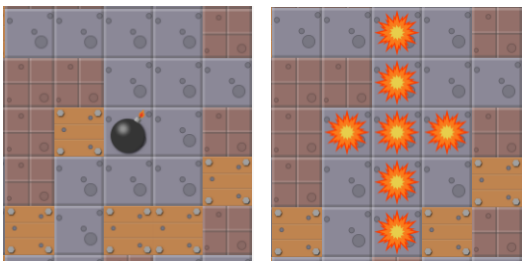
Pommerman Agent

Omri Ben Dov Meirav Segal
Gal Katzhendler Varda Zilberman



(a) Before (b) After

Figure 1: Bomb explosion

I. About the game

In the original bomberman game, 4 players each control a virtual character. Each character can go in 4 direction, drop a bomb or wait. After a few moments the bomb goes off and any character standing vertically or horizontally in the blast radius (shown in fig. 1) loses.

The winner of the game is the last player standing. If the last agents die at the same time, it is a tie.

At the beginning of the game, each player can only drop one bomb. The ammo is renewed after the bomb explodes.

The version of the game we chose to solve is called Pommerman[1]. Pommerman is an internet competition between AI agents sponsored by NIPS, Google AI, Facebook AI Research and more.

A. Differences from original Bomberman

In this variant of the game, the game board is made of 11×11 tiles. As the game starts, each player is placed in a different corner on the board.

There are also a few differences from the original Bomberman:

- There are 2 type of walls:
  - Rigid walls: Agents and flames can't go through. Can't be blown by a bomb.
  - Wood walls: Agents and flames can't go through. Can be blown by a bomb, and when blown has a chance to produce a pickable powerup.

Figure 2: A game in process. The faces are the players, the green object is a powerup. The other objects are self-explanatory.

- Each game's map is randomized. In each game walls (wood and rigid) will be created at different random places.
- Powerups can be picked by the agent who reaches them first and can either:
  - Increase the blast radius of the agent's bomb.
  - Increase the available ammo of the agent.
  - Give the ability to kick bombs. With this ability, an agent can move a bomb it runs into.

B. Game representation

At each turn, the agent gets a list of observables from the game in order to decide how to act. The observables are:

- The board itself.
- The position of the agent.
- Agent's remaining ammo.
- Agent's blast radius.

- Can the agent kick bombs.
- Who are the agent's enemies.
- The strength and time to explosion of each bomb on the boards.

II. Agents

A. Q-learning

We first tried using reinforcement learning. The state of the board at each turn is made of vectors and matrices of a constant size. Consequently, this game can be represented as an MDP:

- The states are the game's states.
- The actions are the possible actions for each player.
- The rewards are given by the game: 1 if won, -1 if lost, 0 if still playing.

Each action can lead to many states as a result of the game being multi-agent, since we can't control the other agents.

As the game can easily be translated to MDP, we could simply implement a Q-learning agent.

For the learning process, we played many games with the learning agent against 3 simple deterministic agents (designed and implemented by the Pommerman team). We chose playing against it since it is a fast and efficient agent.

The learning agent is initialized with some model parameters α, γ (learning rate, discount respectively) and would choose an action accordingly. After every turn a reward is given by the game and the agent would update its q-values according to this reward (r). The update is made by:

$$Q(s,a) \leftarrow (1-\alpha) Q(s,a) + \alpha \left(r + \gamma \max_{a'} Q(s', a') \right)$$

Where s is the previous state, a is the taken action and s' is the state we reached.

We randomized the starting corner of our agent every 50 games.

We also improved our learning efficiency by also learning from the other 3 players in the game by sending their states and rewards to our agent's Q-values.

Finally, we tried different methods to choose the next action and handle the rewards. For each of those, we created an agent and trained each of them using different model parameters.

1) Basic Q-learning: The most simple representation is to use the given game state as is (described in I-B).

For this agent we explored using the ε-greedy method. That is given ε, with probability ε we choose a random action. With probability 1-ε we choose the action with the maximal Q-value. In case of a tie, we choose randomly from the maximal actions.

2) Q-learning with feature extractor: In order to reduce the state space and the training time, we implemented data extractors to extract what we think is relevant and important. The extracted state included:

- Valid directions - for each of the 4 directions, True means that direction is free and on the board and False otherwise.
- Dangerous bombs - for each direction, if there is no bomb in that direction, the value is 0. Otherwise, the value is an int representing the time left for the bomb to explode.
- Adjacent flames - for each direction, if there are no flames in that direction, the value is False, otherwise, True.
- Current quarter - representing which of the 4 quarters of the board the agent is in.
- Enemy in range of our bomb - True or False
- Wood wall in range of our bomb - True or False
- Stands on bomb - True or False
- Powerups in range of two steps for each direction

The reward and training setting were identical to the basic Q-learner. We used ε-greedy exploration here too.

3) Upper Confidence Bound (UCB): A disadvantage to using ε-greedy exploration is that with some probability it choose a random action with a uniform distribution. Therefore, actions that look promising or not certain about won't be picked with a better chance. The UCB exploration method tries to fix that.

It does so by adding a confidence score to the Q-values before choosing the next action. The next action for current state s is taken according to:

$$a = \arg\max_{a'} \left[Q(s,a') + c \cdot \sqrt{\frac{\ln t_s}{N_s(a')}} \right]$$

Where $N_s(a')$ is the number of times we chose action a' at state s and $t_s \equiv \sum_{a'} N_s(a')$ is the total number of times we reached state s. c is the confidence parameter, given to the agent the same way as ε is given to the ε-greedy agent. This agent uses the same extracted features as before.

4) Backplay: One of the problems in reinforcement learning is that it requires a large number of trials to learn a good policy, especially in environments with sparse rewards, such as in the Pommerman environment. The reward is only granted at the end of each game, and since the state space is large it takes a long time for the reward to propagate to q-values of earlier states.

In a recent paper[2], a new method called Backplay for reducing training time for such problems was proposed. In this approach, rather than starting each training episode in the environment 's initial state, we

start the agent near the end of the game and move the starting point backwards during the course of training until we reach the initial state. The paper showed that this guidance provides significant gains in sample complexity in sparse reward environments.

We implemented this concept by saving full states for the entire game. Then, for every window of 5 states starting from the end of the game, we randomly selected one of the states to be an initial state from which we run a learning episode. After each learning episode we shifted the window 5 states backwards and repeated the process until we reached the true initial state. The reward and other training setting were identical to the basic Q-learner.

This agent uses the same extracted features as before.

B. Snorkel

Snorkel is an open-source framework for labeling training data, designed to overcome the largest bottleneck in deploying machine learning systems - getting labeled data. The framework works in the following order[3]:

1) Writing labeling functions: Users write labeling functions that express arbitrary heuristics, which can have unknown accuracies and correlations. In our project the input of the functions is the full list of observables as described in I-B.
2) Modeling accuracies and correlations: Snorkel automatically learns a generative model over the labeling functions, which allows it to estimate their accuracies and correlations. This step uses no ground-truth data, learning instead from the agreements and disagreements of the labeling functions. In some cases the generative model will not get better results than a simple majority vote. For this reason the system also includes an optimizer for deciding when to model accuracies of labeling functions, and when learning can be skipped in favor of a simple majority vote. After this step Snorkel can produce a single, probabilistic label for each data point.
3) Training a discriminative model: Train a machine learning model using a labeled training set. In our project we used Random Forest for this step.

Snorkel is optimized for binary classification, returning the probability for the label to be 1. For this reason we used binary classification for each of the 6 possible actions.

For the labeling process we used the following heuristics:

- Adjacent flames - for each direction, if there are no flames in that direction, the value is False, otherwise, True.
- Valid directions - for each direction, True means that direction is free and on the board and False otherwise.
- Adjacent powerups - for each direction, if there are powerups in that direction, the value is True, otherwise, False.
- Is in range of bomb - for each direction, does moving there puts us in range of a bomb.
- Dead ends - return -1 for directions that will put the agent in a dead end (no passage to any direction except back).
- Enemy adjacent - checks if there is an enemy adjacent to current position. If there is, we return 1 for dropping a bomb and 0 for each other action. If there is no enemy, we return 0 for all actions.
- Wood wall adjacent - return 1 for dropping a bomb if the agent is next to a wood wall.
- Move to enemy - return 1 for the direction that takes us to the closest enemy.
- Move to wood wall - return 1 for the direction that takes us to the closest wood wall.

After Snorkel learned a generative model (step 2) for each of the 6 possible steps, we were able to get classification for a given state. We returned the action for which the probability was maximal between all other actions. Although Snorkel was originally designed for labeling a training set, it can also be used for labeling the test set - in our application the current state when the agent needs to decide on an action.

This agent was trained for 300 games.

1) Random Forest: In addition to using step 2 as a classifier, we also completed the Snorkel pipeline by using a Snorkel labeled training set to train a Random Forest model. The training data did not include the full list of observables, but only the extracted features as described in II-A2.

We used the scikit-learn implementation[4] with 50 trees. This agent was trained for 1,500 games.

C. Monte Carlo Tree Search (MCTS)

Having the problem modeled as an MDP, creating a search tree and optimizing the moves over it seemed like a useful approach. In a tree search we traverse the game tree to derive an evaluation of the best move our agent can make. As traversing the whole tree is computationally very expensive, and as our previous attempts to create domain-specific heuristics did not yield impressive results, we set out to try a variation of Monte-Carlo tree search.

In Monte Carlo tree search the node expansion mechanism is based not on a domain-specific heuristic function, but a general one: The agent simulates node expansions according to the most promising policy, reward (win/loss) at the terminal node, and choosing to expand (i.e. take action) the node with the most promising expected reward calculated this way - that is, the probability of winning when choosing that node.

Notice that this heuristic is different than most of the heuristics we have seen in class. It is not necessarily admissible nor consistent, as the terminal states are sampled randomly. We use the estimate and apply a best-first method. To balance the exploration-exploitation trade-off between the deeper nodes (i.e. not the ones involved with the immediate action) we use UCB as described in II-A3.

This tree search method presents a trade off between accuracy and time-per-step, where more samples create a better estimation of winning using a specific branch while taking more time (less efficient) and vice versa. This trade-off can be especially detrimental to an agent in real-time games such as Pommerman (the way it is usually played), but we tested it without a short time limit per move as it would otherwise significantly limit it's estimation accuracy. Using the slow MCTS agent behavior, we planned to imitate its behavior via deep learning to speed up the decision making, but gave up on the idea as we were lacking time. Another drawback of the method is that using a probabilistic method trying to solve a deterministic game, we might encounter a branch with lots of winning leaves but small amount of losing leaves that an expert opponent can force us into.

III. Analysis

A. Q-learning training and parameters

We first wanted to find the best parameters for Q-learning by training the ε-greedy extractor agent (described in II-A2) with all permutations of these parameters:

α (learning rate)	0.01	0.1	0.2	0.4
γ (discount)	0.8	0.9	0.99	0.999
ε (exploration)	0.05	0.1	0.2	0.5

We chose those high discount values since each game gives a non-zero reward only at the end of the game, after a large number of turns.

We were not as sure for α and ε, so we gave them a larger range.

Each of those $\left(4^3 =\right) 64$ agents was trained for 2,250 games, each time staring in a random position.

The comparison was made by choosing two parameters to be constant - leaving one free parameter - and put all agents with those parameters against each other (24 games for each pair of agents). For each value of the free parameter we graphed its $\frac{\#\text{Wins}}{\#\text{Games played}}$.

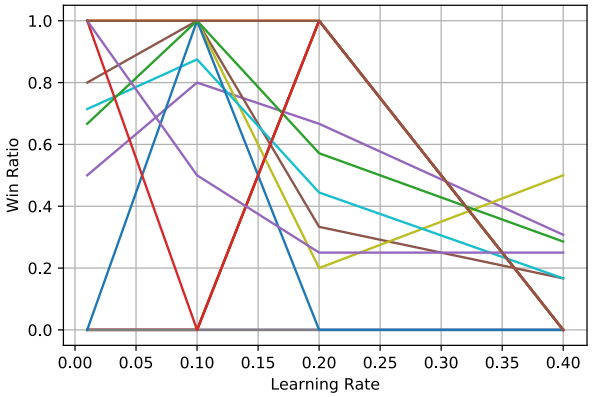


Figure 3: Win ratio as a function of α. Each line represent an agent with different γ and ε.

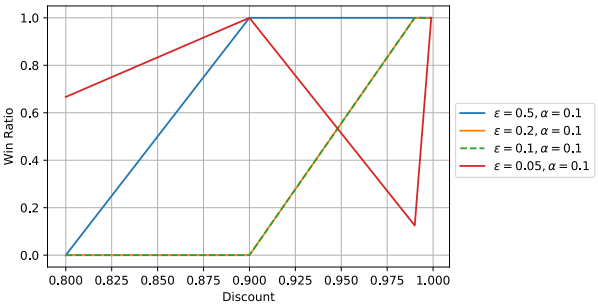


Figure 4: Win ratio as a function of γ with $\alpha = 0.1$. Each line represents a different ε

In Fig. 3 we can see there's a main peak at $\alpha = 0.1$. So we filtered the other α's and graphed the same for γ and ε.

It seems as the best parameters are $\varepsilon = 0.2$ (Fig. 5) and $\gamma = 0.99$ (Fig. 4).

After we found the optimal parameters for the Q-learner, we trained the 4 versions of Q-learners (described in II-A) for 12 hours using these optimal parameters.

We had to train the raw-state agent (II-A1) for a much smaller number of games. This is because it has a huge state space and the file storing its Q-values reached its size limit.

B. Testing

We wanted to compare the different agents with each other and also test them against a smart baseline.

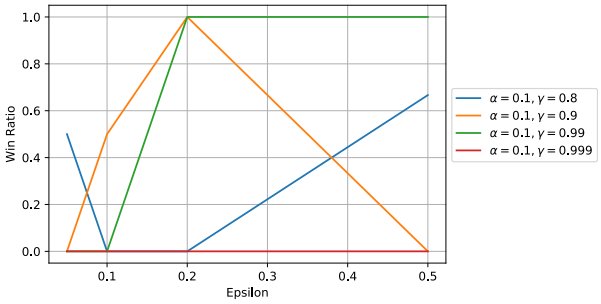


Figure 5: Win ratio as a function of ε with $\alpha = 0.1$. Each line represents a different γ

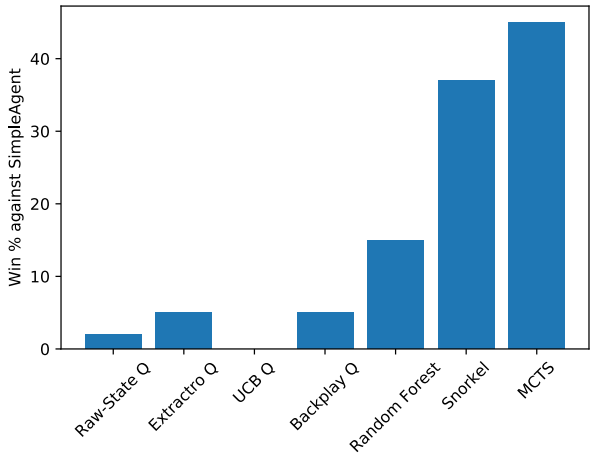


Figure 6: Win % of each agent against SimpleAgent

The Pommerman competition offers a baseline called "SimpleAgent", a deterministic agent that acts according to smart heuristics. When testing this agent against inexperienced human players it wins most of the times[1].

We tested each of our agents against three SimpleAgents for 100 games. The starting position of our agent was randomized every game. For each agent we show the number of games won by our agent divided by the number of games that did not end with a tie (win ratio). If the game took longer than 500 steps we consider it as a tie. For the testing we also defined all Q-learners to have $\varepsilon = 0$ (for ε-greedy) and $c = 0$ (for UCB). The results are shown in Fig. 6.

Since some agents only rarely beat SimpleAgent we needed another method to compare the agents. To do so, we compared our different agents against each other.

[1]We mostly lost in playing it.

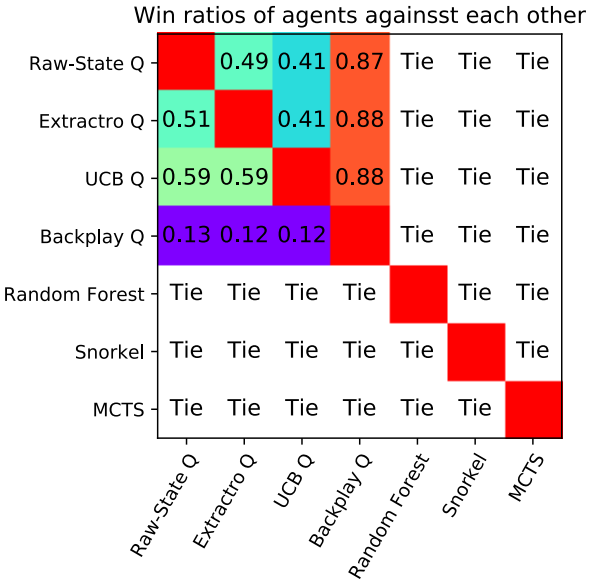


Figure 7: Win ratio of each agent against other agents. The numbers are the the ratio of wins of the agent from the bottom (x axis) playing against the agents from the left (y axis). The sum of two opposite cells w.r.t. the diagonal is 1. The main diagonal is red and empty because those are the agents playing against themselves.

We tested each pair of agents for 100 games. Since the game is played by 4 players we inserted two players of each type in opposite corners of the board (as we did in III-A).

The results are shown in Fig. 7.

In some games we noticed that the agents are inactive and spring into action only when an enemy is getting closer. Therefore, many games ended in ties (the number of turns in a game is limited). From Fig. 7 we learn that the Random Forest, Snorkel and MCTS suffer heavily from this problem.

IV. Discussion

A. *Q-learning*

We expected the raw-state Q-learner to not succeed at all for the reason that it can choose a "good" action only if it has been in the same state before. Since its state space is incredibly large and the maps are randomized, there's virtually no chance it would reach a state it had seen before. It won in a single game against SimpleAgent. We believe it is because the other agents killed each other before. It also surprised us and won against the extractor agent about half the times.

The UCB agent completely failed against SimpleAgent, and lost most of the times against the other Q-learners. A possible reason for it is a wrong choice of the confidence parameter (we used $c = 1$). It might have also needed to train on more games.

From Fig. 7 we also learn that the Backplay agent beats the other Q-learners quite easily. We used this method because otherwise it would take a long time for the rewards to propagate to the early game states. It seems that this method is quite effective.

The general bad performance of the Q-learning agents may results from a large number of reasons. State extractors could be modified and added, more parameters could be tested, the rewards are sparse and training could be run for more games.

Using Q-learning to solve this game has many methods to explore, which we have not implemented.

One such method is to insert artificial rewards. Since the rewards in this game are sparse (only at the end of each game) , we think it is possible to "motivate" the agent to get a powerup or plant a bomb by giving the agent a reward from us, and not from the game directly.

While this could make the agent act more aggressively, it might also hurt it. For example, it might want to get a reward that will put it in some bomb's radius.

Another possible improvement for Q-learning is to use the last few turns to represent a state, and not only the current turn. It will require more complicated extractors, but it brings the agent a wider perspective of the game board.

B. MCTS

The best performer against SimpleAgent was the MCTS agent. Its disadvantage is its runtime. The time it takes for it to act is significantly longer than the other agents, longer than the official Pommerman competition time limit. A possible way to lower its runtime is by using a neural network. We could have also tried to run it with a time limit and see how its win ratio is affected.

C. Snorkel and Random Forest

The Snorkel agent also performs well against SimpleAgent, and relatively faster. Though, we can see that against out other agents, it almost always reaches a tie. As it seems, the heuristics we implemented didn't make it active, but defensive. Consequently, if it plays against other "inactive" agents it would also remain inactive. SimpleAgent almost always moves, thereby getting closer to the Snorkel agent and "activating" it.

The Random Forest agent also shows some potential. The choice of 50 trees was arbitrary, and if we had the time to check more forests with varying number of trees and more time to train, we might have reached better results.

REFERENCES

[1] https://www.pommerman.com/

[2] Resnick, Cinjon, et al. "Backplay: 'Man muss immer umkehren'." arXiv preprint arXiv:1807.06919 (2018)

[3] Ratner, Alexander, et al. "Snorkel: Rapid training data creation with weak supervision." Proceedings of the VLDB Endowment 11.3 (2017): 269-282

[4] Pedregosa, Fabian, et al. "Scikit-learn: Machine learning in Python." Journal of machine learning research 12.Oct (2011): 2825-2830