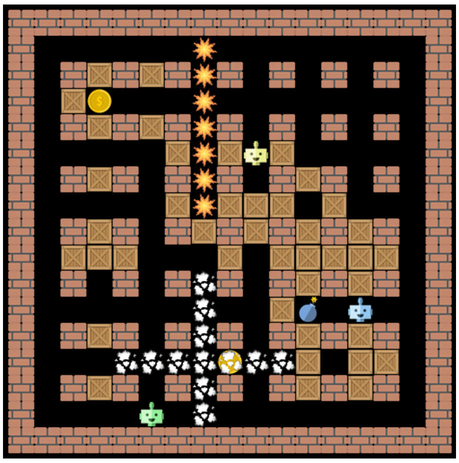
© U. Köthe, J. Kruse, F. Draxler
ullrich.koethe@iwr.uni-heidelberg.de

Final Project

Reinforcement Learning for Bomberman

Deadline: Mon, 29.03.2021, 16:00.
Ask questions to #final-project-questions



In 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 four agents in discrete 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 getting 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.

This is an open project. You may use any combination of models and techniques you have learnt during the semester. You are also free to extend your knowledge to new domains and use other/more advanced models and techniques. There is only one rule: Your solution **must involve machine learning or it will be rejected**.

1 Regulations

Submission

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

- A directory containing your agent code, including all trained parameters. This is the subdirectory of `agent_code` that you are developing your agent in (see details below).
- A PDF report of your approach. Aim for a length of about 10 pages per team member and – for leagel reasons – 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. Please **do not upload your report** to this repository.

Zip all files into a single archive with naming convention (sorted alphabetically by first names)

`firstname1-lastname1_firstname2-lastname2_final-project.zip`

or (if you work in a team of three)

`firstname1-lastname1_firstname2-lastname2_firstname3-lastname3_final-project.zip`

and upload this file to Moodle before the given deadline.

You are allowed to switch groups between the homework assignments and this final project. Please organize yourself, e.g. via #homework-team-finding.

Development

To share resources fairly between competing agents, you are not allowed to use multiprocessing in your final agent. However, multiprocessing or anything else that comes to your mind to improve *training* is perfectly fine.

If you choose to learn and use neural networks, this is okay. Take into account that neural networks will be executed on the CPU during official games, although you may use GPUs during training. Also beware that your agent may not be competitive when network training takes longer than expected and has not converged by the deadline.

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 about the final project with other teams are very much encouraged. You can share your trained agents (*without training code*) on #final-project-beat-my-agent and download other teams' agents to test your approach. 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 lectures from February 17th to 26th will explain the necessary basics of reinforcement learning. In addition, the internet provides plenty of material on all aspects of RL. Study some of it to learn more 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.

2 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,
- a strong rule-based and very weak random agents that do not learn anything,
- a template for your own agent.

Direct any questions about the framework to Felix Draxler on #final-project-questions.

To show the game in a user interface, we are using the `pygame` package. If you do not install `pygame`, the game will still run, but without the GUI. It is also recommended to install the `tqdm` package which gives you feedback how fast the rounds are simulated. You can install both packages in your existing `conda` environment from the exercises:

```
conda activate ml_homework
pip install pygame tqdm
```

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.

You are now ready to watch a few rounds of Bomberman as played by our strong rule-based agent as follows:

```
cd bomberman_rl
python main.py play
```

3 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. There is a fixed time limit of 0.5 seconds[1] per step for agents to arrive at their decisions. Agents exceeding this time will not execute the action picked, but stand still as if they had returned `'WAIT'`. In the subsequent step, the thinking time is reduced by the time excess in the previous step.

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.

4 Tasks your agents will have to solve

Training an agent from scratch that directly masters the game as given is hard. 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 6.

1. On a game board without any crates or opponents, 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 yet without opponents, 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.

[1] Assume that your agent will have exclusive access to one core of an `AMD Ryzen 7 3700X` processor and can use up to 8 GB of RAM when we run the tournament.

3. On a game board with crates, hold your own against one or more opposing agents and fight for the highest score.

5 Framework structure and interface for your agent

In reinforcement learning, we 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 optionally keeps a reference to the GUI. 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 tracking position, score etc., each `Agent` loads your code, located in an underlying agent folder (see below).

This part of the code is fixed during the tournament. For training, you are allowed to perform arbitrary changes to help your agent train, e.g. by changing the board.

Agent

The environment calls your agent's code at certain times in the game loop. Your code must be located in a subdirectory of `agent_code`.

Always loaded: `callbacks.py`

Before running the first game, your evaluation code from a file called `callbacks.py` is imported. This file must contain at least two functions:

```
def setup(self): ...
def act(self, game_state: dict): ...
```

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

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

def act(self, game_state: dict):
    return self.model.propose_action(game_state)
```

Note that some fields of `self` are pre-set:

`logger: logging.Logger`	A logger with which you can print debugging messages (see Section 7).
`train: bool`	This is set to `True` when your model should train. For example, you might want to load a previously trained model when `self.train` is `False` and else create a fresh model to be trained.

The function `act(self, game_state)` 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 if your code exceeds the time limit. Choose an action by retuning the corresponding string from `act(self)` (see above).

The current state of the game world is passed in the dictionary `game_state`. It has the following entries:

`'round': int`	The number of rounds since launching the environment, starting at 1.
`'step': int`	The number of steps in the episode so far, starting at 1.
`'field': np.array(width, height)`	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. Note that we use image coordinates (x, y), so `print(game_state['field'])` will show the board transposed compared to the GUI.
`'bombs': [(int, int), int]`	A list of tuples $((x, y), t)$ of coordinates and countdowns for all active bombs.
`'explosion_map': np.array(width, height)`	A 2D `numpy` array stating, for each tile, for how many more steps an explosion will be present. Where there is no explosion, the value is 0.
`'coins': [(x, y)]`	A list of coordinates (x, y) for all currently collectable coins.
`'self': (str, int, bool, (int, int))`	A tuple $(n, s, b, (x, y))$ describing your own agent. n is the agent's name, s its current score, b is a boolean indicating whether the `'BOMB'` action is possible (i.e. no own bomb is currently ticking), and (x, y) is the coordinate on the field.
`'others': [(str, int, bool, (int, int))]`	A list of tuples like the one above for all opponents that are still in the game.

Loaded only when training: `train.py`

When set to training mode, the environment additionally imports your training code from a file called `train.py` from the same subdirectory. This file must contain the following additional callbacks:

- `setup_training(self)` is called when loading the agent, after calling `setup` in `callbacks.py`. Use this to initialize variables you only need for training.
- `game_events_occurred(self, old_game_state, self_action, new_game_state, events)` is called once after each step except the last. At this point, all the actions have been executed and their consequences are known. Use this callback to collect training data and fill an experience buffer.
- The other callback is `end_of_round(self, last_game_state, last_action, events)`, which is very similar to the previous, but only called once per agent after the last step of a round.

In any of the last two callbacks, your learning should take place. You have the freedom to choose where you see it more useful. Note that neither of these functions has a time limit.

The `events` parameter will hold a list of game events that happened as a consequence of your and the opposing agents' actions. 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 without training mode. All available events are stored in `events.py`:

```
import events as 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.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.
`e.SURVIVED_ROUND`	End of round reached and agent still alive.

These events summarize a lot of the changes between two game states. One key factor in successfully training your agent may be to give useful auxiliary rewards to your agents based on the predefined and your own events.

Summary of callbacks

The following block of code summarizes how the environment calls the agent's callbacks:

```python
import callbacks
callbacks.setup(...)

if train:
    import train
    train.setup_training(...)

for round in range(n_rounds):
    ... # Reset world

    for step in range(n_steps):
        callbacks.act(...)
        ... # Perform agents' actions
        ... # Evaluate world: Bombs, explosions
        if train and not dead:
            train.game_events_occurred(...)
    if train:
        train.end_of_episode(...)
```

Customize your agent

You can customize your agent's appearance in the GUI by adding files `avatar.png` and `bomb.png` in your agent directory. They should be 30x30 pixels large PNG images.

6 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. It is a good idea to start by copying the `tpl_agent` to your own directory. We choose to name it "my_agent" as an example. Within the new `agent_code/my_agent/`, you should find the files `callbacks.py` and `train.py` as described above. Other custom files, such as trained model parameters, must also be stored in this directory!

Run your agent on the command line:

```
python main.py play --my-agent my_agent
```

Now you should see your newly created agent fail against three rule-based agents.

If you want to specify more than one agent to run, call

```
python main.py play --agents my_agent random_agent rule_based_agent peaceful_agent
```

to see your agent compete with a random agent, a rule-based agent, and a peaceful version of the random agent, which does not drop bombs.

You can also create and run 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 training, call the main script with `--train N`: Here, N specifies that the first N agents passed to `--agents` should be trained. If you are using the `--my-agent` parameter, only $N = 0, 1$ run successfully, since the rule based agents cannot be trained.

```
python main.py play --agents my_agent random_agent rule_based_agent
    peaceful_agent --train 1
# or
python main.py play --my-agent my_agent --train 1
```

This automatically shortens the game to run until the last training agent has died. You can prevent this behaviour with `--continue-without-training`.

The game has a number of parameters like the field size, the amount of crates, the timeout limit, etc. They are defined in and can be accessed from the code importing from `settings.py`:

```
import settings as s
```

If you change anything, remember that you should test your agent with the original values before submitting it.

7 Practical hints to get you started

To test an agent on the tasks 1 or 2 outlined in section 4, pass *only* that agent to `--agents`. For task 1, additionally set `CRATE_DENSITY` in `settings.py` to 0.

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`. If you find yourself spammed with log messages, decrease the log levels in `settings.py`.

`agent_code/rule_based_agent/` contains code for a rule-based agent that plays the game very 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 choose this agent as a training opponent, or even use it to create training data while your own agent is still struggling. However, your own agent must be trained (not rule-based) to fulfill the project requirements.

If you pass `--save_replay`, a new replay file will be saved in `replays/` for each episode. To watch it back, call the command as follows:

```
python main.py replay <stored-replay>
```

Both the `play` and `replay` modes of `main.py` receive several options influencing the interface of the game. Be sure to get an overview by running

```
python main.py [re]play --help
```

For efficient training, you should of course turn off the graphical user interface by passing `--no-gui`. The game will then skip the rendering and run as fast as the agents can make decisions.

There is also the option to control one of the agents via keyboard in the GUI. To do so, include the agent named `user_agent` 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. It is useful to combine this agent with `--turn-based`. This will wait for a key press until the next action is executed.

8 Submission test

To make sure your agent does not throw any errors and is compatible with our setup, we offer you pre-runs of your code. For this, upload your solution to Moodle in the format specified in section 1. We will proceed as follows with your uploaded `.zip`:

1. Unzip the submitted file.
2. Search for the first directory in the unzipped files that contains a `callbacks.py`.
3. Copy this directory to our `agent_code`.
4. Run a single game with `self.train = False` against three `random_agents`.

We will then send you the script's output to the console including stack traces, the game logs, and the logs printed to your agent's `logs` folder to the names read off the naming convention.

We will test all submissions uploaded to Moodle before the following times: 15.3.21, 22.3.21, 26.3.21, and 28.3.21, each at 4pm.

9 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 describe 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.