Fundamentals of Machine Learning

Final Project

Applying Reinforcement Learning to Bomberman

Submitted By :
Hein-Erik Schnell
Karl Thyssen

Contents

1 Learning Method and Regression Model

Hein-Erik Schnell

This section is divided into the three crucial tasks for which we needed to develop a concept in order to get the agent into training. Those tasks were:

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

1.1 State representation

Hein-Erik Schnell

The first task was to choose a suitable state representation. In case of regressors provided by *scikit-learn*, the training data is usually passed to the regressor as a 2D-array where each row (first index) represents a single state and each column (second index) represents a feature of the respective states. Analagously, the prediction then demands an array of similar form. The regressor then returns an array with as many predicted values as there were rows (states) in the input array. If one wants to predict only a single value, one may not pass a 1D-array to the regressor but create an additional dummy dimension. All this means that if we want to use precoded regressors from scikit-learn, we need to find a 1D-array representation for a single state.

After each step, the relevant data is passed to the agent via the dictionary `self.game_state`. In the agents `callbacks.py` we defined the function `create_state_vector(self)` which turns the information provided by `self.game_state` into a 1D-array. We chose to store the relevant features in the following way:

- For each cell:
  - **Agent, Opponent, None** $\{1, -1, 0\}$:
    The dictionary provides the entry `self.game_state['arena']` which is a 2D representation of the game board. We use this to create a numpy-array `agent_state` of the same shape which is 1 on the agents position, -1 on cells with an opponent and 0 on all other cells.

  - **Crate, Coin, Empty/Wall** $\{-1, 1, 0\}$:
    A copy of `self.game_state['arena']` is manipulated in such a way that it is -1 for a crate, 1 for a coin on the respective cell and 0 in all other cases. This would mean that the agent could not distinguish between empty cells and walls. This issue is resolved later when we delete all cells which contain walls. These cells are always the same and therefore do not contribute to the learning process. In our code, this whole part is represented by the variable `loot_state`.
  - **Bombs** $\{6, 5 \dots 2, 1, 0\}$:
    The variable `bomb_state` is of the same shape as `self.game_state['arena']`. All cells are by default 6. If the cell will soon be affected by a bombs explosion, the values $5 \dots 2$ represent the 4-time-steps countdown. $1 \dots 0$ represent the 2-time-steps explosion. This way, `bomb_state` provides a danger level for each cell. 6 means no danger at all.

- Just once (implemented in our code as `extras`):
  - **Current step number** $\{1, \dots, 400\}$:
    `extras[0]` contains the current time step.
  - **Danger level** $\{0, \dots, 6\}$:
    `extras[1]` represents the danger level on the agents current position. It is calculated by $6 -$ `bomb_state[x,y]`, where `x` and `y` are the coordinates of the agents position. Consequently, this danger level in invers to the danger level in `bomb_state`, i.e. 0 means no danger, $1 \dots 4$ means increasing danger and $5 \dots 6$ would be bombs exploding. The least point is rather irrelevant since the agent would already have been deleted by the environment.
  - **Bomb action possible** $\{0, 1\}$:
    `extras[2]` is 1 if the agent could place a bomb and 0 if not (i.e. if an own bomb is still ticking).
  - **Touching enemy** $\{0, 1\}$:
    `extras[3]` is 1 if an opponent is on a neighbouring cell and 0 if not.

After manipulating the data in the described way, all cells containing walls are deleted from the 2D-arrays `agent_state`, `loot_state` and `bomb_state`. As already described above, this is done because these entries will always be the same and therefore never contribute to the learning of the agent. The three arrays are then flattened and concatenated after one another into the 1D-array `vector`. Finally, we append the `extras` to the `vector`, which is then returned by the function `create_state_vector`.

1.2 Rewards

1.3 Regressors

2 Training Process

3 Results

4 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.

4.1 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** $\{0, \ldots, 4\}$: How many time steps until an explosion will hit the cell.

- Just once at the end of the array:
  - **Current step number** $\{1, \ldots, 400\}$
  - **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.

4.2 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

4.3 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*.

4.4 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.

4.5 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.

5 Karls thoughts

How this project can be approached:

- Neural Network approach *vs.* Classical Machine learning approach
  - Supervised learning *vs.* Reinforcement learning *vs.* Unsupervised learning
  - Feature selection to determine relevant information the agent should "see" and learn from
- Data management for training - not necessary as we will be collecting all the relevant data while training rather than using a large pre-compiled dataset

As we do not have access to more than our own laptops which can not be classed as high performance systems with low end dedicated GPUs training neural networks may be impractical and time consuming.

5.1 Neural Networks

As we didn't cover neural networks in the lecture but are allowed to apply them to this Project I explored it as an option despite knowing we are likely to select the classical approach. Some algorithms that are worth exploring are:

- Supervised learning:
  - Gradient descent involves forming a continuous error function using rewards to minimize
- Reinforcement learning:
  - Q-Learning (as covered in lectures)
  - Genetic Algorithm:
    - Training occurs through improvements from generation to generation by improving the strength and bias values for the edges of the neural network (stored in a 1D array). Each generation has a set number of instances that follow a variation of the previous generations values. To decide which of the instances in the previous generation had the best strength and bias values a scoring system must be created based on the performance of each instance.
    - Outputs are predetermined as 5 movement options (up, down, left, right, stand still) and the Bomb placement option
    - Factors to be decided on:
      - Inputs e.g. *weighted score of distance to enemies/crates/coins/bombs*
      - Number of hidden layers and nodes (not sure how this is done yet)
      - Method for genetic variation e.g. *ranking by score and weighting the likelihood for this process to be selected, random selection.* This is the exploitation aspect. Some values however should be randomly changed or multiple arrays crossed together (like nature) to increase the likelihood of finding the global minimum rather than local (exploration.

5.2 Classical approach

We could use Q-learning to train a ϵ-greedy agent. We want to find the optimal policy $\mathbf{Q}$ to solve the game.

Here its important to identify the 2 sets of attributes that contribute to decision making, **state** and **action**. The state will be input to decision making function to determine the action, the features that describe the state however must first be decided on. To reduce runtime it may be advantageous to perform a dimension

reduction if it becomes clear that some features for example the states of the cells outside of a 5x5 radius around the agent have very little impact on the decision. This could be particularly important in the early tests before the final train to reduce training time required when for example testing various reward value boundaries.

State appraisal

Before a decision can be reached as to the next action clearly the state of the board must first be assessed. With an almost infinite number of possible inputs possible it is important to have as few and as relevant features as possible.

Ideas to test/discuss:

- Are all accessible fields relevant
- Is it possible to approximate the state with knowledge about only the non-empty fields. In extreme cases this can be 2 (final 2 agents) or 176 (all fields occupied, although this is a stalemate) so this is likely not constructive.
- How should the distances to other agents or items be stored? Possibilities include:
  - The route that could be taken
  - Only the coordinates
  - The number of moves required to move the the desired opponent or coins field and the direction. Here the shorter route could also be through boxes and therefore include bomb wait times
- Should any previous states have a weighted input? - probably not, could be useful in some edge cases maybe?

Rewards

Naively we want to positively reinforce proactive, safe actions and negatively reinforce hazardous or even fatal actions so we should set our rewards based on this structure. This may however not always lead to the optimal solution as there are some situations where a sacrifice can lead to a greater boon in later steps e.g. *queen sacrifice for checkmate in chess.*

We have essentially 3 choices for how to issue rewards:

- Updating Q after each episode:

  - This leads to faster training times as the entire episode can play out without interruptions, however each episode leads to slower improvement as the Agent can only perceive the results of all decisions made during the episode.

- Stepwise Q updates:
  - Very slow episodes (opposite of episodic updates above)
  - Ultimately it will depend on how we set our rewards, whether a movement to a cell should be rewarded based on proximity to other agents/bombs or coins and how we allow these factors to influence the positivity or negativity of an action. I.e. A step towards a coin that is also a step into an exploding bomb should lead to a negative reward.
  - This would allow for punishment for the repetition of actions, such as the movement to a previously occupied square which may be undesirable and therefore punished with a small negative value
  - Decisions that are not allowed in certain situations such as movement into a wall/crate should also be punished harshly
- N-step Q updates: see above every N steps

Importantly the values for rewards should be small to ensure numerical stability in later operations

Exploration vs. Exploitation

- e-greedy policy

  [[learn to Tex maths]]
- soft max policy to max the "Temperature" S over time

  [[learn to Tex maths]]

6 Karls observations

These are just notes to document stages of development and note questions that arise.

6.1 The state matrix

- How should a bombs presence be documented in the 'state' matrix?
- Should a bomb timer be the input?
- Is there any value in including the fields that have an explosion in them? I can't think of a situation in which this would be relevant to making a decision for the next action.

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