Machine Learning: Final project
bomberchamp

Johannes Vogt Anna Sterzik

25.03.2019

1 Introduction

Anna Sterzik, Johannes Vogt

This semesters project for the lecture "Fundamentals of Machine Learning" consists of implementing a reinforcement learning agent which is able to play the game Bomberman in multiplayer mode. Hereby the agent needs to kill opponents and collect coins. To make this possible it has to place bombs strategically in order to destroy crates and opponents. Some of the crates contain coins which the agent can collect as soon as the crates are destroyed. The agent is placed in one of the four corners of a maze-like environment with opponents in the other corners. During the following 400 time steps it has to gain more points than the other agents through fighting and collecting points. For this project we implemented the Rainbow DQN[3], a Deep Q Network that includes a variety of extensions to improve its performance, with the exception of the distributional extension. After solving the singleplayer tasks, we employ a self-play strategy inspired by OpenAI Five[5].

The implementation can be found at

`https://github.com/bomberchamp/bomberchamp`.

1.1 Reinforcement Learning

Anna Sterzik

In reinforcement learning, sequential decision making problems need to be solved by an agent. The agent is provided with some input state and then

needs to choose an action. Input state and chosen action are used to create a subsequent state and a reward signal for the agent. By maximizing the reward signals the agent develops a policy for acting in the environment and thus gets better in solving the given problem. Finding the optimal policy is the goal of reinforcement learning. Hereby the environment is fully characterized by the state.
Time is discrete in reinforcement learning. Therefore the consecutive states form a chain. Nevertheless transitions from one state to its subsequent state are not dependent on history, they only depend on the current state. This is called the Markov assumption.
For finding the optimal policy the state value function $V_\pi(s)$ can be used. It is defined as the expected reward of a policy π that can be reached by starting at the state s

$$V_\pi(s) = \mathbb{E}\left[R\right]|_{\pi,s}. \tag{1}$$

The expected reward R for a given state s_0 is defined as the sum of the current reward and the discounted future rewards

$$R = \sum_{t=0}^{\infty} \gamma^t \, R_t. \tag{2}$$

$\gamma \, \epsilon \, [0, 1]$ is the discount factor.
The optimal policy π^* maximizes the state-value function.

1.2 Neural Networks

Johannes Vogt

As regression model we use a neural network, consisting of multiple fully connected and convolutional layers. A fully connected layer can be represented as

$$y = wx + b, \tag{3}$$

where w and b are learned through backpropagation. By stacking multiple of these layers and separating them with an activation function, e.g. the rectified linear unit

$$\phi_{relu}(y) = \begin{cases} y & y > 0 \\ 0 & otherwise, \end{cases} \tag{4}$$

more complex functions can be approximated.

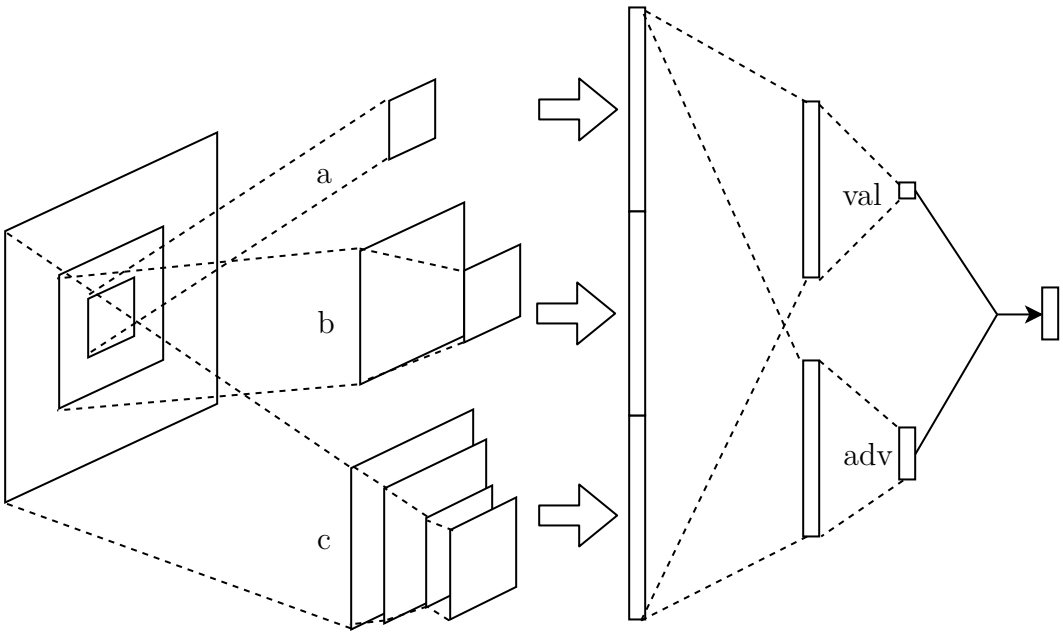


Figure 1: Convolutional network focused on the central region.

Instead of fully connected layers, convolutional layers can be used to process images or other spatially connected inputs. Each convolutional layer takes an input of size $w \times h \times c$ and generates an output of size

$$\frac{w - f + 2p}{s} + 1 \times \frac{h - f + 2p}{s} + 1 \times n_c$$

by passing n_c filters over regions of $f \times f \times c$ neighboring values in the input. f is the filter size and n_c is the number of filters. The stride s defines the distance that the region is shifted for every step horizontally and vertically. The input is often padded with p rows / columns of zeros on either side to avoid border artifacts. For this project we use *same* padding, so that the output is always of size

$$\frac{w}{s} \times \frac{h}{s} \times n_c.$$

We use a convolutional neural net that is focused on the central region of the input as shown in figure 1. The input layer gets separated into streams a, b and c, each taking a centred cropping of the input. Each stream then processes the croppings through a number of convolutional layers depending

on the size of the cropping. The results get flattened and concatenated after which they are passed on to the value and advantage streams further described in 3.3.

2 DQN

2.1 Q-learning

Anna Sterzik

In Q-learning [2] instead of the state-values, Q-values are being used. They are defined as the current reward depending on the chosen action and the discounted future rewards under the premise of always using the policy π for further decision making after the initial action choice:

$$Q_\pi(s,a) = \mathbb{E}\left[R\right]|_{a,s,\pi}. \tag{5}$$

The best policy is then determined by maximizing Q_π over the actions in every step. Usually it is not possible to learn every action for every state explicitly because the problems are often to extensive. Therefore the Q-values get approximated by a parametrized function $Q(s,a;\theta_t)$ with θ_t being the parameters. The Q-function is then updated towards the target value

$$Y_t^{\mathrm{Q}} \equiv R_{t+1} + \gamma \max_a Q\left(s_{t+1}, a; \theta_t\right) \tag{6}$$

after each action via:

$$\theta_{t+1} = \theta_t + \alpha\left(Y_t^{\mathrm{Q}} - Q\left(s_t, a; \theta_t\right)\right)\nabla_{\theta_t} Q\left(s_t, a; \theta_t\right). \tag{7}$$

This makes it an off-policy algorithm, as the optimal Q-value: Q^* is approximated by Q directly (regardless of the followed policy) in contrast to for example SARSA. The policy is still important as it determines, which state-action pairs are used to update the model [9]. Q-learning is also a model free approach, as it doesn't make a model of the environment but instead directly estimates Q^*.

2.2 Q-Networks

Anna Sterzik

For some reinforcement learning problems, the simplest implementation of the Q-learning algorithm, a table is perfectly sufficient to find a good policy.

If we consider more complicated problems with bigger inputs other methods are needed. For state of the art reinforcement learning, usually neural networks are being used. They are especially convenient as they are trained from raw inputs, which makes handcrafted features redundant [4]. Neural Networks which are implemented to learn with the Q-algorithm are called Q-networks. They are non linear function approximations for $Q(s, a; \theta_t)$. To update the Q-network the loss function

$$L_t(\theta_t) = \mathbb{E}_{s,a\sim\rho(s,a)}\left[\left(Y_t^{\mathrm{Q}} - Q(s, a; \theta_t)\right)^2\right] \tag{8}$$

is used, where $\rho(s, a)$ is a probability distribution over states and actions. To make this compatible with the Q-learning algorithm, the weights need to be updated at every step and the expectations exchanged with samples from the probability distribution $\rho(s, a)$.

2.3 DQN

Anna Sterzik

Deep Q-networks [4] are multilayered neural networks which make use of the Q-learning algorithm. The two mayor improvements of the model are the use of two separate networks and an experience buffer. The two networks are called online network and target network. The target network is used to calculate the targets Y_t^{DQN}. It has the same structure as the online network but to make learning more stable, the weights of the target network θ_t^- stay constant for a longer time. They are being copied from the online network every τ steps. The target is then calculated by:

$$Y_t^{\mathrm{DQN}} \equiv R_{t+1} + \gamma \max_a Q\left(s_{t+1}, a; \theta_t^-\right). \tag{9}$$

If only one network was used, an update of $Q(s, a)$ would often not only lead to a higher value of $Q(s_t, a_t)$ but also to higher expected Q-values $Q(s_{t+1}, a)$ for all actions. If the target was also calculated by this network this could then lead to oscillations or divergence of the policy [4]. The additional improvement of DQN is experience replay. Without experience replay, only new experiences are used in training and discarded right afterwards. Therefore important but rare experiences are almost immediately forgotten and the updates are not independent and identical distributed but strongly correlated. To address this problem, an experience buffer is implemented. There

the experiences are stored and then at training time sampled uniformly at random. Usually a simple FIFO algorithm is being used. But there are more sophisticated methods for this, one of those is discussed later.

3 Rainbow / DQN Extensions

3.1 Double DQN

Anna Sterzik

Q-learning and DQN tend to learn overestimated action values. During the maximization over the action choices, values are rather over than underestimated. This isn't necessarily a problem if they are uniformly overestimated or if interesting experiences are overestimated. However, in DQN overestimations differ for different actions and states. Combined with bootstrapping, this results in the propagation of wrong values and thus to worse policies [2]. To reduce those overestimations, the Double Q-learning algorithm [2] is used. The main idea of the Double Q-learning algorithm is to decouple value selection and evaluation. DQN, with separate online and target networks provides an excellent framework for this decoupling: The online network is used to select an action from the action choices via maximization whereas the target network evaluates the actions to generate the Q-values. The resulting double DQN yields more accurate values and hence leads to better policies than DQN [2]. Updating double DQNs is similar to updating DQNs when the target is rewritten as:

$$Y_t^{\text{Double DQN}} \equiv R_{t+1} + \gamma Q\left(s_{t+1}, \underset{a}{\operatorname{argmax}}\, Q\left(s_{t+1}, a; \theta_t\right); \theta_t^-\right). \tag{10}$$

θ_t^- and θ_t are the weights of the target and online network respectively.

3.2 Prioritized Experience Replay

Anna Sterzik

In standard experience replay, the agent is forced to pick experiences uniformly from all experiences in its memory. Therefore all experiences are sampled with the same frequency that they were originally encountered. This is not necessarily good for the learning process, as some experiences might not hold any valuable information for the agent but occur very often while other rare situations could be crucial for learning.

This can be improved by prioritized experience replay [7]. Here every experience in the buffer gets a priority according to its TD-error. The TD-error measures the difference between the actual Q-value and the Target-Q-value, so if experiences with bigger TD-errors are provided with bigger priorities, experiences from which there still is a lot to learn are favoured. The priority p_i is determined from the TD-error δ_i according to:

$$p_i = |\delta_i| + \epsilon. \tag{11}$$

Hereby $\epsilon > 0$ denotes a small parameter to ensure, that every experience has a priority bigger than zero and thus can be picked for a sample batch. New experiences are always added with maximum priority to the memory. Problematic with this greedy approach is, that only experiences that are picked for learning get their priority updated. Therefore experiences with low initial priority might, because of the buffer memory structure be removed from memory before they could have been picked for learning. The buffer is also very sensitive to noise spikes [7].
To overcome this problems stochastic prioritization is used and p_i adjusted according to:

$$P(i) = \frac{p_i^\alpha}{\sum_k p_k^\alpha}. \tag{12}$$

Here $\alpha \, \epsilon \, [0, 1]$ is another parameter which adjusts the amount of prioritization that is used. For $\alpha = 0$ we get the uniform case (no prioritization), whereas $\alpha = 1$ leads to greedy prioritization.
Stochastic prioritization introduces bias to our model. This needs to be considered for updating the model, because it could change the solution the model is converging to [7]. To correct this, importance sampling weights (IS weights) are introduced:

$$w_i = \left(\frac{1}{N} \cdot \frac{1}{P(i)} \right)^\beta, \tag{13}$$

with N being the replay buffer size and $\beta \, \epsilon \, [0, 1]$ being a hyperparameter for adjustment of the bias. For $\beta = 1$ the bias gets fully compensated. This is most important at the end of the training process. Therefore β starts at an initial value and is then being annealed during training.

An efficient data structure for the memory is crucial for good performance.

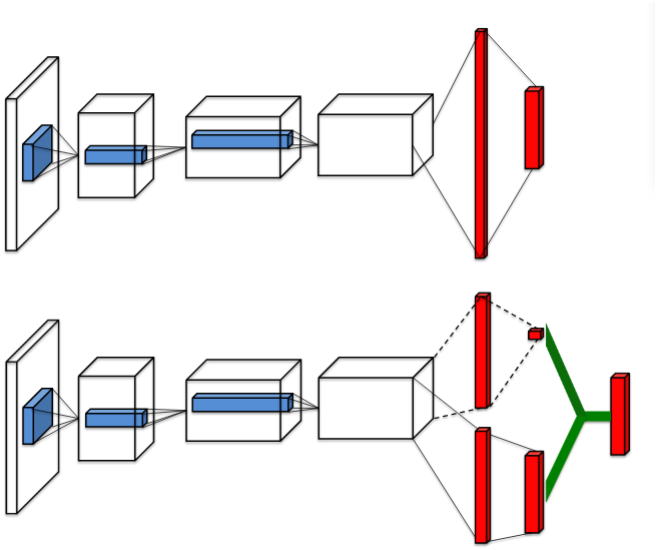


Figure 2: Top: Standard Deep Q-network. Bottom: Dueling DQN with two separate streams [11].

To guarantee this, we implemented a sum tree to store the data, where searching is of complexity $O(1)$ and updating of complexity $O(\log N)$. A sum tree is a binary tree, in which the parent node values are the sum of the child node values. In our sum tree, the transition priorities were saved in the leave nodes. Therefore the root holds the total priority. An array was used to hold the associated data values to the priorities. For the purpose of sampling the total priority is divided into k priority ranges, with k being the number of experiences in one sample. From each of these priority ranges one value is sampled uniformly and its corresponding leave node is searched. The data belonging to this priority is than used for the sample.

3.3 Duelling networks

Anna Sterzik

The idea behind duelling networks [11] is, that for some states only the state-value function is important while for others the chosen action is crucial.
Consider a small toy problem: Our agent needs to catch coins which are falling down form above, he can either move right or left or wait to catch them. In some states there are no coins at all. In that state it is not important which action is chosen, whereas for other states its is. To exploit this

the neural network is split into two streams: the action and the state-value stream, as can be seen in figure 2. The network is split after the convolutional layers. Therefore the two streams consist of linear layers. The benefit of this is, that it is possible to get separate estimations for state-value function and action-value function. Hereby the state-value $V(s;\theta,\beta)$ is a scalar property while the action-vector $A(s,a;\theta,\alpha)$ has the dimension of the quantity of possible action choices, in our case six. θ are the weights of the convolutional layers, while α and β are the weights for the A- and the V- stream respectively.
For getting the Q-value A and V need to be recombined. It is not a good idea though to simply add them together:

$$Q(s,a;\theta,\alpha,\beta) = V(s;\theta,\beta) + A(s,a;\theta,\alpha). \tag{14}$$

In that case it would get impossible to retrieve V and A from Q uniquely. One could add a constant to A and subtract it from V without changing Q. It is better to calculate Q by

$$Q(s,a;\theta,\alpha,\beta) = V(s;\theta,\beta) + \left(A(s,a;\theta,\alpha) - \frac{1}{|\mathcal{A}|} \sum_{a'} A\left(s,a';\theta,\alpha\right) \right), \tag{15}$$

where A and V keep their identifiability and the optimization becomes more stable [11].

3.4 Noisy Networks

Anna Sterzik

So far we did use an ϵ-greedy policy for exploration. Another technique which has been found to produce better result for many of the Atari games are Noisy Nets [1]. They add parametric noise to the weights and thus aid exploration without the need to pick random actions as part of a policy (e.g. ϵ-greedy). This is very convenient, because there is no need to tune additional hyper parameters as the reinforcement learning algorithm tunes the weights automatically. Consider a neural network $y = f_{\theta_n}(x)$. Where θ_n are the noisy weights. A linear layer of a neural network can be written as:

$$y = wx + b, \tag{16}$$

whereas a noisy linear layer can be written as:

$$y = \left(\mu^w + \sigma^w \odot \varepsilon^w\right) x + \mu^b + \sigma^b \odot \varepsilon^b. \tag{17}$$

Here x is the input, w, and $\mu^w + \sigma^w \odot \varepsilon^w$ are the weights and b and $\mu^b + \sigma^b \odot \varepsilon^b$ are the biases for linear and noisy linear layer respectively. All of the named parameters are trainable except for ε^w and ε^b which are noise random variables. We chose factorized gaussian noise for the distribution of the ε parameters, as it reduces computation time for random number generation, which is important for single thread-agents such as ours [1]. Here only one independent noise per input and another independent noise per output is needed, in contrast to independent Gaussian noise, where one independent noise per weight would be required. We factorized ε^w to $\varepsilon^w_{i,j}$. The noise random variables can then be written as:

$$\begin{aligned} \varepsilon^w_{i,j} &= f(\varepsilon_i)\, f(\varepsilon_j) \\ \varepsilon^b_j &= f(\varepsilon_j) \end{aligned}, \tag{18}$$

where we used

$$f(x) = \operatorname{sgn}(x)\sqrt{|x|}. \tag{19}$$

The parameters $\mu_{i,j}$ were initialized as samples from a random uniform distribution $\left[-\frac{1}{\sqrt{p}}, +\frac{1}{\sqrt{p}}\right]$ with p being the number of inputs for the noisy linear layer. $\sigma_{i,j}$ were set as $\sigma_{i,j} = \frac{\sigma_0}{\sqrt{p}}$.
For the Noisy Networks implementation we replaced the Fully Dense layers of the state-value and action streams by Noisy layers. [1]

3.5 Multi-Step Learning

Johannes Vogt

In standard Q-Learning, the Q-function is updated with a single reward R_{t+1} and the Q value estimate for s_{t+1} (6). This can be extended to multi-step targets[8] by accumulating the rewards $R_{t+1}, \ldots, R_{t+n}$ and the Q value estimate for s_{t+n+1}. We get the target

$$Y_t^{\text{Multi-Step Q}} \equiv \sum_{k=0}^{n-1} \gamma^k R_{t+k+1} + \gamma^n \max_a Q(s_{t+n}, a; \theta). \tag{20}$$

When combined with the Double DQN target (10), we get

$$Y_t^{\text{Multi-Step DDQN}} \equiv \sum_{k=0}^{n-1} \gamma^k R_{t+k+1} + \gamma^n Q\left(s_{t+n}, \operatorname*{argmax}_a Q(s_{t+n}, a; \theta); \theta^-\right). \tag{21}$$

Q-Learning is thereby similar to multi-step Q-Learning with $n = 1$. Higher n lead to faster propagation of reward information to previous time steps which can be useful or even a necessity if the rewards are sparse over a long number of time steps. On the other hand Q may not be able to approach the correct $Q*$ values for an arbitrary policy[6]. So a fine-tuned n can lead to faster learning.

4 Training

4.1 Feature space

Johannes Vogt

To capture the spatial relations between objects on the field, we use a 2D convolutional neural network. This network takes a matrix $X \in \mathbb{R}^{w \times h \times c}$ as input, where w and h are width and height respectively and c is the number of channels. In the case of bomberman, we have $w = h = 17$. For the arena and objects in the game world we give each its own channel (Figure 3):

Walls	$\in \{0, 1\}$
Crates	$\in \{0, 1\}$
Self	$\in \{0, 1, 2\}$ (omitted when centring inputs)
Others	$\in \{0, 1, 2\}$
Bombs	$\in [0, 1]$
Explosions	$\in \{0, 1\}$
Coins	$\in \{0, 1\}$

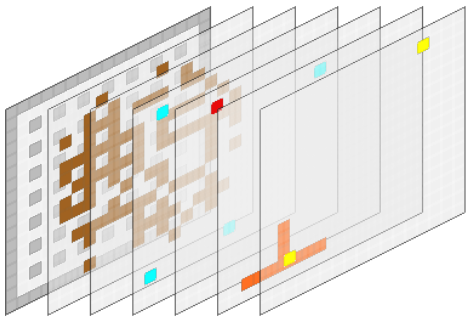
For all channels with $X_{x,y}|_c \in \{0, 1\}$,

$$X_{x,y}|_c = \begin{cases} 1 & \text{if the entity exists at } (x, y), \\ 0 & \text{otherwise.} \end{cases}$$

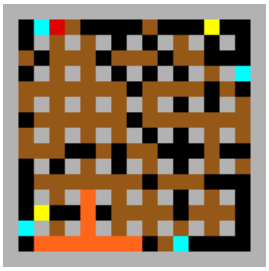
The channels *self* and *others* both encode information about the respective players as

$$X_{x,y}|_{c=self/others} = \begin{cases} 2 & \text{player with bomb,} \\ 1 & \text{player without bomb,} \\ 0 & \text{otherwise.} \end{cases}$$

There is no differentiation between the different opponents in *others*, since the state only consists of the current time step. The channel *self* gets omitted when we later centre the inputs.



(a) Visualization of the separate channels



(b) All channels combined

Figure 3: Input channels (from left to right): walls (gray), crates (brown), self (turquoise), other players (turquoise), bombs (red), explosions (orange), coins (yellow)

For bombs (x, y, t), t being the time left until the bomb explodes, we calculate

$$X_{x,y}|_{c=\text{bombs}} = 1 - \frac{t}{\text{bombtimer} + 1} \quad \forall (x, y, t) \in \text{bombs}$$

with

$$X_{x,y}|_{c=\text{bombs}} = 0 \quad \forall (x, y, t \geq 0) \notin \text{bombs}.$$

So with bombtimer $= 4$, if $X_{x,y}|_{c=\text{bombs}} = 0.2$, the bomb has just been planted, and if $X_{x,y}|_{c=\text{bombs}} = 1$, the bomb will explode this turn. While an explosion is active for two turns, this includes when the bomb timer hits zero, so the explosion is only visible to agents during one turn.

The matrix X is sparse in all but two channels, but this feature space captures the spatial correlations well.

A crucial part of the input is the position of the agent itself. If the player position is simply transformed into a channel, it is reduced to one of many variables and it can be difficult to pick up the importance of this specific variable. So to simplify learning for our agent, the agent was centred in the feature space, so that it stays at a fixed position while the environment (coins, crates, other agents) move around it. To implement this, the size of the feature space is increased to $(2w - 1, 2h - 1, c)$ and the agent placed in the middle of the new board. As the position of the agent is now known implicitely, we can remove the corresponding channel *self*. This also removes knowledge from the state about whether the player has a bomb available,

but the knowledge is unimportant for most situations because we later mask the actions.

4.2 Augmented data

Anna Sterzik

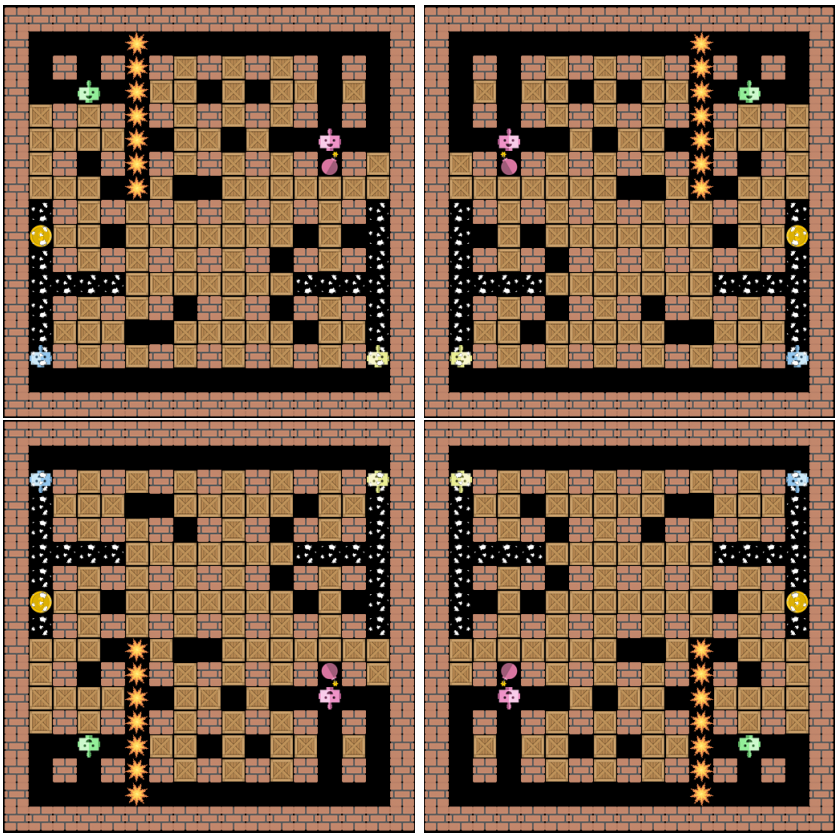


Figure 4: Data augmentation:
Upper left: original; Upper right: horizontal mirroring;
Lower left: vertical mirroring; Lower right: combined mirroring

As the inputs for our bomberchamp agent are symmetric, we wanted to use data augmentation to increase the number of samples for training and to make learning more symmetric. From each original sample, three augmented

samples were created. For augmentation, we had to mirror the environment and to change action choices accordingly. The augmented environment consisted of horizontal mirroring, vertical mirroring and a combination of both as seen in figure 4. For horizontal mirroring the agent choices left and right were exchanged, for vertical mirroring up and down and for the combination both were swapped.
Data augmentation did not work as well as we expected, so it was not included in our final implementation.

4.3 Invalid actions

Johannes Vogt

We are only selecting from valid actions, since we found that otherwise the invalid actions can cause a lot of trouble. Essentially invalid actions are equal to WAIT, meaning that the rewards for different actions get mixed up. Another solution would have been to penalize invalid actions and let the agent learn to avoid them, but since filtering out invalid action is really easy and fast, we use that. Invalid actions include placing a bomb when the agent does not have a bomb and movements to tiles with walls, crates, agents and bombs. During training movement into explosions is allowed, but during play this is prevented.

4.4 Auxiliary Reward Design

Johannes Vogt

As we use only movement actions for the singleplayer coin collection mode, we do not need any auxiliary reward. Starting from the singleplayer mode with coins in crates, we found that the agent does not learn well with only having coins as rewards. So we give the agent an auxiliary reward of 0.2 for destroying crates. Interestingly the agent does learn to destroy crates and collect coins pretty well even without a penalty for death. But there are still a lot of games where the agent bombs itself in the beginning.

With a penalty for death, the agent learns to avoid death by choosing WAIT. When penalizing this action, the resulting agent learns to destroy the crates and collect coins, but after training for a long time it learns to alternate UP-DOWN and RIGHT-LEFT as a more elaborate version of WAIT to avoid death.

So we chose to give a reward of -0.2 for WAIT and track the last moves $m_{t-k} = (x_{t-k}, y_{t-k}, a_{t-k})$ for $k = 1, \ldots, 20$. The agent then gets penalized for

the number of moves that are similar to the current move with

$$R_{\text{similar\_moves}} = -\frac{1}{20}\sum_{k=1}^{20}(m_t = m_{t-k} \to 1) \wedge (m_t \neq m_{t-k} \to 0).$$

For dying the agent gets a reward of -5.

4.5 Minigames

Johannes Vogt

As even the seemingly simple task of collecting coins in the bomberman arena can be challenging, we made an even simpler minigame. The minigame consists of collecting coins on an otherwise empty arena. Consequently, the feature space is $X \in \{0,1\}^{w \times h \times 1}$ and we can limit the actions to movement in the directions that do not lead outside the arena. We can also vary w, h and the number of coins n_{coins} to see how a agent is affected by the arena size and coin count.

The game is terminated when all coins have been collected or after a certain duration which is calculated as

$$T_{max} = (w + h) * n_{\text{coins}}.$$

This is done to ensure that every coin can be collected, but there is a time limit to collect them.

Having this minigame helped us debug our reinforcement learning agent and find a suitable network for the next task.

4.6 Self-Play

Johannes Vogt

We use self-play to train the agent in a four player free-for-all environment. The agent plays against itself, a past version of itself or the heuristic *simple_agent*. Since the players use the same model, the experiences for the player copies can be saved in a shared experience replay buffer. The *simple_agent* experience also gets saved in the buffer with the Q value estimation being calculated using the current player model. The past versions only serve as opponents with no data added to the buffer.

For each multi-player game, we take the original agent and select the other three agents randomly from the copies, past selves and *simple_agents*.

Shared network		
Dense	64 units	relu
Dense	64 units	relu
Advantage / value stream		
NoisyDense	64 units	relu

(a) Dense 64

Shared network		
Dense	256 units	relu
Dense	256 units	relu
Advantage / value stream		
NoisyDense	256 units	relu

(b) Dense 256

Figure 5: Simple fully connected (dense) network. The last value and advantage stream layer is omitted.

The agent also plays a certain percentage of the games solo, as we found that this helps in keeping the agent movement efficient.

Further, we randomize the game properties for each episode during training. A percentage of games is played on the coin variant with crate density $\rho_{crates} = 0$ and maximum game duration $T_{max} \in [100, 200]$. The rest of the games is played with $\rho_{crates} \in [0.5, 1]$ and $T_{max} \in [200, 400]$. This randomization is done for robustness, so that the agent does not overfit to the exact game properties.

The agent reward is postprocessed by subtracting the average reward of its opponents. This leads to the agent actually learning to beat the oppenents instead of just collecting points and allowing the others to do the same. It encourages the agent to play better than its opponents.

The self-play strategy was heavily inspired by OpenAI Five[5]. Playing against past versions helps prevent "strategy collapse" which we also noticed during early versions. Randomization helps exploration of the strategy space. Subtracting average reward of opponents helps prevent positive-sum strategies.

5 Observations

5.1 Network Architecture

Johannes Vogt

Having introduced our minigame in 4.5, we tried different network architectures to see what is necessary for an arena of size $(17, 17)$.

Shared network				
	n_c	f	s	
Conv2D	32	8	4	relu
Conv2D	64	4	2	relu
Conv2D	64	3	1	relu
Advantage / value stream				
NoisyDense	512 units			relu

(a) Conv from Rainbow paper[3]

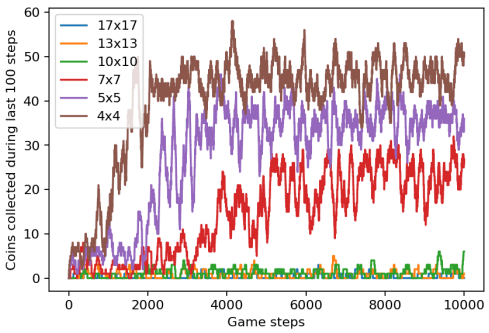
Shared network				
	n_c	f	s	
Conv2D	32	1	1	relu
Conv2D	64	4	2	relu
Conv2D	64	3	2	relu
Advantage / value stream				
NoisyDense	512 units			relu

(b) Modified convolutional NN

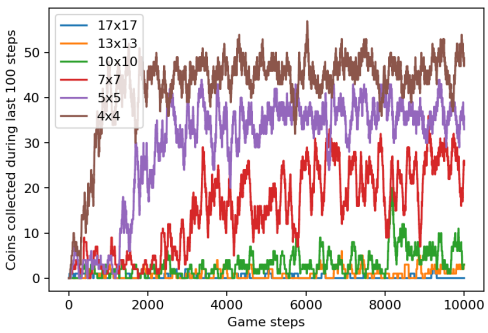
Figure 6: Convolutional network with number of channels n_c, filter size f and stride s. The last value and advantage stream layer is omitted.

We chose a simple fully connected network with different numbers of neurons (Figure 5) for our first tests and expanded on it with the convolutional network that was used on the Atari 2600 benchmark in the Rainbow DQN paper[3] (Figure 6a). Since the input for the Atari benchmark was raw pixels with $w = h = 84$, this might not be suitable for our problem with meaningful inputs of size $w = h = 17$. So we modified the convolutional network to mimick the dimensions of the different layers of the original (Figure 6b).

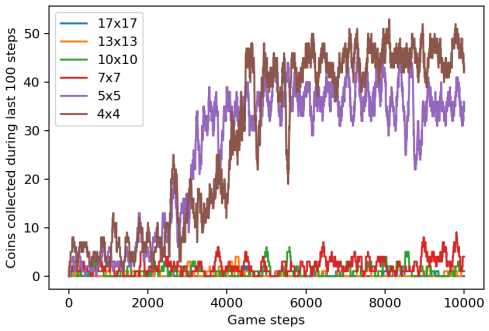
Figure 7 shows the performance of the agents on different arena sizes of the minigame. While the dense networks (7a, 7b) learn very quickly for small boards, they have difficulties capturing boards of size $w = h \geq 10$. When trying the convolutional network designed for Atari games (7c), its performance is even worse than the dense networks. This is likely due to the first convolutional layer having a filter size of 8 and a stride of 4. For raw pixels sparse with information, a high filter size and stride can help summarize the information and reduce the dimensions. But on our input it may lead to weakening the important values. So for our modified convolutional net, we set the filter size f and stride s to 1. The resulting 17×17 layer output is close to the originals 21×21 layer output, making the remaining network usable. For the minigame, a layer with $f = 1$, $s = 1$ has no impact because there is only one input channel. But our bomberman input has six channels, so the first layer can transform the information at each location into a more convinient form. The result can be seen in figure 7d, as it can collect coins efficiently for at least $w = h = 10$ after a short while. If trained longer, this



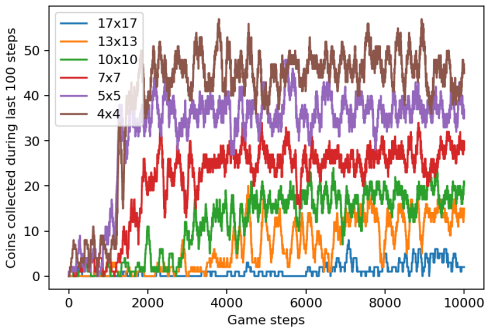
(a) Dense 64 (5a)



(b) Dense 256 (5b)



(c) Conv from Rainbow paper[3] (6a)



(d) Conv modified (6b)

Figure 7: Agents with different network architectures playing the minigame introduced in 4.5 for arena sizes from 4×4 to 17×17. There are three coins that can be collected per episode. The graphs show coins collected during the last 100 steps, continuing over episodes. For a big arena this number is naturally lower because of the large distances between coins, but it can be seen when and if the agent is plateauing.

Shared network

a stream				b stream				c stream			
Cropping2D 14				Cropping2D 10							
	n_c	f	s		n_c	f	s		n_c	f	s
Conv2D	64	3	1	Conv2D	64	3	1	Conv2D	32	1	1
				Conv2D	64	3	2	Conv2D	64	4	2
								Conv2D	64	3	2
								MaxPool			2
								Conv2D	64	3	1

Advantage stream			Value stream		
	units			units	
NoisyDense	512	relu	NoisyDense	512	relu
NoisyDense	D	relu	NoisyDense	1	relu

Figure 8: Convolutional net focused on central region as shown in figure 1. Each convolutional layer is followed by a *relu* activation.

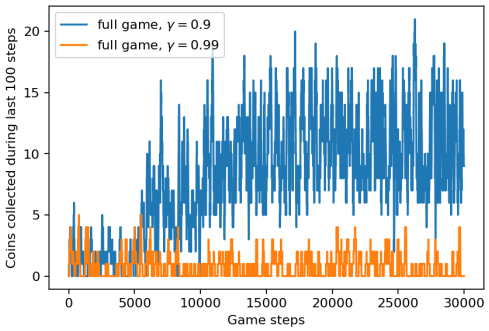
network can also solve the minigame for $w = h = 17$.

After testing the modified convolutional net on the minigame and the singleplayer bomberman coin collection, we found that it does not train well for $\gamma = 0.99$ as can be seen in figure 9a. So we designed a network that can better handle the feature space and maze structure and is more robust to changes in γ. So we designed the focused net as shown in figure 1. This network architecture works very well as seen in figure 8 and has continued to work for the singleplayer bomberman mode 10 and also the multiplayer game.

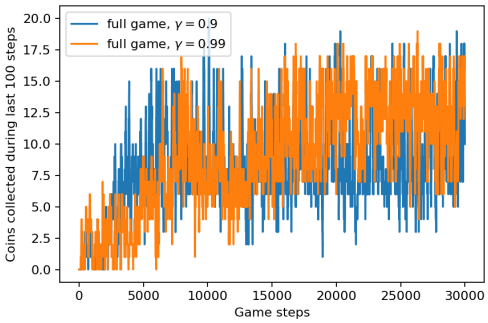
6 Summary

Johannes Vogt

It was quite difficult to get started, implementing the agent to train even the simplest task. It can be unclear at times, especially with a complex agent such as Rainbow DQN, whether the agent does not learn due to a bug or simply due to bad hyperparameters or an unfit network architecture. But



(a) Conv modified (6b)



(b) Conv focused (8)

Figure 9: Agents with different network architectures playing the bomberman game in singleplayer coin collection mode with discount factors $\gamma = 0.9$ and $\gamma = 0.99$.

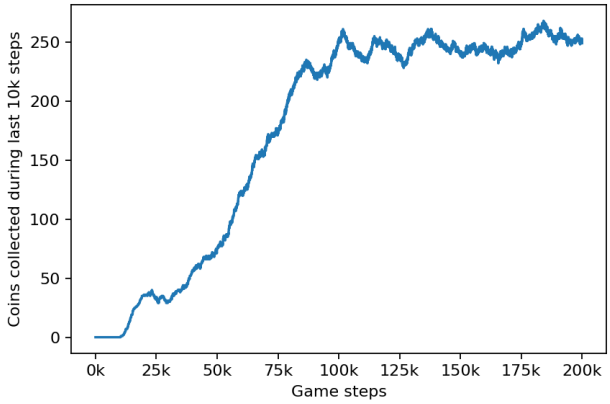


Figure 10: Focused convolutional net (8) on singleplayer bomberman.

once the task of collecting coins was solved, the next tasks followed soon after. As it turns out, the results can be good even if there are still some serious bugs. During most of our self-play runs the agent only saw a bomb the second turn after it was planted.
Neural networks themselves already have a good number of hyperparameters to tune. Reinforcement learning also adds some and once self-play is factored in, there are a lot of possibilities. We found that the agent is quite sensible to auxiliary rewards.

In the end, we did not beat the *simple_agent* consistently, but the final agent is able to destroy crates, collect coins and also bomb an opponent from time to time.

6.1 Improvements

Johannes Vogt

There are a number of points likely to increase the agents performance:

- Tuning hyperparameters and adjusting the neural network. As we did not have a lot of time or resources during the final stretch, we did not test alternative network configurations and only a few hyperparameters. This would likely improve the agent quite a bit.

- The current network has a large concatenated flattened layer before the advantage and value streams. So most of the parameters (11.8M / 11.96M) are concentrated there. This seems very inefficient computationally and could probably be reduced by a lot, helping the agent act faster and avoid the -1 penalty for being the slowest agent. It would also help reduce the size of the weights.

- Having different agents play with different auxilliary rewards, like the tendencies recently introduced in AlphaStar[10] where every agent has its own personal objective. This would require a suitable skill evaluation metric and matchmaking for a four player game, which we did not have time for in the end.

- The agent currently has no information of what happened in previous time steps. While bomberman is a game with full information, the concrete timer of when bombs are available for agents is currently a hidden state and the expected actions are still sequences. So using a

recurrent neural network might be useful, but it is difficult to combine with a replay buffer.

- We did not implement the distributional extension of Rainbow DQN. This could improve the performance, but in the Rainbow paper[3] it seemed to be only helpful after the agent has already trained for some time and surpassed human performance.

6.2 Improvements for Game Setup

Johannes Vogt

All in all the project was very cool and it motivated us to dive into reinforcement learning.
But there are a few points which made things difficult to start:

- The provided framework is quite cumbersome to work with. While this may be true for many real world applications and it is necessary to work around it in those cases, for educational purposes it would probably be better to provide something easily usable to be able concentrate on the reinforcement learning task. Due to this, we reimplemented the environment ourselves.
  - Currently main.py, settings.py and possibly callbacks.py have to be changed for a simple switch between train and test mode.
  - Separate rendering and environment so that the environment can be called from e.g. a jupyter notebook on a cloud service that provides free GPUs for training.
  - Provide a version without Multiprocessing. MP may be useful for the tournament, but it is difficult to work around when training, so essentially one has to reimplement the environment.
- The task, even the simplest subtask, is quite difficult because of the large arena size and maze structure. Providing an even simpler game to get started would help guide the students to the bigger tasks, see our Minigame section.
- If reinforcement learning is covered a bit earlier in the lecture and there are exercises for it, the final project can be made more difficult. As of now, it is probably too difficult if one has no previous knowledge of the field.

That said, the project is motivating and was a lot of fun.

References

[1] Meire Fortunato et al. "Noisy Networks for Exploration". In: *CoRR* abs/1706.10295 (2017). arXiv: 1706.10295. URL: http://arxiv.org/abs/1706.10295.

[2] Hado van Hasselt, Arthur Guez, and David Silver. "Deep Reinforcement Learning with Double Q-learning". In: *CoRR* abs/1509.06461 (2015). arXiv: 1509.06461. URL: http://arxiv.org/abs/1509.06461.

[3] Matteo Hessel et al. "Rainbow: Combining Improvements in Deep Reinforcement Learning". In: *AAAI*. 2018.

[4] Volodymyr Mnih et al. "Playing Atari with Deep Reinforcement Learning". In: *CoRR* abs/1312.5602 (2013). arXiv: 1312.5602. URL: http://arxiv.org/abs/1312.5602.

[5] OpenAI. *OpenAI Five*. https://blog.openai.com/openai-five/. 2018.

[6] Jing Peng and Ronald J. Williams. "Incremental multi-step Q-learning". In: *Machine Learning* 22.1 (Mar. 1996), pp. 283–290. ISSN: 1573-0565. DOI: 10.1007/BF00114731. URL: https://doi.org/10.1007/BF00114731.

[7] Tom Schaul et al. "Prioritized Experience Replay". In: *CoRR* abs/1511.05952 (2015). arXiv: 1511.05952. URL: http://arxiv.org/abs/1511.05952.

[8] Richard S. Sutton. "Learning to predict by the methods of temporal differences". In: *Machine Learning* 3.1 (Aug. 1988), pp. 9–44. ISSN: 1573-0565. DOI: 10.1007/BF00115009. URL: https://doi.org/10.1007/BF00115009.

[9] Richard S. Sutton and Andrew G. Barto. *Introduction to Reinforcement Learning*. 1st. Cambridge, MA, USA: MIT Press, 1998. ISBN: 0262193981.

[10] Oriol Vinyals et al. *AlphaStar: Mastering the Real-Time Strategy Game StarCraft II*. https://deepmind.com/blog/alphastar-mastering-real-time-strategy-game-starcraft-ii/. 2019.

[11] Ziyu Wang, Nando de Freitas, and Marc Lanctot. "Dueling Network Architectures for Deep Reinforcement Learning". In: *CoRR* abs/1511.06581 (2015). arXiv: 1511.06581. URL: http://arxiv.org/abs/1511.06581.