Final Project for the lecture:

Fundamentals of Machine Learning

Daniel Gonzalez, Matrikel Nr. 3112012

Maria Regina Lily, Matrikel Nr. 3347645

Ferdinand-Joseph Vanmaele, Matrikel Nr. 3443674

25 March 2019

Abstract

Bomberman is a competitive action game where the environment is fully observable, but the strategies of the agents are not. In this project, we wish to develop an agent which plays the game of Bomberman as efficiently as possible. To build such an agent, we use methods of *Reinforcement Learning*. The method we have chosen is Q-learning, with numerical rewards. The initial approach in using Q-learning is using lookup tables to estimate an optimal *action-value* function, which denotes the expected reward $Q(s, a)$ after taking action a in state s. However due to the size of the possible game states in Bomberman, a reasonable good approximation is required. We have chosen *linear function approximation* together with carefully selected *features* of the Bomberman environment.

To make this approximation as close as possible to the real Q values, we have used gradient descent. Due to the strong correlation of experiences from the

same round, this approach may not converge to an optimal value. To alleviate this, we have used *experience replay*, a concept used in Deep Learning, as well as *residual gradient descent*.

The complete code from our team can be found at: `https://github.com/mlteam-ws2018/RL_boom`

1 Reinforcement learning

Reinforcement learning methods are computational methods that allow an agent to learn from its interaction with a specific environment. After percieving the current state, the agent reasons about which action to select in order to obtain most rewards in the future. In this project, we use reinforcement learning (RL) to play the game of Bomberman, a strategic maze game where the player, through strategic placement of bombs, must kill other players and collect coins. We discuss how to give a reasonable approximation of the environment using **feature extraction**, and how to learn an optimal game strategy for the agent, the **control** of the agent.

1.1 Game Setup [Lily]

To quickly summarize the game setup as described by the project instructions: in the given Bomberman framework, our agent faces other agents and compete to find coins in crates and also defeat other agents using bomb, all while avoiding exploding bombs. This agent's tasks could be split into three levels:

1. Collect visible coins as quickly as possible without bumbing into walls on the side and in the middle of the game board.

2. Find coins hidden in crates and collect them by placing bomb without killing itself.

3. Defend itself against opposing agents and earn the highest score.

Possible actions for the agent are `UP`, `DOWN`, `LEFT`, `RIGHT`, `BOMB`, and `WAIT` and the environment (game board) is fully observable. The framework includes two agents to train against: `simple_agent`, an agent using hand-written policies, and `random_agent`, an agent which decides randomly on what action to take in the game.

1.2 Control[Ferdinand]

Recall that a Markov Decision Process (MDP) is a model for sequential decision making problems. The game of Bomberman is an MDP where the game state is fully observable. (The strategies of the opponents are however not.) We recall some basic definitions.

Let $\mathcal{S}$ be a finite set of states and $\mathcal{A}$ a finite set of actions. The reward function $r : \mathcal{S} \times \mathcal{A} \times \mathcal{S} \to \mathbb{R}$ denotes the expected reward when transitioning from the state $s \in S$ to a new state $s' \in S$ after performing the action $a \in \mathcal{A}$. For every state-action pair, a Q-value $Q(s, a)$ denotes the expected sum of rewards obtained after

performing action a in state s. The transition function $P(s, a, s')$ gives the probability of ending in state s' after selecting action a in state s.

For example, assume the Bomberman agent is in a given tile in the arena, in close vicinity to an opposing agent, and acts to move left towards a free tile. If the opposing agent acts to move towards to the same square, we may either be blocked in our movement, depending on which agent is allowed to act first. In this case, the probability $P(s, a, s')$ would less than 1, and the transition is not deterministic.

The **policy** $\pi : \mathcal{S} \to \mathcal{A}$ of an agent is a mapping between states and actions. Our goal is to learn a (near-)optimal policy for our agent. If the agent can predict state-transitions and rewards, the agent is **model-based**. If the agent only considers the present state, it is **model-free** and learns values and policies directly without explicitely constructing the transition function.

We distinguish off-policy and on-policy methods. **On-policy** methods attempt to *evaluate* and *improve* the policy used for making decisions. **Off-policy** methods seperates these two tasks. In an off-policy method, the policy used to generate behavior (the **behavior policy**) may be unrelated to the policy that is evaluated and improved (the **estimation policy**).

1.2.1 Q-Learning [Daniel]

For our agent, we have implemented the *Q-learning* method. This algorithm aims to approximate the optimal action-value function Q^* directly, and can be thought of as a sample-based, approximate version of value iteration that generate some sequence of action-value functions $(Q_k; k \geq 0)$.

A major attraction of Q-learning is its simplicity, and that it can be used off-policy. This allows us to use an arbitrary sampling strategy to generate the training data.

Definition. Q-learning is an off-policy method with updating rule:

$$Q(s_t, a_t) \longleftarrow Q(s_t, a_t) + \alpha[r_{t+1} + \gamma \max_a Q(s_{t+1}, a) - Q(s_t, a_t)]$$

Here, $\alpha \in [0, 1]$ is the **learning rate** parameter, which determines how fast the Q-value is pushed in a certain direction, and $\gamma \in [0, 1]$ the **discount factor**, which assigns a lower importance to future rewards. We call the value

$$\delta_t := r_t + \gamma \max_{a'} Q(s_t, a') - Q(s_t, a_t)$$

the **temporal-difference error** of Q-learning. When the MDP is finite, we typically choose $\gamma = 1$.

Q-Learning directly approximates Q_* independent of the policy being followed. Note that for the method to converge to the optimal Q_*, it is required that all pairs $Q(s, a)$ continue to be updated. [Sutton, 6.5 Q-learning] An optimal policy is then given by

$$\pi^*(s, a) = \arg\max_{a \in \mathcal{A}} Q^*(s, a)$$

1.2.2 Exploration methods [Ferdinand]

In Q-learning, the behavior policy still has an effect in that it determines which state-action pairs (s, a) are visited and updated. This gives rise to the *exploration-exploitation dilemma.* The agent can:

- **exploit** the knowledge that it has found for the current state s by doing one of the actions that maximizes $Q(s, a)$.
- **explore** in order to build a better estimate of the optimal Q-function. The

agent should sometimes select a different action from the one it currently thinks best.

There are a number of suggested ways to trade off exploitation and exploration. For more information how the choice affects the game of Bomberman, see [Kormelink, 4. Exploration Methods].

- **Random-Walk.** The agent executes a randomly chosen action at every time-step. This method produces completely random behavior, and is therefore suitable to compare against other methods.

- **Greedy method**. This method is the opposite of the Random-Walk exploration method. It assumes that the current Q-function is highly accurate, and therefore every action is based on exploitation, that is:

$$\pi(s) \hat{=} \arg\max_a Q(s, a)$$

  The agent always takes the action with the highest Q-value, because it assumes that this is the best action.

  Remark. This method attempts to improve on the Random-Walk in Bomberman. If the agent dies constantly in the early game, it will not get to explore the later part of the game. Because this method never selects exploration actions, it is however usually not suitable for learning the optimal policy.

- **ε-Greedy method**. Using a Greedy method, the agent may repetitively take sub-optimal actions if the Q-function is not a good approximation of the expected reward. ε-Greedy attemps to solve this problem by exploring the effects of different actions. It uses a parameter $0 \leq \varepsilon \leq 1$ to determine what percentage of actions is randomly selected by the agent: the action with the highest

Q-value is chosen with probability $1-\varepsilon$, and a random action otherwise. Thus $\varepsilon = 0$ translates to no exploration (Greedy), and $\varepsilon = 1$ to only exploration (Random-Walk).

- **Diminishing ε-Greedy**. If the agent improves its behavior over many games, the need for exploration lessens, and the exploration method may become more greedy. This can be reflected by gradually diminishing ε in the ε-Greedy method. In particular, set G_i as the amount of games (one game or *generation* equals 10 rounds) the agent has played so far, and G for the total amount of games. We define:

$$\varepsilon_{\text{Diminishing}} = \min\left(0.05, \varepsilon \cdot \left(1 - \frac{G_i}{G}\right)\right)$$

  Here, a minimum of 0.05 is set to make sure the agent keeps exploring in the long run.

1.2.3 SARSA [Ferdinand]

For comparison purposes, we briefly discuss the SARSA learning method. The name SARSA originates from an update rule which uses every element of the quintuple of events $(s_t, a_t, r_{t+1}, s_{t+1}, a_{t+1})$, which make up a transition from one state-action pair to the next.

Definition. SARSA is an *on-policy* method with updating rule:

$$Q(s_t, a_t) \longleftarrow Q(s_t, a_t) + \alpha[r_{t+1} + \gamma Q(s_{t+1}, a_{t+1}) - Q(s_t, a_t)]$$

Here, the action a_{t+1} is explicitely chosen by the behavior policy. If the state s_{t+1} is terminal, then $Q(s_{t+1}, a_{t+1})$ is defined as zero.

The convergence properties of SARSA depend on the nature of the policy's dependence on Q. For example, one could use a ε-greedy policy. SARSA converges almost surely to an optimal policy and action-value function, as long as all state-action pairs are visited an infinite number of times, and the policy converges in the limit to the greedy policy. (For example, by using a diminishing ε-greedy policy, or setting $\varepsilon = \frac{1}{t}$). For more information, see [Sutton, 6.4 Sarsa: On-policy TD Control].

1.3 Linear function approximation[Ferdinand, Lily and Daniel]

Even in the scenario where the Bomberman arena only contains (9 randomly distributed) coins and a single agent, the size of the state space $\mathcal{S}$ is significant:

$$\begin{aligned} |\mathcal{S}| &= 176 \cdot \underbrace{(176-1)}_{\text{coin 1 square}} \cdots \underbrace{(176-9)}_{\text{coin 9 square}} \\ &= \prod_{i=0}^{9} (176-i) \approx 2,20 \cdot 10^{22} \end{aligned}$$

Since every state uses its own Q-value for every action, this makes using lookup tables problematic: Q-learning needs to explore all actions in all states before being able to infer which action is best in a specific space. To address these issues, we use **linear function approximation**.

Other approximation methods include **non-linear function approximations** such as neural networks and decision trees. Due to their high demands on processing power and our lack of familiarity with the topic, we have not considered neural networks considered in our treatment.

Another possible method to avoid using look-up tables is to use regression algorithms, such as a **decision forest (ensemble methods)**. An ensemble can perform better than the individuals in the ensemble. As mentioned in the lectures, and as we

could also see in the assignments, an ensemble could get to 99% accuracy even if the accuracy of each individual in the ensemble is not better than 80%. This approach would also not require extensive feature engineering and could compete with neural networks.

However this method also tends to have a high demand of memory and processing power. We would need to make sure we have enough decision trees to cover all the possibilities in the game and make sure that all trees are independent of one another. One way of doing this is to use decision stumps to consider each free tile in the game, which means using more than 200 stumps. Therefore, we decided to not use decision trees in our project.

1.3.1 Feature extraction[Ferdinand & Daniel]

The downside of linear approximation is feature extraction. To make linear approximation work we need to use the right features. Finding good features requres extensive feature engineering. These features are a reasonably small amount of parameteres extracted from each given state. With these extracted features we then use a weight vector to approximate the Q-value:

$$Q(s,a) \approx w_0 + F_1(s,a)w_1 + F_2(s,a)w_2 + \cdots + F_n(s,a)w_n =: Q_w(s,a)$$

with $F : \mathcal{S} \times \mathcal{A} \to \mathbb{R}^n$ the **feature extraction** function.

By assumption, we have far more states than weights, thus making one state's estimate more accurate invariably means making others' less accurate. A good strategy in this case is to try to minimize error on the observed samples using gradient descent. Since we however do not know the true value of Q, but depend on the current value of the weight vector w_t during the time step t. For the approximation

Q_{w_t}, we thus have a bias and not a true gradient descent method. The update rule is given by:

$$\begin{aligned} w_{t+1} &= w_t + \alpha_t[r_{t+1} + \gamma \max_{a'} Q_{w_t}(s_{t+1}, a') - Q_{w_t}(s_t, a)]\nabla_w Q_{w_t}(s_t, a_t) \\ &= w_t + \alpha_t \delta_{t+1}(Q_{w_t}) \cdot Q_{w_t}(s_t, a_t) \end{aligned}$$

Q-learning and SARSA with linear function approximation and online updates are given in Algorithm 1 and 2. [Szepesvári]

Algorithm 1 Q-learning with linear function approximation

function QLearningLinFApp($X, A, R, Y, w, F, \alpha, \gamma$)
Input: X is the last state, Y is the next state, R the immediate reward associated with this transition, $w \in \mathbb{R}^d$ the parameter vector, $F(S, A)$ the feature extraction, α the learning parameter, and $\gamma \in [0, 1]$ the discount factor.

1. $\delta \leftarrow R + \gamma \cdot \max_{a' \in \mathcal{A}} w^T F(Y, a') - w^T F(X, A)$
2. $w \leftarrow w + \alpha \cdot \delta \cdot F(X, A)$
3. **return** w

Algorithm 2 SARSA with linear function approximation

function SARSALinFApp($X, A, R, Y, A', w, F, \alpha, \gamma$)
Input: X is the last state, A is the last action chosen, R is the immediate reward received when transitioning to Y, $w \in \mathbb{R}^d$ the parameter vector, $F(S, A)$ the feature extraction, α the learning parameter, and $\gamma \in [0, 1]$ the discount factor. A' is chosen from Y using the policy derived from Q.

1. $\delta \leftarrow R + \gamma \cdot w^T F(Y, A') - w^T F(X, A)$
2. $w \leftarrow w + \alpha \cdot \delta \cdot F(X, A)$
3. **return** w

Remark. arg max may return multiple actions that have the same Q value. In that case, the agent can randomly choose between the two actions to avoid bias. This

makes the agent strategy (that is, a probability distribution over the actions for this agent) a *stochastic strategy*. See [Poole, 11.4 Reasoning with Imperfect Information] for more information.

1.3.2 Feature selection[Daniel]

We construct the features incrementally, trying to solve the tasks mentioned in Section 1.1 one after another.

Overall we represent the states and actions using 15 features, described as follow:

1. Reward the action that move the agent towards the nearest coin.

   If there is no such reachable coin (which means that there is neither a visible coin for the agent, nor that it can be reached by a movement), then the feature for all actions is set to zero, that is, $F_1(s, a) = 0$ for all $a \in \mathcal{A}$.

   If there is a reachable coin for the agent, then $F_1(s, a) = 1$ is set for the best action a leading to this coin and $F_1(s, a') = 0$ for $a' \in \mathcal{A} \setminus \{a\}$.

   To find the closest reachable coin to the agent, a breadth-first search of the free tiles is performed.

2. Penalize the action that most likely leads to the agents death.

   In the settings of our game, once a bomb is dropped, it will detonate after a *bomb_timer* number of steps and create an explosion that extends *bomb_power* tiles up, down, left and right. The explosion destroys crates and agents, but will stop at stone walls and does not reach around corners. The duration of the explotion is given by an *explosion_timer*.

   Based on this setting, $F_2(s, a) = 1$ is set for every action a leading the agent to cross the path of an exploding bomb. If the action a does not lead to the

death of the agent, then $F_2(s, a) = 0$.

3. Penalize the action that leads the agent to going towards, or remain inside an area threatened by a bomb.

   For this feature, any bomb present in the arena is considered from the moment it was placed by an agent, until it explodes. The blast range of all the bombs are computed, as if the bombs were about to explode and the actions leading the agent to one of the bombs' range is penalized. This means: $F_3(s, a) = 1$ if the action a leads the agent into blast range, and $F_3(s, a) = 0$ otherwise.

4. Reward the best movement that leads the agent outside of any blast range.

   This feature is only computed when the agent could be threatened by an explosion of any bomb in the arena. If this is not the case, then $F_4(s, a) = 0$ is set for all $a \in \mathcal{A}$.

   If the agent is inside of the blast range of any bomb, then a breadth-first search of the free tiles is performed to find the closest reachable free space outside the blast range. If the action a leads the agent towards this free space, then $F_4(s, a) = 1$, and $F_4(s, a) = 0$ otherwise.

5. Penalize invalid actions.

   Invalid movements occur when the agent tries to move into a crate, bomb, wall or any other agent in the arena. Another invalid action is to place a bomb when the agent has no more remaining bombs. This is because in the settings of our game, the agent can only drop a new bomb after its previous one has exploded.

   $F_5(s, a) = 1$ is set for any invalid action a and $F_5(s, a) = 0$ otherwise.

6. Reward the movement for collecting a coin.

   The difference between this and F_1, is that the movement is only rewarded if the agent would collect a coin in the next step. This means that this feature only applies, if the agent is located next to a coin.

   $F_6(s, a) = 1$ is set if the action a leads to collect a coin and $F(s, a) = 0$ otherwise.

7. Reward the agent for placing a bomb next to a crate, if the agent can possibly escape from the blast radius.

   This feature is intended only for placing bombs, for any other action $F_7(s, a) = 0$.

8. Reward the best movement that leads to the next crate.

   This feature uses a very similar approach as Feature 1, but looks for the next reachable crate instead of coin. $F_9(s, a) = 1$ is set for the action a leading as fast as possible to the next crate and $F_9(s, a') = 0$ for all $a' \in \mathcal{A} \setminus \{a\}$. If the agent is next to crate or all the crates in the arena are alredy distroyed, then $F(s, a)$ is set to zero for all $a \in \mathcal{A}$.

9. Penalize the movement of going into a dead_ends if the agent can get trapped by his own bomb.

   This feature fixes one of the issues generated by Feature 7 or 8, which is, letting the agent get trapped in a dead_ends by his own bomb. To adress this issue, the agent checks if his last action was a bomb and penalizes the movement of going into a dead_end. $F_{10}(s, a) = 1$ is set for the action that leads to the agent getting trapped in this way. In case that the agent is not located next

to an dead_ends or didn't place a bomb in his last action F_{10} is set to zero for all $a \in \mathcal{A}$.

10. Reward placing a bomb next to an opponent if the agent can escape from it. This feature only take place, if there are less than 4 crates in the arena and no visible coins. In another case $F_{11}(s, a) = 0$ for all $a \in \mathcal{A}$.

    The agent is rewarded for placing a bomb, only if he is next to another agent and if he can scape from its blast range. The approach used for this, is the same as for Feature 7. In this cases $F_{11}(s, a) = 1$ and $F_{11}(s, a') = 0$ for all $a' \in \mathcal{A} \setminus \{a\}$ for

11. Reward the best movement that leads to an opponent.

    As Feature 11, this feature only applies if there are less than 4 crates in the arena and not visible coins anymore. Similar as the other features, a breadth-search is performed in order to find the nearest reachable apponent.

    $F_{12}(s, a) = 1$ is set for the action a leading as fast as possible to another agent and $F_{12}(s, a) = 0$ otherwise. If we are already next to an agent, then $F_{12}(s, a)$ is set to zero for all $a \in \mathcal{A}$

12. Reward placing a bomb that could trap an opponent in a dead_ends.

    This feature is used to trap opponents that are in a dead_ends with a bomb. As this feature is intended only to reward placing a bomb, $F_{13}(s, a) = 0$ is set for any other action a.

    If the agent is next to an opponent which is in a dead_ends, then $F_{13}(s, a) = 1$ for placing a bomb in any other case $F_{13}(s, a) = 0$.

13. Reward placing a bomb that could kill an opponent.

Similar as feature 13, this feature only consider the action of placing a bomb, for any other action $F_{14}(s, a)$ is set to zero.

The blast range of the bomb that might be placed is computed and then it is checked if this could reach another opponent. If the agent might sucess on that and might scape from his own explotion, then $F_{15}(s, a)$ is set to 1, otherwise to zero.

14. Reward the best movement that leads to a nearby opponent.

    This feature does the same as feature 12 but independently of the number of crates and coins in the arena and just taking consideration agents that are reachable at most by three steps.

We considered the following feature but did not use during our training process:

- Reward the best movement that leads the agent towards a dead end.

  By a dead ends, we mean tiles in the arena where only one movement action is valid. This means, tiles surrounded by walls and crates with only one "opening".

  Placing a bomb in such a tile has the advantage that it is not possible to get trapped by an own bomb. Similar to features mentioned before, a breadth-first search is performed to find the movement that leads as fast as possible to a dead_end, $F_8(s, a) = 1$ is set in this case and $F_8(s, a) = 0$ otherwise. In case that the agent is already next to a dead_ends, then $F_8(s, a)$ is set to zero for all $a \in \mathcal{A}$.

1.4 Implementation [Daniel]

In the duration of the project, we developed a workflow to easily and quickly develop new features that can be calculated quickly for all possible actions during runtime.

$$\begin{pmatrix} F_1(S,A_1) & F_2(S,A_1) & \cdots & F_{15}(S,A_1) \\ F_1(S,A_2) & F_2(S,A_2) & \cdots & F_{15}(S,A_2) \\ \vdots & \vdots & \ddots & \vdots \\ F_1(S,A_6) & F_2(S,A_6) & \cdots & F_{15}(S,A_6) \end{pmatrix}$$

Figure 1: Feature Matrix

Our feature functions F_1 - F_{15} return a vector $V \in \{0,1\}^6$, each value in the vector representing the value of the feature function for an action. We then stack these vectors together vertically to form a matrix of size $N \times 6$, whereas N is the number of features. A row in this matrix would then be the feature vector of an action. Using this implementation the Q-value could be calculated with a simple dot product between the matrix and the weight vector, which NumPy does very efficiently.

Many of our features uses similar basic calculations, i.e. distance calculation. It would further improve our implementation's efficiency if we run calculations that will be used for multiple features only once. This is the motivation behind the Python class `RLFeatureExtraction` in `feature_extraction.py`. Using this class, we could easily add and remove different features for testing.

1.5 Convergence[Ferdinand & Lily]

While lookup tables ensure that the above gradient descent method converges (Figure 2), linear function approximators may cause all weights and values to diverge. We have confirmed this for our chosen features in the game of Bomberman, see 3. Experimental Results. There are several approaches at improving convergence, in particular **experience replay**. [DeepMind]

		Not Residual			Residual		
		Fixed distribution (on-policy)	Fixed distribution	Usually-greedy distribution	Fixed distribution (on-policy)	Fixed distribution	Usually-greedy distribution
Markov Chain	Lookup table	Y	Y		Y	Y	
	Averager	Y	Y		Y	Y	
	Linear	Y	N		Y	Y	
	Nonlinear	N	N		Y	Y	
MDP	Lookup table	Y	Y	Y	Y	Y	Y
	Averager	Y	Y	N	Y	Y	Y
	Linear	N	N	N	Y	Y	Y
	Nonlinear	N	N	N	Y	Y	Y
POMDP	Lookup table	N	N	N	Y	Y	Y
	Averager	N	N	N	Y	Y	Y
	Linear	N	N	N	Y	Y	Y
	Nonlinear	N	N	N	Y	Y	Y

Figure 2: Convergence of Reinforcement Learning, courtesy of `http://www.cs.cmu.edu/afs/cs.cmu.edu/project/learn-43/lib/photoz/.g/web/glossary/converge.html`

1.5.1 Experience replay [Ferdinand]

Instead of updating the weights w after each time step, experience replay adds experiences (s, a, r, s') to a buffer in memory (the training set), and samples a mini-batch from this buffer to update the current weights. The goal in this sampling is to avoid corellation between experiences in the same episode, which may lead to divergence during gradient descent. Sampling also avoid the rapid forgetting of possibly rare experiences that may be useful later on.

Note that when learning by experience replay, it is necessary to learn off-policy: by construction, the parameters w used to generate the samples (s, a, r, s') are different from our current parameters w'. This motivates the choice of Q-learning over on-policy algorithms like SARSA.

The weight updates can be done by **incremental gradient descent**, where the weights change while the algorithm iterates over the samples, or **batched gradient**

descent, where the algorithm computes the changes to the weights after every sample, but only applies the changes after the batch. It is typical to start with small batches to learn quickly and then increase the batch size so that it converges. [Poole, 7.3.2 Linear Regression and Classification] A possible experience replay algorithm for experience replay is given by Algorithm 3.

Algorithm 3 Experience replay with batched gradient descent

---

function ExperienceReplayLinFApp($B, m, w, F, \alpha, \gamma$)

Input: B is the replay buffer of size N, containing experiences (s, a, r, s'), m is the size of the mini-batch to be sampled from the replay buffer, $w \in \mathbb{R}^d$ is the parameter vector, $F(S, A)$ is the feature extraction, α is the learning parameter, and $\gamma \in [0, 1]$ is the discount value.

1. $b \leftarrow$ RandomSample(B, m)
2. $u \leftarrow 0$
3. for X, A, R, Y in b do:
   $\delta \leftarrow R + \gamma \cdot \max_{a' \in \mathcal{A}} w^T F(Y, a') - w^T F(X, A)$ if Y is not terminal;
   $\delta \leftarrow R$ if Y is terminal.
   $u \leftarrow u + \frac{\alpha}{m} \cdot \delta \cdot F(X, A)$
4. $w \leftarrow w + u$

A variation on this algorithm is **prioritized experience replay**, where we sample from the replay buffer with a probability depending on highest reward, temporal-difference error, or other factors such as the time step for the experience. We discuss these methods in more detail in Section 2.

1.5.2 Residual gradient descent [Lily]

Another approach to improve convergence is using **residual algorithms**, which minimizes the mean squared Bellman residual and in effect "take the effects of generalization, but attempt to make each state match both its successors and its pre-

decessors." [Baird] This combined with incremental gradient descent helps stabilize the weights, as residual gradient is guaranteed to converge. The update rule using residual gradient descent is changed as follows:

$$\Delta W_{rg} = -\alpha \sum_{x} [R + \gamma V(x') - V(x)][\nabla_w \gamma V(x') - \nabla_w V(x)]$$

Algorithm 4 Q-learning with linear approximation residual gradient

function QLearningLinFAppResidual(H, w, F, α, γ)

Input: H is a list of tuples containing X the last state, A the last action, Y the next state, A' the next action, R the immediate reward associated with this transition, $w \in \mathbb{R}^d$ the parameter vector, $F(S, A)$ the feature extraction, α the learning parameter, and $\gamma \in [0, 1]$ the discount factor.

In the end of each episode:

1. TD = 0
2. for all states X, A, R, Y, A' in H:
   (a) $\delta \leftarrow R + \gamma \cdot \max_{a' \in \mathcal{A}} w^T F(Y, a') - w^T F(X, A)$
   (b) $TD \leftarrow TD + \delta \cdot (\gamma \cdot w^t F(Y, a') - w^t F(X, a))$
3. $w \leftarrow w + \frac{\alpha}{size(H)} \cdot TD$
4. **return** w

2 Agent training

2.1 Training Setup [Lily]

Our general training setup is to run the game with the graphical interface turned off and the option to end the game once the training agent dies on. We used the given configuration of 4 players, 400 steps per round and 10 rounds per game to simulate an actual game during the tournaments. However, as it would be much too slow

and too much of a hassle to manually restart the game after just 10 rounds; letting he game run for 1000 would be a more efficient way to train. To still simulate the lenght of a real game, we adjusting our training system to reset necessary training parameters, such as total rewards in a game, after each 10 round.

We trained by playing against 3 simple agents with $\alpha = 0.01, \gamma = 0.95$ and $\epsilon = 0.15$ when training using the ϵ -greedy policy

2.2 Evaluating Training Results [Lily]

For each training session, we saved the trained weights, how the weights changes over time and also how the agent accumulate rewards over time. These informations are saved as an array in a `.npy` file which can be loaded an a later time without running the entire training again for further analysis of training performance.

We have mostly been evaluation our training performance based on the accumulated rewards over a game (10 rounds) to see how much better the trained agend perform after training.

2.3 Initial Weights to Speed Training [Daniel]

To speed up our training, we come up with an initial weight values so that our training does not have to start from a random value, which would lead to very low rewards in the game for a very long time. With our feature matrix, an agent with random weights would almost do nothing and perform much worse than even the given random agent.

Since we crafted the features and rewards ourselves, we could also crafted an initial guess for the weight values, giving high values for features that we think are more important than others. This is one of the advantages of using an approach that

relies heavily on feature engineering.

Using feature F_1 to F_{15}, our initial guess for the weight values is:

$$[1, 1.5, -7, -1, 4, -0.5, 1.5, 1, 0.5, 0.8, 0.5, 1, -1, 0.6, 0.6]$$

2.4 Choice of rewards [Ferdinand, Daniel]

We transform action consequences into something that Q-learning can use to learn the Q-function by giving in-game events a numerical reward. For learning the optimal behavior, the rewards of different objectives should be set carefully so that maximizing the obtained rewards results in the desired behavior.

Intermediary rewards (rewards given before the game end) are used because rewards in Bomberman are handed out sparsely – for example, 1 point when a coin is collected, and 5 points when an opponent is destroyed. Note that a-priori, the "squares" where the agent is positioned are equivalent, as are the movements from one square to the next.

We have chosen the following rewards to clearly distinct between good and bad actions. Dying is represented by a very negative reward. The reward of killing a player is attributed to the player that actually placed the involved bomb. The rest of the rewards promote active behavior, and no reward is given to finally winning the game (when all other players died).

- While the amount of time steps is limited, we have not given a specific reward ($R = 0$) when taking any action. Rewards for specific events are always accumulated to R. Similarly, we give no specific reward when the agent waits (`e.WAITED`).

- We give a slight reward of 1 to the action of dropping a bomb (`e.BOMB_DROPPED`),

to leave flexibility in the chosen strategies by the agent.

- Collecting a coin (`e.COIN_COLLECTED`) is the secondary goal of the game, and gives a reward of 200. Coins are intially hidden inside crates which must be destroyed by the agent. Destroying a single crate (`e.CRATE_DESTROYED`) gives a reward of 10, since multiple crates may be destroyed at once by placing a bomb. Finding a coin inside a crate (`e.COIN_FOUND`) gives a reward of 100.

- We wish to avoid killing ourselves through walking into an explosion of a bomb we placed (`e.KILLED_SELF`), so we give this action a strong negative reward of -1000.

- Killing an opposing agent (`e.KILLED_OPPONENT`) is the primary goal of the game, and gives a reward of 700. Similarly, dying at the hands of an opponent gives a penalty of -500. This penalty is lower than the penalty for `e.KILLED_SELF`, since the strategies of opposing agents are unknown.

- An invalid action, such as bumping into a wall or crate or attempting to place a bomb when the agent has none left (`e.INVALID_ACTION`) is equivalent to performing no action at all. We give a small negative reward of -2 when performing an invalid action.

- If the agent survives the current round (`e.SURVIVED_ROUND`), we give a positive reward of 200. This reward is not as high as some other rewards, because the objective of the game is killing opposing agents and collecting coins, not necessarily for the agent to remain alive for the longest period.

3 Experimental results

3.1 Initial training [Daniel, Lily]

Our initial training idea was to use a direct gradient descent to train our weight vector. With this approach, we perform gradient descent and update the weights after each time step in the game. However, during training we noticed that the weights very quickly explode into large values and never converge. Moreover, the cummulative rewards over different episodes was decreasing into very high negative values, meaning that our trained agent was not getting better and was in fact dying very quickly in the game. This is probably due to the quickly growing weight vector. This divergence in the weight vector happens regardless what exploration method we used (greedy, ϵ-greedy, or diminishing ϵ-greedy).

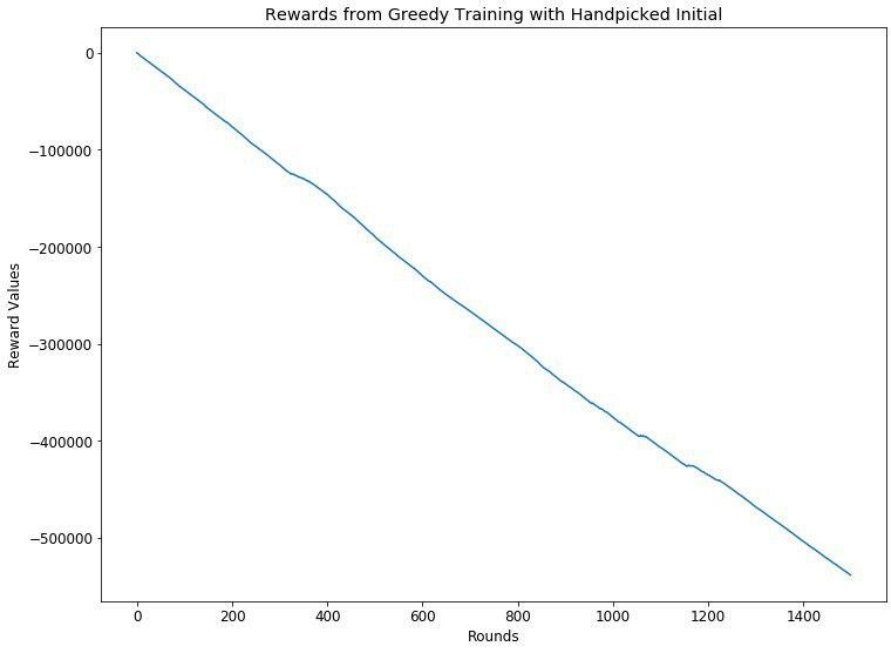

Figure 3: Accumulated rewards over 1500 rounds with 1-step, greedy learning.

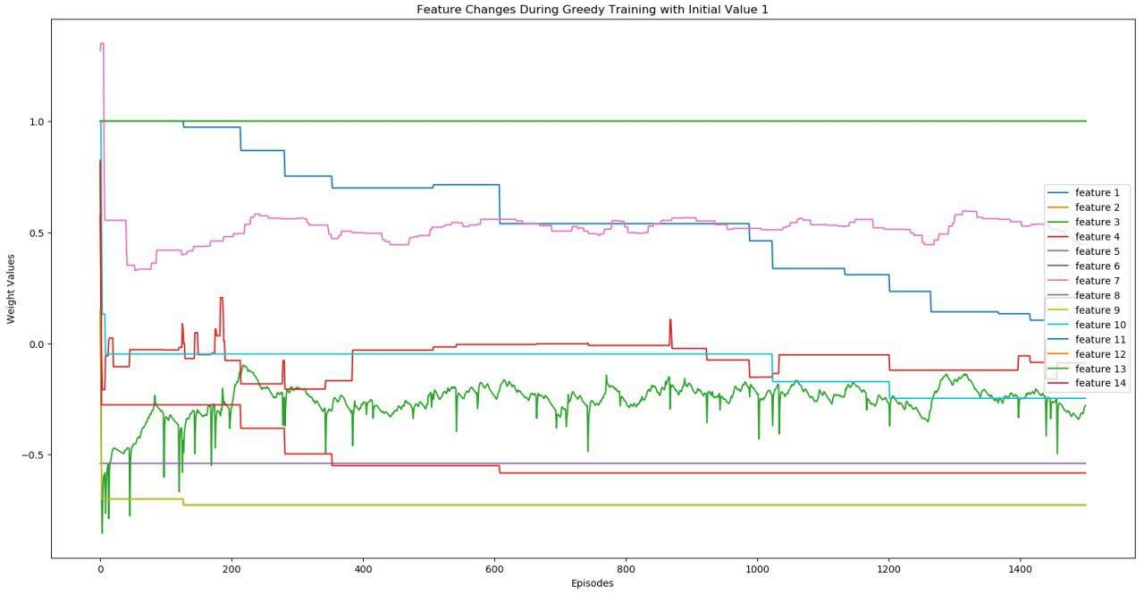

Figure 4: The values of the weight vector over 1500 rounds with 1-step, greedy learning.

3.2 Improving Training Results [Ferdinand, Lily]

3.2.1 Residual Gradient Descent [Lily]

What we've found to improve our training results significantly is using incremental gradient descent, that is calculating weight changes for each time step, and updating the weights using the average of all weight changes. We combined this with the idea of residual gradient descent, which seems to possibly improve the performance of our agents. We also thought to normalize the weight vector after each update to keep the weight values from blowing up. The idea for this is However, the improvement is only very slightly, and the performance of our agent fluctuates greatly between games.

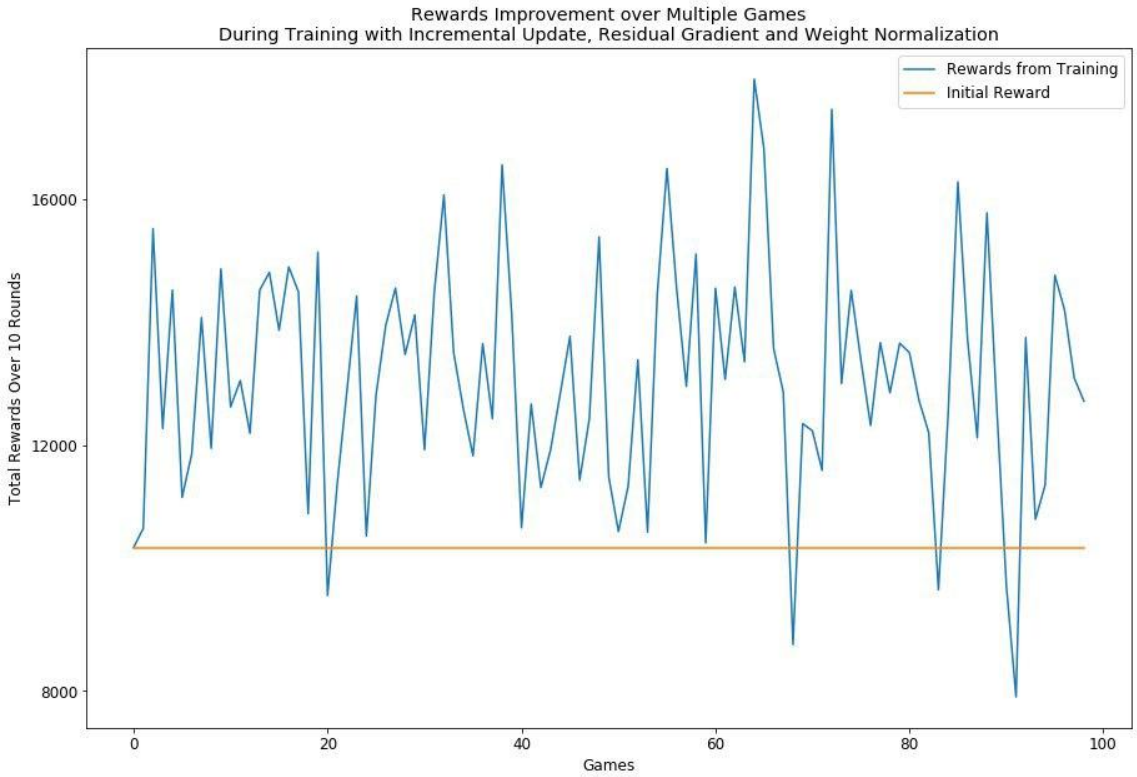

Figure 5: Total rewards over 10 rounds, using residual gradient descent with incremental updates.

3.3 Prioritization of experience replay [Ferdinand]

As described in 1.5, we can use **prioritized experience replay** to sample experiences from the replay buffer. Prioritizing which transitions are replayed can make experience replay more efficient. The key idea is that an agent can learn more efficiently from some transitions than from others. Transitions may be more or less surprising, redundant, or task-relevant. Some transitions may not be immediately useful to the agent, but might become so when the agent competence increases. In other words, experience replay liberates online learning agents from processing transitions in the exact order that they are experienced, and prioritized replay further liberates agents from considering transitions with the same frequency they are

experienced. [DeepMind]

For example, one could use the temporal-difference (TD) error in every experience, or the associated reward R, to set the probabilities for which we take samples from a (mini-)batch rather than sample uniformly. We have adopted a more naive approach after observing that most important events happen in the first 100 to 200 time steps in a round of Bomberman.

We have defined the following hyperparameters B_i:

- `self.replay_buffer_max_steps`. Defaults to $B_1 = 200$. Only add those experiences (s_t, a_t, r_t, s_{t+1}) to the replay buffer where $t \leq B_1$.
- `self.replay_buffer_update_after_nrounds`. Defaults to $B_2 = 10$. The amount of rounds that are played before the replay buffer is sampled, and the weights are updated through (batch) gradient descent.
- `self.replay_buffer_sample_size`. Defaults to $B_3 = 50$. This is the starting size of the mini-batch sampled uniformly from the replay buffer. After every completed generation, this value is increased by 20 to improve convergence.
- `self.replay_buffer_every_ngenerations`. Defaults to $B_4 = 1$. The amount of generations played before the replay buffer is emptied.

In this approach, we can use either incremental or batch gradient descent, though interestingly batch gradient descent works significantly better.

In the end the weights that we use for the competition have been those generated by Figure 7:

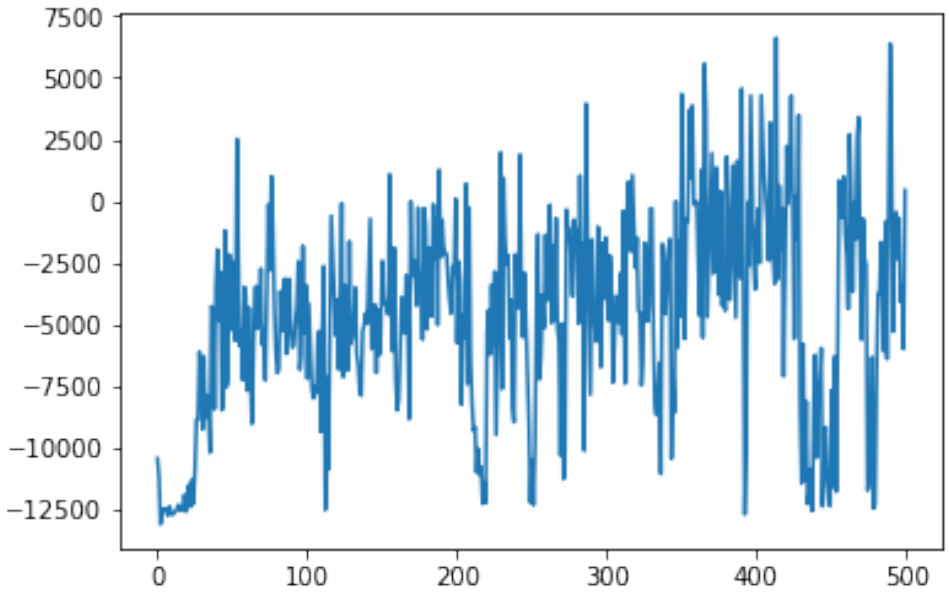

Figure 6: Experience replay using incremental gradient descent, and starting from w intialized with random values. The rewards are taken over 10 rounds each, over a total of 5000 rounds.

$$[1, 2.36889437, -7.40026249, -1.34216494,$$
$$3.69025486, -0.64679605, 2.34142438,$$
$$1.04329996, 0.50236224, 1.25573668,$$
$$0.47097264, 1.00000857, -1, 0.51097347, 0.22704274]$$

These weights resulted in considerable better performance of our agent over our intial guess.

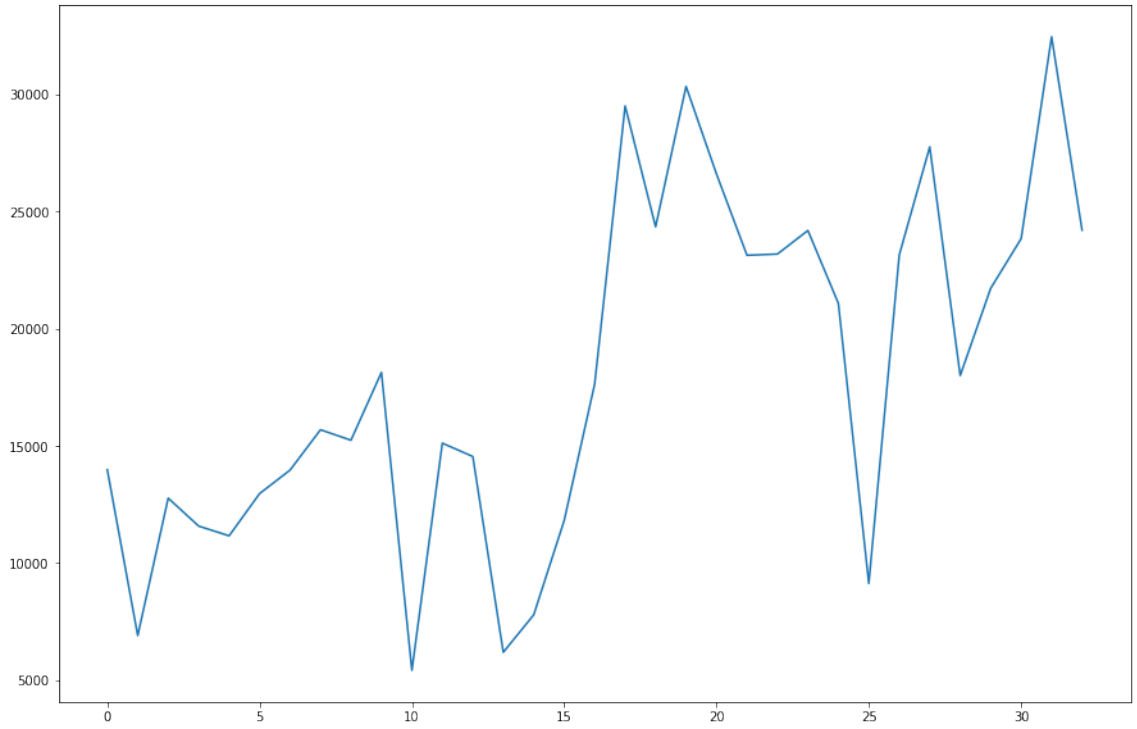

Figure 7: Experience replay using batch gradient descent, and starting from an initially guessed w. The rewards are taken over 10 rounds each, over a total of 500 rounds.

4 Outlook [Daniel]

If we had more time for the project, we would have further improved our features, as well as continue to explore various methods in experience replay, and residual gradient descent. During the course of the project, we have studied the "Double Q-Learning" algorithm, but we had no time to implement and test it in our code. It is described in Section 4.1 below.

For the game setup in next year's lecture, we would propose a more elaborate documentation of the basic Bomberman framework. We spent considerable time to study how the given framework is implemented, and how it interacts with our agent code.

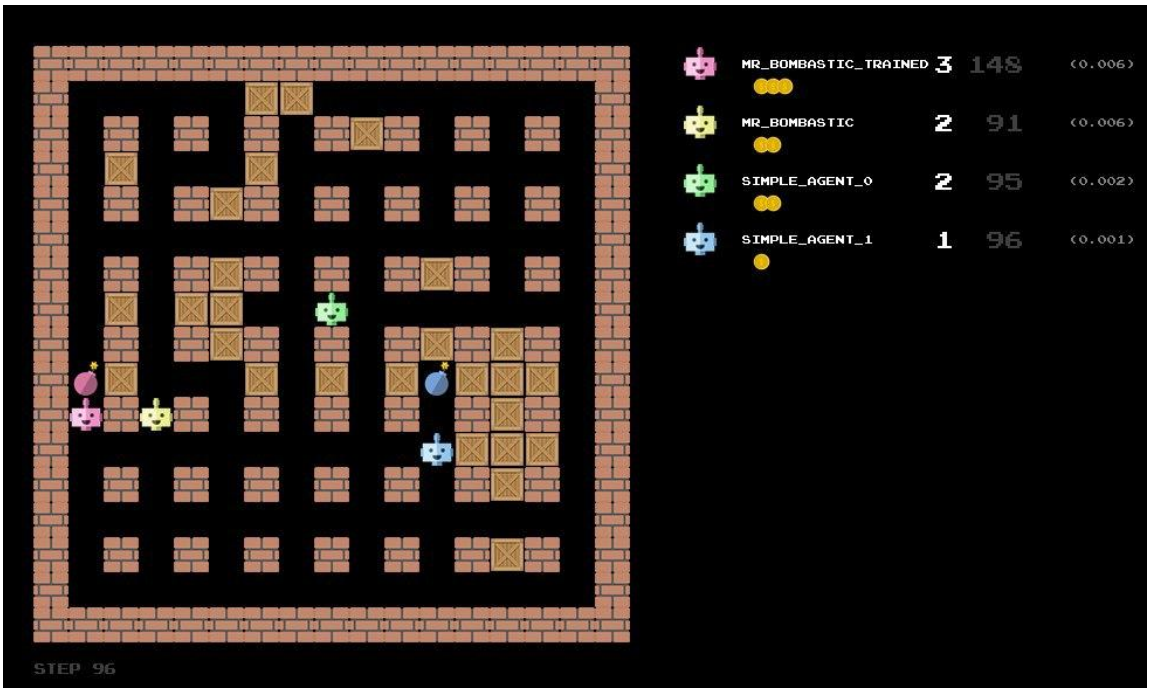

Figure 8: The trained agent reaches significantly better results than the agent with hand-picked parameters.

4.1 Double Q-learning[Ferdinand, Daniel and Lily]

The control algorithms we have discussed in Section 1.2 involve maximization in the construction of their target policies: in Q-learning, the target policy is the the greedy policy $\pi^*(s, a)$, which is defined with a maximum, and in SARSA, the policy may be ε-greedy, which also involves a maximization operation. In these algorithms, a maximum over estimated values is used implicitely as an estimate of the maximum value, which can lead to a positive bias.

For example, consider a single state s where there are many actions a whose true Q values are all zero, but whose estimated values $\hat{Q}(s, a)$ are uncertain and thus distributed some above and some below zero. The maximum of the true values is zero, but the maximum of the estimates is positive. We call this **maximization bias**. [Sutton, 6.7 Maximization Bias]

The idea of double learning extends to Q-learning, called **Double Q-learning**. Here we consider two values Q_1 and Q_2, with one of both chosen randomly at each time step. The update rule is then either given by:

$$Q_1(S,A) \leftarrow Q_1(S,A) + \alpha\left(R + \gamma Q_2\left(S', \arg\max_{a\in\mathcal{A}} Q_1(S',a)\right) - Q_1(S,A)\right)$$

or:

$$Q_2(S,A) \leftarrow Q_2(S,A) + \alpha\left(R + \gamma Q_1\left(S', \arg\max_{a} Q_2(S',a)\right) - Q_2(S,A)\right)$$

An ε-greedy policy for Double Q-learning could be based on the average (or sum) of the two-action value estimates.

Remark. Although we learn two estimates, only one estimate is updated on each play. Therefore, double learning doubles the memory requirements, but does not increase the amount of computations per step.

Algorithm 5 Double Q-learning with linear function approximation

---

function DoubleQLearningLinFApp($X, A, R, Y, w_1, w_2, F, \alpha, \gamma$)
Input: X is the last state, A is the last action chosen, R is the immediate reward recieved when transitioning to Y, $w_1, w_2 \in \mathbb{R}^d$ are the parameter vectors, $F(S,A)$ the feature extraction, α the learning parameter, and $\gamma \in [0,1]$ the discount factor.

1. With 0.5 probability:
   (a) $\delta \leftarrow R + \gamma \cdot w_2^T F(Y, \arg\max_{a'\in\mathcal{A}} w_1^T F(Y,a')) - w_1^T F(X,A)$
   (b) $w_1 \leftarrow w_1 + \alpha \cdot \delta \cdot F(X,A)$

   else:
   (a) $\delta \leftarrow R + \gamma \cdot w_1^T F(Y, \arg\max_{a'\in\mathcal{A}} w_2^T F(Y,a')) - w_2^T F(X,A)$
   (b) $w_2 \leftarrow w_2 + \alpha \cdot \delta \cdot F(X,A)$
2. **return** w_1, w_2

References

[Szepesvári] C. Szepesvári. *Algorithms for Reinforcement Learning*, 2009.

[Kormelink] J.G. Kormelink, M. Drugan, M. Wiering. *Exploration methods for Connectionist Q-Learning in Bomberman*, 2018.

[Poole] D. Poole, A. Mackworth. *Artificial Intelligence: Foundations of Computational Agents*, 2017.

[Lopes] M. Lopes. *Bomberman as an Artificial Intelligence Platform*, 2016.

[Sutton] R. Sutton, A. Barto. *Reinforcement Learning: An Introduction*, 2018.

[Baird] L. Baird. *Residual algorithms: Reinforcement Learning with Function Approximation*, 1995.

[DeepMind] DeepMind Technologies. *Playing Atari with Deep Reinforcement Learning*, 2013.

[DeepMind] Google DeepMind. *Prioritized Experience Replay*, 2016.