `settings.py`. They may be subject to change until seven days before the deadline, in case you inform us about major problems with the current settings. You will be notified if any of the rules are adapted.

3 Tasks your agents will have to solve

We are trying this project setting for the first time and don't know how difficult learning of the full game will be. We thus define preliminary tasks to help your method evolve from simple to complex and ease debugging. The tasks are subsets of each other, so an agent that can handle task 3 should also be able to solve 1 and 2. Configuration instructions for these tasks are given in section 5.

1. On a game board without any crates, collect a number of revealed coins as quickly as possible. This task does not require dropping any bombs. The agent should learn how to navigate the board efficiently.
2. On a game board with randomly placed crates, find all hidden coins and collect them within the step limit. The agent must drop bombs to destroy the crates. It should learn how to use bombs without killing itself, while not forgetting efficient navigation.
3. On a game board with crates, hold your own against one or more opposing agents and fight for the highest score.

4 Framework structure and interface for your agent

In reinforcement learning, we typically distinguish between the agent and the environment it has to interact with. Let us start with the environment here.

Environment

The game world and logic is defined in `environment.py` in the class `BombeRLeWorld`. It keeps track of the board and all game objects, can run a step of the game, start a new round and render everything to the screen. What's most interesting for you is that it keeps track of the agents playing the game via objects of the `Agent` class defined in `agents.py`. In addition to position, score etc., each `Agent` object contains a handle to a separate process which will run your custom code. This process is also defined in `agents.py` as `AgentProcess`.

Agent

The agent process runs a loop that interacts with the main game loop. Before it enters the loop, it imports your custom code from a file called `callbacks.py` within a subdirectory of `agent_code`. Your script must provide two functions which will be called by the agent process at the appropriate times:

[1] Assume that your agent will have exclusive access to one core of an `Intel i7-8700K` processor and can use up to 8 GB of RAM when we run the tournament.

The function `setup(self)` is called once, before the first round starts, to give you a place to initialize everything you need. The `self` argument is the same as in an instance method – a persistent object that will be passed to all subsequent callbacks as well. That means you can assign values or objects to it which you may need later on:

```
def setup(self):
  self.model = MyModel()
  ...
  self.model.set_initial_guess(x)
```

The function `act(self)` is called once every step to give your agent the opportunity to figure out the best next move. The available actions are `'UP'`, `'DOWN'`, `'LEFT'`, `'RIGHT'`, `'BOMB'` and `'WAIT'`, with the latter being the default. To choose an action, simply assign the respective string to `self.next_action` within your `act(self)` implementation:

```
def act(self):
  ... # think
  self.next_action = 'RIGHT'
  ... # think more, if time limit has not been reached
  self.next_action = 'BOMB'  # seems to be better
  ... # you can update your decision as often as you want
```

Once your agent is sure about the best decision, simply return from `act(self)`. If `act(self)` doesn't return within the time limit, the game engine will interrupt its execution. In either case, the current (possibly default) value of `self.next_action` will be executed in the next step (this strategy is generally called "any-time algorithm").

In order to make informed decisions on what to do, you need to know about the agent's environment. The current state of the game world is stored within each agent before `act(self)` is called. It is a dictionary, which you can access as `self.game_state`, and has the following entries:

`'step'`	The number of steps in the episode so far, starting at 1.
`'arena'`	A 2D `numpy` array describing the tiles of the game board. Its entries are 1 for crates, -1 for stone walls and 0 for free tiles.
`'self'`	A tuple (x, y, n, b) describing your own agent. x and y are its coordinates on the board, n its name and $b \in \{0, 1\}$ a flag indicating if the `'BOMB'` action is possible (i.e. no own bomb is currently ticking).
`'others'`	A list of tuples like the one above for all opponents that are still in the game.
`'bombs'`	A list of tuples (x, y, t) of coordinates and countdowns for all active bombs.
`'explosions'`	A 2D `numpy` array stating, for each tile, for how many steps an explosion will be present. Where there is no explosion, the value is 0.
`'coins'`	A list of coordinates (x, y) for all currently collectable coins.

When the game is run in training mode, there are two additional callbacks: `reward_update(self)` is called once after each but the final step for an agent, i.e. after the actions have been executed and their consequences are known. This information is needed to collect training data and fill an experience buffer. The other callback is `end_of_episode(self)`, which is very similar to the previous, but only called once per agent after the last step of a episode. This is where most of your learning should take place, as you have knowledge of the whole episode. Note that neither of these functions has a time limit.

In both callbacks, `self.events` will hold a list of game events that transpired in the previous step and are relevant to your agent. You can use these as a basis for auxilliary rewards and penalties to speed-up training. They will not be available when the game is run out of training mode. All available events are stored in the constant `e` which you can import from `settings.py`:

```
from settings import e
```

They are defined as follows:

`e.MOVED_LEFT`	Successfully moved one tile to the left.
`e.MOVED_RIGHT`	Successfully moved one tile to the right.
`e.MOVED_UP`	Successfully moved one tile up.
`e.MOVED_DOWN`	Successfully moved one tile down.
`e.WAITED`	Intentionally didn't act at all.
`e.INTERRUPTED`	Got interrupted for taking too much time.
`e.INVALID_ACTION`	Picked a non-existent action or one that couldn't be executed.
`e.BOMB_DROPPED`	Successfully dropped a bomb.
`e.BOMB_EXPLODED`	Own bomb dropped earlier on has exploded.
`e.CRATE_DESTROYED`	A crate was destroyed by own bomb.
`e.COIN_FOUND`	A coin has been revealed by own bomb.
`e.COIN_COLLECTED`	Collected a coin.
`e.KILLED_OPPONENT`	Blew up an opponent.
`e.KILLED_SELF`	Blew up self.
`e.GOT_KILLED`	Got blown up by an opponent's bomb.
`e.OPPONENT_ELIMINATED`	Opponent got blown up by someone else.
`e.SURVIVED_ROUND`	End of round reached and agent still alive.

5 Putting it all together

In order to train a new agent, you must create a subdirectory within `agent_code` with your agent's name – this name will also identify your agent during the tournament. Let's go with "my_agent" as an example. Within `agent_code/my_agent/`, you put your script `callbacks.py` as described above. Other custom files, such as trained model parameters, must also be stored in this directory!

You can put the agent into the game by passing its name to the game world's constructor in `main.py`. To have it play against two instances of our rule-based example agent and one agent that chooses random actions, change the code like this:

```
world = BombeRLeWorld([
          ('my_agent', True),
          ('simple_agent', False),
          ('simple_agent', False),
          ('random_agent', False)
        ], save_replay=False)
```

The Boolean values after each agent name indicate whether this agent will be run in training mode, i.e. whether its `reward_update(self)` and `end_of_episode(self)` functions will be called. You can also specify several agents of your own, or even the same agent multiple times, in order to train your agents by self-play. If you choose this strategy, it may be helpful to add a random element to the choice of actions to avoid quick convergence to a bad local optimum.

For efficient training, you should of course turn off the graphical user interface. To do so, simply change the value of `'gui'` in `settings.py` to `False`. The game will then skip the rendering and run as fast as the agents can make decisions. Many of the other settings should not be changed, but you can safely play around with the following:

`'update_interval'`	Minimum time for a step of the game, in seconds.
`'turn_based'`	If `True`, wait for a key press before executing each step.
`'n_rounds'`	Number of episodes to play in one session.
`'save_replay'`	If `True`, record the game to a file in `replays/` (see below).
`'make_video_from_replay'`	If `True`, automatically render a video during a replay.
`'crate_density'`	What fraction of the board should be occupied by crates.
`'max_steps'`	Maximum number of steps per episode.
`'stop_if_not_training'`	If `True`, end episode as soon as all agents running in training mode have been eliminated.
`'timeout'`	Maximum time for agents to decide on an action, in seconds.
`'log_agent_code'`	Logging level for your agent's code, see below.

These settings can be accessed from the code by again importing from `settings.py`:

```
from settings import s
print(s.timeout)
```

To test an agent on the tasks 1 or 2 outlined in section 3, pass *only* that agent to the game world's constructor in `main.py`. For task 1, additionally set `'crate_density'` in `settings.py` to 0.

6 Some hints to get you started

The `self` object passed to your callback functions comes with logging functionality that you can use to monitor and debug what your agent is up to:

```
self.logger.info(f"Choosing an action for step {self.game_state['step']}...")
self.logger.debug("This is only logged if s.log_agent_code == logging.DEBUG")
```

All logged messages will appear in a file named after your agent in your subdirectory. Continuing with the example from earlier, that would be `agent_code/my_agent/logs/my_agent.log`.

`agent_code/simple_agent/` contains code for a rule-based agent that plays the game reasonably well. Look at this agent's `act(self)` callback to see an example of how reading the game state, logging and choosing an action are done. You can adapt this agent as a training opponent, or even use it to create training data while your own agent is still struggling.

If you have `s.save_replay` enabled, a new replay file will be saved in `replays/` for each episode. To watch it back, change the world initialization in `main.py` to

```
world = ReplayWorld('replay_filename_without_extension')
```

and run the game like normal. You can watch the playback at a step speed and `fps` different from the original game by changing the values in `settings.py`. If `s.make_video_from_replay` is enabled and you have `ffmpeg` installed with `libx264` or `libvpx-vp9` codecs, what you see will automatically be encoded as a video file and saved to `screenshots/`.

There is also an option to control one of the agents via keyboard. To do so, change one of the agents passed to the game world to

```
('user_agent', False)
```

and use the arrow keys to move around, `Space` to drop a bomb and `Return` to wait. This is clearly not an efficient way to create training data for your agent, but helps you develop an intuition for the game's behaviour. This might, for example, be useful for designing auxilliary rewards or learning curricula.

7 Report and code repository

Put all your code into a public `Github` or `Bitbucket` repository and include the URL into your report. The zip-file to be uploaded on Moodle shall only contain the subdirectory of `agent_code` containing your fully trained player, along with the report.

You report should consist of roughly ten pages per team member, not counting title page etc. The first section shall describe the reinforcement learning method and regression model you finally implemented, including all crucial design choices. You may also describe approaches you tried and abandoned later, including the reasons. The second section should decribe your training process, including all tricks employed to speed it up (e.g. self play strategy, design of auxilliary rewards, prioritization of experience replay and so on). The third section shall report experimental results (e.g. training progress diagrams), describe interesting observations, and discuss the difficulties you faced and how you overcame them. The final section shall give an outlook on how you would improve your agent if you had more time, and how we can improve the game setup for next year.