Fundamentals of Machine Learning

Final Project

Applying Reinforcement Learning to Bomberman

Submitted By :
Hein-Erik Schnell
Karl Thyssen

Contents

1 Heins thoughts

Tasks to be done are:

- Find a suitable *state representation* to be then passed to an estimator to estimate the expected reward for each of the possible actions
- Find a suitable rewards in order to communicate the goals of the game to the agent
- Find a suitable model which estimates the expected reward for each possible action at the current state of the game.

1.1 State representation

My proposal is a 2D numpy array. Each row (first index) represents a single state. Each column represents a feature. All available information is stored within each agent in the dictionary `self.game_state`. What features do we store?

- For each cell:
  - **Crate, wall, free** $\{1, -1, 0\}$: The same representation is provided by `self.game_state`. Only it has to be reshaped into a 1D-array.
  - **Cell contains agent** $\{0, 1\}$
  - **Cell contains opponent** $\{0, 1\}$: This and the point above need to be stored seperately because it is possible for agent and opponent to occupy the same cell. It would just be impossible to distinguish whether a cell is occupied by just one or more opponents. But this should be a negligible issue.
  - **Cell contains coin** $\{0, 1\}$
  - **Explosion on cell**: Provided as 2D-numpy array by `self.game_state`.
  - **Danger level** $\{0, \dots, 4\}$: How many time steps until an explosion will hit the cell.
- Just once at the end of the array:
  - **Current step number** $\{1, \dots, 400\}$
  - **Bomb action possible** $\{0, 1\}$

- **Danger level:** This is a danger level for the agent. It is not necessary but provides a clearer measure of whether the agent is actually in danger. I propose to calculate this by multiplicating the danger level of each cell with the *Cell contains agent* entry of each cell and then take the maximum of all the results. This way we will only get a non-zero value if the agent is on a cell with a danger level > 0.
- **Reward already received in this episode**
- **Reward received at the end of the episode**: This one has to be added subsequently at the end of the episode to each ocurred state.
- **Reward gained after this state occurred:** This one is the difference of the two above. This should be our *target* which we aim to maximize.
- **Agent touches opponent** $\{0, 1\}$: The simple agents consider dropping a bomb if they touch an opponent. This may be a very good indicator for dropping a bomb for our model.

This gives us 6 entries for each cell and 7 entries at the end of the array. There are 17×17 cells in the arena. However, the cells located at the rim are always *walls*. These do not need to be stored. What remains is a 15×15 grid. I am not sure whether we want to store cells with *walls* at all, because these are always the same. They should add nothing to the state of the game.
Therefore we have $15 \times 15 = 225$ entries ($15 \times 15 - 7 \times 7 = 176$ if we don't store walls) plus 7 at the end of the array for each state (time step). Storing these total 232(183) entries in a numpy array might produce big amounts of data. We should consider specifying the data type explicitly. However, the numpy documentation states that the data type is chosen as the minimum type required to hold the objects in the sequence.

1.2 Rewards

The most rewarded actions should be those which will win us the game. Those are collecting coins (1 point) and killing opponents (5 points). Therefore, I propose the same scaling between those two when rewarding them.
Subgoals which we consider helpful for winning should be rewarded too, but only with few points. It should be almost impossible to substitute the *winning actions* (coins and killing) with *helpful actions* (destroying crates).
I also propose a penalty of -1 for every action so that the agent learns to act as efficiently as possible.
Dying should be (beware! wordplay:) gravely punished.

Action	Reward
Collect coin	100
Kill opponent	500
Destroy crate	1-2 per crate
Perform action	-1
Die	-500fd

1.3 Estimator

I don't know much about neural networks which is why I spent most of my thoughts on how to implement this with the models we used in the exercises. Plus, I like the idea of the challenge to come up with an agent which doesn't use a neural network. Since we want to estimate the expected reward of performing an action at in a given state, we need a regressor. This regressor needs to be highly flexible in order to cope with the variety of states. I guess the most flexible regressor we used would be a *Random Forest Regressor*. On the one hand, this one would need tons of learning data but on the other hand, which proper regressor doesn't?
This means, we would plug the data of the states and the received rewards into the model and get an estimate of what reward we could expect for what action. Since we need to distinguish between the six possible actions, we need six Random Forests. One for each action. Each forest trained only with the states after which the respective action was performed. This also means that we need to store all data in six different set for six possible actions or store the action performed afterwards in the *state vector*.

1.4 Learning algorithm

I propose to use the Max-Boltzmann (MB) method as described in [2] or something similar. It is mostly a ϵ-greedy algorithm. But when it decides to explore, it doesn't choose randomly among the remaining actions but assigns probabilities corresponding to the value (expected reward) of the remaining action. This way, exploration is not just random but more targeted.

There is no point in letting the agent learn after every single episode. In the long run those learning interruptions will cost us a lot of time. Plus, it is very unlikely that the next episodes are similar to the one before which means that the learning will most likely not be applied in the subsequent episodes. Thus, the learning process itself should be divided into learning intervals of many episodes, say 100 to 1000, maybe even 10000 episodes, given the number of possible states. While we're already

using the term *episodes*, let's call those learning intervals *seasons*.

I propose the following procedure:

1. Use the provided simple agent as our agent to play the first season. I would use the MB algorithm already in the first episode. This already gives us the possibility of exploration which I consider as important since the simple agent is fully determined and hard coded.
2. Use the data of the first season to train our model.
3. From now on, let the agent rely only on the predictions of our model. Create data, explore according to the MB algorithm.
4. Use the data of all former seasons to again train our model.
5. Repeat the two former steps.

The simple agents should be good sparring partners to start with. They are very good at avoiding bombs but sometimes lack the ability to collect coins. We might improve this ability in the simple agent code to get better starting conditions. Maybe by automatically decreasing the distance (and thus increasing importance) of coins.

1.5 Necessary functions

Some functions that need to be implemented in order to prepare everything:

- Calculate danger levels of each cell
- Insert *coins* into *state vector*
- Insert *opponents* into *state vector*
- Insert *self* into *state vector*

We need to decide what structure the *state vector* should have. There are two ways:

1. All features of a concerning cell (coin, opponent, self, danger level,...) subsequently. The all features of the next cell and so on ...
2. All data of one feature (e.g. coin) for all cells, then all data of the next feature for all cells and so on ...

In both cases, the functions that insert the values into the array should make use of the slicing possibilities of numpy arrays (e.g. `x[starting point:end point:stepsize]`). In the first case, only every sixth entry represents the same

feature. In the second case, the first hundred entries represent only one feature for different cells.

Bibliography

[1] Richard S. Sutton, Andrew G. Barto (2018)
Reinforcement Learning: An Introduction
Available at: http://incompleteideas.net/book/bookdraft2018jan1.pdf

[2] Joseph Groot Kormelink, Madalina M. Drugan and Marco A. Wiering (ICAART 2018)
Exploration Methods for Connectionist Q-Learning in Bomberman
Available at: https://bit.ly/2GReSxQ