University of Applied Sciences
Cologne

Research Project

Hyperparameter search for playing StarCraft II with DQN with non-spatial features and performance comparison between DQN, DDQN and Ape-X

Florian Soulier

supervised by

Prof. Dr. Beate Rhein

August 14, 2019

Contents

List of Figures

List of Algorithms

List of Abbreviations

A3C	Asynchronous Advantage Actor Critic
AI	Artificial Intelligence
API	Application Programming Interface
CPU	Central Prozessing Unit
DQN	Deep-Q-Network
DDQN	Double DQN
GPU	Graphics Processing Unit
IS	Importance Sampling
LSTM	Long short-term memory
ML	Machine Learning
MP	Markov Process
MDP	Markov Decision Process
MRP	Markov Reward Process
NPC	non-player character
RAM	Random Access Memory
RTS	Real Time Strategy
RL	Reinforcement learning
SARSA	State-action-reward-state-action

SC2	StarCraft II
SC2LE	StarCraft II Learning Environment
SGD	Stochastic Gradient Descent
TPU	Tensor Processing Unit

List of Tables

Artificial Intelligence (AI) is a frequently mentioned term in today's news and political statements similar to terms like "industry 4.0" or autonomous driving. Lately the government of China has said, it will devote $5 Billion to AI technologies and businesses. This shows the economic relevance of machine learning. Whereas artificial intelligence is an umbrella term for a big class of similar problems but very different methods, this report will deal with a specific field of the sub domain of machine learning. In particular, this paper deals with playing the real-time strategy game StarCraft II with the help of reinforcement learning.

1 Introduction

In this chapter the terms concerning the domain of machine learning will be introduced and be delimited from the special domain of reinforcement learning. Afterwards the motivation for this project and the objectives of this work will be presented.

1.1 Types of machine learning

The objective of Machine Learning (ML) is to find information by experience and to develop a solution to perform a given special task optimally, without using explicit instructions of how to solve the task, like in classical programming. Instead a mathematical model of the problem is trained with input data in order to make it able to make decisions without being explicitly programmed to perform the task. In the field of machine learning, we distinct between three sub domains, which are supervised learning, unsupervised learning and reinforcement learning.

Supervised learning describes the learning from data with labels. A common task of this kind of machine learning is the classification. In this field there are commonly huge data sets available from which the model can learn. These data sets contain the so called input data, which is the data that is available to solve the problem and a so called label, which contains the right value. The data is uniformly distributed divided in a training set and a validation set. The model can now be trained with the training set by letting the model do the classification task of the given input and comparing the output with the label. The model will be adjusted by the loss, which is calculated from the difference between output and label value. After training the model can be validated by the validation set. This method is only applicable if there is a so called supervisor, which means, there has to be a big data set available and it has to be labeled by a human expert or a measurement system.

Unsupervised learning also learns from present data, but without labels. A common task is clustering. The goal of clustering is to find data that has similar properties or to find hidden structures in data. It is used to find *new* information. Therefore an unsupervised learning algorithm is applied to the data and the newly

found clusters have to be evaluated by a human. Common criteria that can be used to parameterize those algorithm are e. g. density or distance of data points.

Reinforcement learning however focuses on learning to act as a software agent in an environment where it has to learn to move or behave adequate. The agent is trained by a reward function, which gives positive reward for good actions and negative rewards respectively. The overall goal of the agent is to maximize the cumulative reward. This kind of machine learning, which learns with critic, suits for problems where no big data sets are available, or the environment is too complex to be described. This kind of learning is most similar to the learning process of a human.

1.2 Motivation and related work

Until 2012 Deep reinforcement learning was no big deal. Since the development of the backpropagation algorithm in the 1980s, only a few substantial results were achieved in the field of neural networks and they could only be trained with difficulty for bigger problems. Since then, the calculation power of computers has raised and there have been developed algorithms to do ML calculations on Graphics Processing Units (GPUs) and specialized Tensor Processing Units (TPUs) have been developed.

In 2013, Mnih et al. (DeepMind) published a paper called "Playing Atari with Deep Reinforcement Learning" [5] which lead to a renaissance of reinforcement learning with nonlinear function approximators like neural networks. The results of the paper were tremendous, the trained agents with the now popular RL algorithm Deep-Q-Network (DQN), performed extremely well on a wide range of Atari games. Since then, DQN has widely been used and adpoted.

In 2017 Vinyals et al., researchers of DeepMind and Blizzard published a paper called "StarCraft II: A New Challenge for Reinforcement Learning" [9] which introduced the StarCraft II Learning Environment (SC2LE), a framework for reinforcement learning on the popular real time strategy game StarCraft II (SC2). Since SC2 is a very complex problem for Reinforcement learning (RL), it is a good research object.

In 2018 and 2019 the research company DeepMind did and is currently doing research in this field. Lately they published a demonstration of their current state of research on YouTube. They demonstrated agents playing against professional players. The agents won all games and played on a very high level. This was a great motivation for me, to start research in this field.

1.3 Objectives of this work

This research project is part of an examination procedure. In the course of this project, a RL environment will be built, in which an agent will be trained with DQN to play so called mini-games, a simplified version of the full SC2 game. It will be investigated, if it is possible to achieve reasonable results with non-spatial observations of SC2 with DQN. Moreover a hyperparameter search will be performed to examine, if the results can be further improved with the right set of hyperparameters.

Since DQN has been invented in 2013 and several modifications and improvements have been published, it will be also investigated, if the successors Double DQN (DDQN) and Ape-X DQN can be applied to this problem. Because of the limited time, the algorithms will be fed with the tuned hyperparamters of DQN. In this context, it will will be investigated, if these algorithms can achieve similar results, without an own hyperparamter search.

In order to answer these questions, the theoretical aspects of the above described algorithms will be covered. Afterwards a hyperparameter search of DQN will be performed. Therefore the important hyperparamters training batch size, discount factor and learning rate will be tuned. Then, the tuned DQN hyperparamters will be applied to DDQN and Ape-X DQN. The results will be presented and compared.

2 Theory

In the following all theoretical aspects that are needed to follow this written report will be addressed.

2.1 Reinforcement Learning Fundamentals

2.1.1 Basic Terms

Reinforcement learning is learning what to do by maximizing a numerical reward signal. It is considered a software agent as shown in Figure 1. The agent interacts with an environment ε, typically it is a stochastic environment. Thereby it can interact with the environment by choosing an action A_t from a set of Actions A at a specific time step t. The environment changes its internal state, which can not be accessed by the agent, but the agent can sense the so called observation and a feedback of how good it is doing. The observation consists of a state signal S_t from a set of states S. The feedback of how good it is doing, consists of a reward signal R_t. These two signals are responded by the environment after every time step . In the following the terms will be explained in more detail.

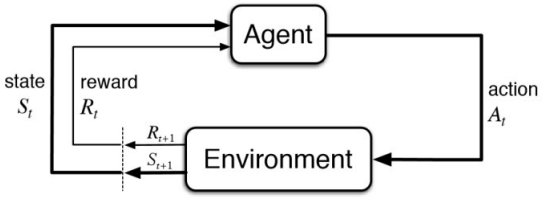


Figure 1: Schematic illustration of a reinforcement learning environment

The Reward R_t is a numerical feedback signal, which indicates how well the agent is doing at time step t. The agent's job is to maximize the cumulative reward G_t. The reward hypothesis says, that all goals can be described by the maximization of expected cumulative reward.

The History H_t is a sequence of actions, observations and rewards and can be described as follows: $H_t = (A_0, O_1, R_1), ..., (A_{t-1}, O_t, R_t)$. What happens next, depends on the history.

The State S_t (to be exact, the information state or Markov state) contains all useful information from the history. In other words, the whole history at time step t can be described by the state S_t. This can be achieved, if the state is Markov, which is described by the equation $P[S_{t+1}|S_t] = P[S_{t+1}|S_1, ..., S_t]$. Thus in reinforcement learning, it is not necessary to bookmark all visited states, but sufficient to store

the state, the agent has visited last. By considering that, it is necessary to give the agent all relevant information, that is available.

The Environment ε is the world, the agent lives in. It has internal procedures which can be described by states. If the agent can observe the internal state S_t^e directly, one says the environment is fully observable. This is formally described as Markov decision process (MDP). If the agent can observe the environment only indirect, one says the environment is partially observable, which is described as partially observable Markov Decision process (POMDP). In the last case, the agent has to construct its own state representation S_t^a. This report deals with a POMDP.

A RL agent includes typically three components. A **policy**, which determines the agent's behavior, a **value function**, which gives feedback how good each state and/or action is and optionally a **model**, which is the agent's representation of the environment.

A policy maps a state to an action. There are deterministic policies, which are described by $a = \pi(s)$ and stochastic policies which are described by $\pi(a|s) = P[A_t = a|S_t = s]$. In a perfect world, we can use a deterministic policy. If we, for example navigate through an ideal maze grid to find a way out of a labyrinth, we can map a state deterministic to an action. But if we are in a real world example, e. g. on a windy way, there could occur, that we perform an action "go ahead" but we come out a step to the left, because of the characteristic of the environment. If we take an action a, we will progress with some probability to state s, but also with some probability to state s'. This is where we have to use a stochastic policy.

If we look at policies, reinforcement learning algorithms can be further divided into on-policy and off policy learners. On-policy learners, as e. g. State-action-reward-state-action (SARSA), use the same policy π for policy learning and behavior. Off-policy learners, such as DQN use a policy π for learning and another policy μ for behavior.

A value function is a prediction of *expected* future reward. It is used to evaluate the goodness or badness of states. By that it gives feedback, what actions to choose, which is mathematically described by $v_\pi = E_\pi[R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + ... \mid S_t = s]$. The parameter γ describes the *discount factor*, which can be chosen between 0 and

1 and describes how far the agent looks in future to maximize the cumulative reward. For a discount factor of 0, only the immediate reward is important, for a discount factor of 1 all future rewards are important and the agent tries to chose actions that maximize the sum of all future rewards. In practice, a discount factor smaller that 1 is chosen, due to the fact, the agent can only partially observe the environment and thus might not have an exact representation of the internal state. In a financial background it could be also more preferable to gain immediate reward than the long term reward, and therefore use a discount factor clearly below 1.

A model is the representation of the environment. It predicts what the environment will do next. It predicts the next state, under the condition of choosing action a in state s. This can mathematically be described as following transition function: $P^a_{ss'} = P[S_{t+1} = s'|S_t = s, A_t = a]$. It also predicts the next expected immediate reward, which can be described by the following transition function: $R^a_s = \mathbb{E}[R_{t+1}|S_t = s, A_t = a]$. Models can e. g. described by Q-tables, where every state is a cell in a state matrix, we call it model-based learning. Models are used for planning, by which we mean any way of deciding on a course of actions by considering possible future situations before they are actually experienced. [8] This is usually the case, if the environment is fully observable and the number of states is small. This is done by dynamic programming techniques like value iteration and policy iteration.

In contrast to model-based methods, model-free methods are used for learning environments in which the learner is a explicit trial-and-error learner. These learners are often used in dynamic and/or complex learning environments, where the computation or storage would be too expensive or the environment is too complex to be described by a model. Model-free learning in contrast to that tries to approximate the value and policy function directly by omitting the model. The software agent acts accordingly to its policy and gathers experience. This experience is directly used to approximate the policy or value function. Thus, dynamic programming techniques cannot be applied anymore.

2.1.2 Exploration versus exploitation dilemma

In RL, there are two important terms, which describe the gathering of experience. The goal of RL is to discover a good policy from its experiences in the environment and thereby gather as much reward as possible. With exploration of the state

space, the agent finds more information about the environment, with exploitation the agent exploits known information about the environment, to maximize the cumulative reward. The exploration versus exploitation dilemma deals with the question, when and how long it has to exploit the environment, before concentrating on exploiting and maximizing reward.

An algorithm that only exploits from known information is the greedy algorithm. It is considered, that the estimated value will converge to the true value, mathematically described as $\hat{Q}_t(a) \approx Q_t(a)$. The greedy algorithm chooses the action with the highest expected reward, which can be described as follows: $a_t^* = \underset{\forall a \in A}{\operatorname{argmax}}\ \hat{Q}_t(a)$. In this case, the greedy algorithm can lock on a local maximum forever, never finding better policies.

To overcome this problem, the algorithm is modified by a random action choice with probability ε (ε-greedy algorithm). Thus, the algorithm will choose a random action with probability ε and follow its policy with probability $1 - \varepsilon$. By that it is ensured, that the agent will explore forever. In practice, the parameter ε is decayed over ascending episodes, so the agent will focus finally more on maximizing the reward, instead of exploring the environment.

2.1.3 Markov Decision Process

A Markov Decision Process (MDP) formally describes an environment for reinforcement learning, where the environment is fully observable. Almost every RL problem can be formalized as a MDP, even partial observable problems can be converted to MDPs, which makes this tool very powerful. As described before a state is Markov if and only if $P[S_{t+1}|S_t] = P[S_{t+1}|S_1, ..., S_t]$. This yields that if the actual state is known, the history can be thrown away. Thus, for a Markov state s and its successor state s' the state transition probability is defined by $P_{ss'} = P[S_{t+1} = s'|S_t = s]$. A Markov Process (MP) $< S, P >$ is a memoryless random process i. e. with random states from a finite set S and state transition probability matrix P . For better understanding, in Figure 2 a simple tunnel is shown.

Every cell correspondents to a state, the agent can only be on one cell respectively state at one time. In the next figure a transition probability matrix is shown.

As it can be seen, not every state can be accessed from every state directly. For

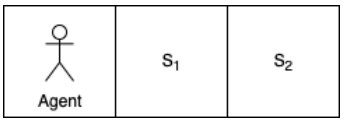


Figure 2: A simple tunnel example for a Markov Process in that the agent has to find the exit. The agent starts in the left (state S_0) and the agent can choose from action left and right to surpass the tunnel. The most right position (state S_2) is the exit.

S0	0.1	0.9	
S1	0.4	0.2	0.4
S2		0.9	0.1
	S0	**S1**	**S2**

Figure 3: A transition probability matrix for a simple Markov Process. The probabilities of every row sum up to 1.

these cases, the probability is 0 (left blank in the transition probability matrix). In Figure 4 it can be seen a MP in form of a graph.

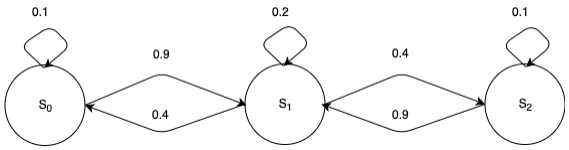


Figure 4: A transition graph for a simple Markov Process. All probabilities of leaving arrows of a state sum up to 1.

The Markov Reward Process (MRP) extends the Markov Process by a reward function and a discount factor. The MDP finally extends the MRP by a finite set of actions A. It is defined by a tuple $< S, A, P, R, \gamma >$, with being S a finite set of states, A a finite set of actions, P the three dimensional transition probability matrix with transition probability $P^a_{ss'} = P[S_{t+1} = s' | S_t = s, A_t = a]$, R a reward function, defined by $R^a_s = \mathbb{E}[R_{t+1} | S_t = s, A_t = a]$ and a discount factor $\gamma \in [0, 1]$. For better understanding the MP from Figure 4 is extended to a MDP, as shown in Figure 5.

As it can be seen in the Figure, there is a set of states $S = \{S_0, S_1, S_2\}$ in which a software agent can move. The agent always starts in S_0 and has an action space of moving left and right $A = \{left, right\}$. The *expected* reward is coupled to a

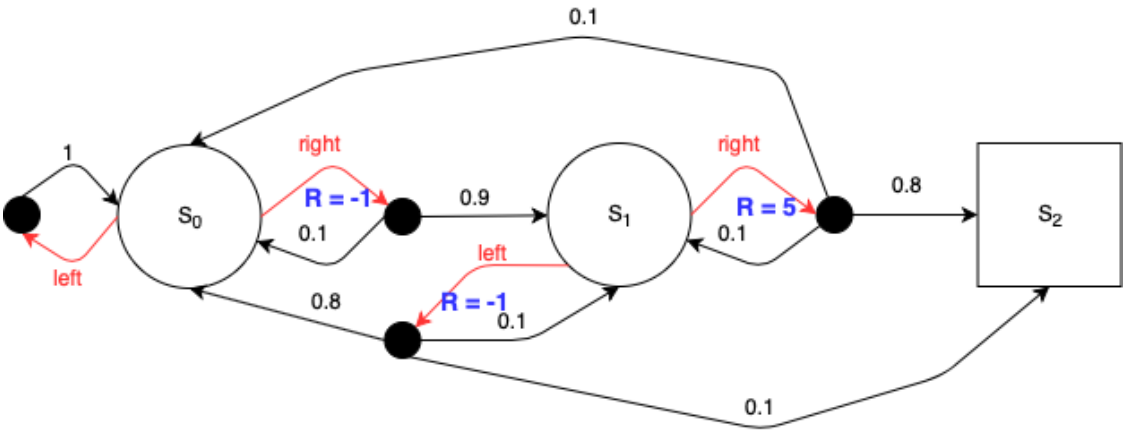


Figure 5: A transition graph for a simple Markov Decision Process.

specific state and a chosen action. In this example every action from every state will give a expected reward of -1, but from state S_1 and action r, where the expected reward is $+5$. The success of chosen actions is probabilistic, for example if the agent is in S_1 and chooses to move right, then he will success to S_2 with a probability of 0.8. But with a probability of 0.1 he will move left which leads to S_0 and with p = 0.1 he will stay in the state and do nothing. With that, real-world related problems, such as "windy" environments, can be covered.

As it can be seen in Figure 6, the two-dimensional transition probability matrix is extended by a third dimension, the actions. Other than the two-dimensional probability matrix for the MP not the sum of all transitions, that leave a state s_i sum up to 1, but all possibilities of possible outcomes after taking a specific action sum up to 1.

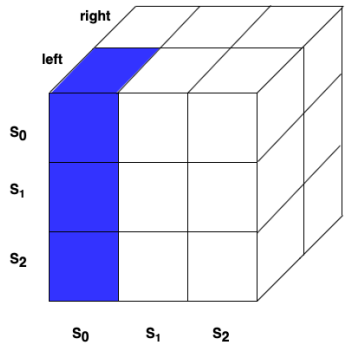


Figure 6: A transition probability matrix for a simple Markov Decision Process. All probable outcomes of choosing an action sum up to 1.

In this report, a partial observable Markov decision process is discussed. POMDPs are MDPs with hidden states and are defined by a tuple $< S, A, O, P, R, Z, \gamma >$, with O being a finite set of observations and Z being an observation function with $Z^a_{s'O} = \mathbb{P}[O_{t+1} = o | S_{t+1} = s', A_t = a]$. Since not all states are known, a new

construct, so called belief states are needed to describe the POMDP. A belief state $b(h)$ is a probability distribution over states, conditioned on history h. Since the history satisfies the Markov property, the belief state does too.

Deep reinforcement learning In deep reinforcement learning the policy function and value function are realized as non-linear function approximators in form of a deep neural network. This is widely used since it was successfully applied in a wide range of Atari games in [5]. Therefore the deep neural network is fed with the observation as input neurons and processed by a number of hidden states. The output consists of the available actions from the action space.

The challenge is made up of the right combination of hyperparameter values. Examples of hyperparameters are the number of hidden states, learning rate, the right discount factor or algorithm specific parameters like the batch size, which will be introduced later. [8] [5]

2.2 Further Reinforcement Learning topics

2.2.1 Q-learning

Q-learning is an off-policy temporal-difference algorithm and one of the early breakthroughs in reinforcement learning. The update process is mathematically described in Equation 1. [8]

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha[R_{t+1} + \gamma \operatorname*{argmax}_a Q(S_{t+1}, a) - Q(S_t, A_t)] \qquad (1)$$

Q-learning tries to find an optimal policy, by doing iterative updates to the Q-function. Q stands for the value, given back by the action-value function, for the action A_t at a specific time step chosen while being in state S_t at a specific time step. We say Q stands for the quality of an action.

The parameter α describes the learning rate, which can be chosen in an interval of $[0, 1]$, practically near to 0, e. g. 10^{-3}. It decides how much weight newer values will have in the new calculated Q-value. The loss is calculated by the difference of highest possible Q-value, multiplied by the discount factor γ of the proceeding state and the actual Q-value. This loss is multiplied by the learning rate to make its

weight smaller. By adding the loss to the actual q-value, the q-value will iteratively converge to the optimal policy.

Thus, easy said, the agent takes an action A_t, proceeds to the next state and identifies the action with the highest Q-value. This value is multiplied with the discount factor and subtracted by the old Q-value. The result by the computation is multiplied with the learning rate and then added to old Q-value.

2.2.2 Double-Q-learning

In algorithms that use maximization in the construction of their target policies, a problem called overestimation of the action values can occur. This problem can occur, because the approximated future maximum action value is evaluated using the same action value function as in current action selection policy. If the action values are noisy, which is common during learning, these algorithms can overestimate the action values, while learning. As described in [1], Van Hasselt et al. argue that overestimations are very common. Those overestimations can lead to poorer policies. As described by Sutton and Barto in [8], we can imagine a state with a lot actions, whose true values $q(s, a)$, are all zero but whose estimated values $Q(s, a)$, are uncertain and thus distributed some above and some below zero. The maximum of the true values is zero, but the maximum of the estimated values is positive, thus a positive bias is present. This is called maximization bias. Sutton and Barto state in [8], one can overcome this problem with a variant called double learning. The idea behind this modification is, to learn two *independent* estimates $Q_1(a)$ and $Q_2(a)$, each an estimate of the true value $q(a)$. Now the first estimate $Q_1(a)$ is used to determine the maximizing action $A^* = \underset{a}{\text{argmax}} Q_1(a)$ and $Q_2(a)$ is used to estimate its value. This algorithm only updates one of both estimates, doubles the memory usage but does not increase the computation of each step. For the special case of Double-Q-learning, Equation 2 shows the update, with red colored change.

$$Q_1(S_t, A_t) \leftarrow Q_1(S_t, A_t) + \alpha[R_{t+1} + \gamma Q_2(S_{t+1}, \underset{a}{\text{argmax}}\ Q_1(S_{t+1}, a)) - Q_1(S_t, A_t)] \tag{2}$$

This can also be done for the second action value function, by switching all indices from 1 to 2 and 2 to 1 respectively. This switch is done randomly with probability 0.5. [8]

2.2.3 Experience replay

In [5] a technique called experience replay has been used by Mnih et al. and lead to impressive results, which has first being studied by Lin in [4]. The idea behind this technique is, to store the agent's experience at each time step in a replay memory, which then is accessed to do weight updates of the policy network. After an action A_t is executed in the environment, the state S_t, the environment is in, the executed action A_t, the returned reward R_{t+1} and the successor state S_{t+1} is added to the tuple $(S_t, A_t, R_{t+1}, S_{t+1})$ and stored to the replay memory. This memory is accumulated over many actions of the same episode. In theory, this replay memory can be infinite big, in practice it is limited by a buffer size N.

At each time step, there will be chosen a number of experiences (mini batch) sampled uniformly at random from the replay memory. On this mini batch multiple Q-learning updates will be performed. Thus, the state S_{t+1} will not become the new state for the next update, but the agent will learn randomly from uncoupled experience tuples. Since Q-learning is an off-policy algorithm, it does not need to be applied along connected trajectories. [8]

This method has three main advantages over learning immediately from the actions done at the moment: First using each experience for many updates of the policy improves the efficiency of learning. Second it reduces the variance of the updates over traditional Q-learning, because successive updates are not correlated with one another. Third by removing the dependence of successive experiences on the current weights, a source of instability is eliminated. [8]

2.2.4 Gradient Descent

As described by Mnih et al. in [5], the basic idea behind reinforcement learning is to estimate the action-value function by using the Bellmann equation as an iterative update, which is also described for Q-learning in Equation 1. Since in practice, this is impracticable, because the the action-value function is estimated separately for each sequence, without any generalization, it is common to use a function approximator. Since in DQN, which is used in this report, a deep neural network is used, the function approximator is non-linear. A Q-network with weights θ can be trained by minimizing a sequence of loss functions $L_i(\theta_i)$ that changes at each iteration i. This combines reinforcement learning with a supervised method.

A gradient is defined as a vector of all partial derivations of a multivariable function $grad\vec{f} = \vec{\nabla} = (\frac{\delta f}{\delta x_1}, \frac{\delta f}{\delta x_2}, ..., \frac{\delta f}{\delta x_n})$. A gradient describes the direction and the magnitude of the greatest slope.

Gradient Descent is a numerical method to find a (local) minimum by following the negative gradient. The step size to update the weights of the neural network is determined by the magnitude of the vectors. In the case of minimizing loss functions of machine learning problems, the magnitude of the gradient is multiplied by the learning rate α, a hyperparamter of the artificial neural network. If α is chosen too low, training will need a lot of time, because the loss function does converge very slow to the minimum, if it is chosen too large, the loss will not minimize or even become greater, because the algorithm can not converge to the minimum due to big steps. Thus, this algorithm is dependent by α and moreover by the initialization of the weights θ, this algorithm has to be repeated for many times (which is done by running it multiple times with a sequence of loss functions).

There are three variants of Gradient Descent, due to calculation performance: full batch Gradient Descent, which calculates the gradients over all samples each iteration, Stochastic Gradient Descent (SGD), which calculates only one gradient of one single sample each iteration and mini-batch Gradient Descent, which is a compromise between full batch Gradient Descent and SGD, and takes small batches (10 to 1000) of samples to perform Gradient Descent. In Figure 7 the methodology of filling the batch of the three methods is shown. In DQN the mini-batch GD is used.

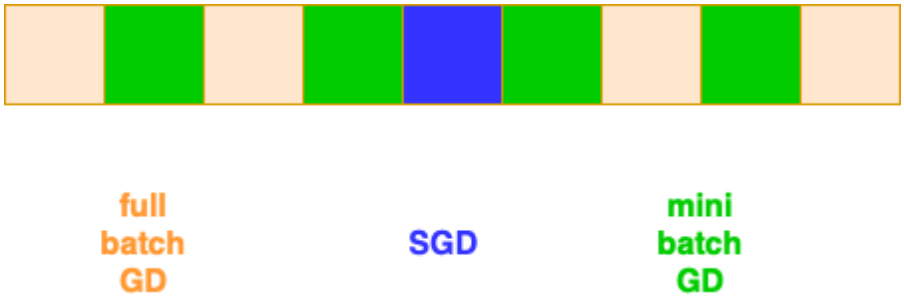


Figure 7: Illustration of different variants of Gradient Descent. Orange: A full batch GD samples all experiences; Blue: SGD samples a single experience; Green: A mini-batch GD samples a small batch of experiences.

2.3 DQN

2.3.1 Deep-Q-Network

In [5] Mnih et al. presented the first model-free deep learning model that successfully learned control policies directly from high-dimensional sensory input using reinforcement learning. In their work they trained a convolutional network with a variant of Q-learning. The Atari raw output frames are 210 x 160 pixel images with a 128 color palette. To minimize computational demand, they applied a basic preprocessing. First they converted the RGB images to their gray-scale representation and downsampled it to 110 x 84 images. Additionally they cropped an 84 x 84 region, which contains the playing area to make it more usable to GPU implementation of 2D convolutions. In their paper the function ϕ represents these steps and is further refered as preprocessing function in this report.

The final input layer consists of an 84 x 84 x 4 image. The input image is multiplied by 4, because they granted the agent the last 4 frames, to be able to "see" movements on the screen. The first hidden layer convolves 16 8 x 8 filters with stride 4 and applies a rectifier nonlinearity. The second hidden layer convolves 32 4 x 4 filters with stride 2, which is again applied a rectfier nonlinearity. The third and final hidden layer is fully-connected and consists of 256 rectifier units. The output layer is a fully-connected linear layer with a single output for each valid action. This neural network is refered as DQN.

In papers and publications the term DQN is multiple used, for the above described Deep-Q-Network and for the Deep Q-learning algorithm, which will be described in a following section. Since the DQN model is not used in this paper, but the belonging algorithm, in the following DQN will be used as synonym of the algorithm.

2.3.2 Mathematical background

As already mentioned, using the Bellmann equation as an iterative update is impracticable in practice, because the action-value function is estimated for each sequence, without generalization. Instead a deep neural network is used as a function approximator $Q(s; a; \theta) \approx Q^*(s, a)$, with θ being the weights of the neural network. A Q-network can be trained by minimizing a sequence of loss functions $L_i(\theta_i)$ that changes each iteration i. If one thinks of Q-learning, as described in

Equation 1, the update of the Q-value is done by taking an action A_t, proceeding to the next state and identifying the action with the highest value from the successor state and then this value is further processed. If this value is calculated, one can use it to calculate the difference between the actual value and the optimal future value, which is the loss. The loss then is used to update the Q-value. Thus, for one training iteration two forward passes are needed to be able to do one backward pass. The formula for calculating the loss is described in Equation 3 [5].

$$L_i(\theta_i) = \mathbb{E}_{s' \sim \rho(); s' \sim \varepsilon}[(y_i - Q(s; a; \theta_i))^2] \tag{3}$$

In the above Equation, $y_i = \mathbb{E}_{S_t, A_t \sim \rho()}[r + \gamma \underset{a'}{\text{argmax}}\ Q(s', a', \theta_{i-1})|s, a]$ is the target for iteration i and $\rho(s, a)$ is a probability distribution over sequences s and actions a that is referred as behavior distribution. Notice, that in y_i the parameter θ_{i-1} is held fixed for the training update, so that the Q-value can approximate to the target value. But by using this equation the target will move to the same direction as the Q-value, which leads to the fact, the loss cannot be minimized, though the weights will be updated for both, the target Q-value and the actual Q-value. By this, instability can occur. To solve this problem, a second network for the target values will be cloned from policy network and its weights θ will be held for a number of time steps. By this, the actual Q-value can approximate to the target Q-value and after a number of training iterations, the weights of the policy network will be copied and the target will be updated. Thus one forward pass in the policy network and one forward pass in the target network will be done instead of two in the policy network per iteration. The fact, that the target depends on the network weights is contrary to supervised learning, where the targets are fixed before learning begins.

The loss function is optimized by stochastic gradient descent. Therefore the gradient of the loss function has to be calculated and can be seen in Equation 4. The typical Q-learning can be arrived by using a single sample s_i instead of a behavior distribution over a sequence of samples and performing updates of the weights after each time step. [5]

$$\nabla_{\theta_i} L_i(\theta_i) = \mathbb{E}_{s, a \sim \rho(); s' \sim \varepsilon}[(r + \gamma \underset{a'}{\text{argmax}}\ Q(s', a'; \theta_{i-1}) - Q(s, a; \theta_i)) \nabla_{\theta_i} Q(s, a; \theta_i)] \tag{4}$$

2.3.3 The Deep Q-learning algorithm

The DQN algoithm can be seen in Algorithm 1. As it can be seen, it uses experience replay with a buffer size of N. The algorithm runs for M episodes. A episode is a path from the initial state through all visited states to the terminal state and in case of video games it represents one played match. In every episode the agent uses an ϵ-greedy strategy to choose an adequate action. Notice, that the Q-values of the actions are averaged over all actors (behavior distribution), which run in parallel. After the environment executed the action, a reward and an observation is given. The observation is preprocessed by the function ϕ. The tuple of $(\phi_t, a_t, r_t, \phi_{t+1})$ is stored to the replay memory $\mathcal{D}$ at every time step. After collecting a number of samples (which is not shown in the algorithm), the agent starts to learn. It fills uniformly random a mini batch with transitions from the replay memory. Afterwards multiple training iterations with gradient descent are performed on the mini batch. A parameter τ defines how periodically the target network will copy the weights of the policy network. For clarification the term emulator is substituted with environment and the lines dealing with the target network update are added to the algorithm.

Algorithm 1 Deep Q-learning with Experience Replay by Mnih et al. in [5]

Initialize replay memory $\mathcal{D}$ to capacity N
Initialize action-value function Q with random weights
for *episode* $= 1$ to M **do**
    Initialize sequence $s_1 = \{x_1\}$ and preprocessed sequenced $\phi = \phi(s_1)$
    for $t = 1$ to T **do**
        With probability ϵ select a random action a_t
        otherwise select $a_t = \text{argmax}_a Q^*(\phi(s_t), a, \theta)$
        Execute action and observe reward r_t and [observation x_{t+1}]
        Set $s_{t+1} = s_t, a_t, x_{t+1}$ and preprocess $\phi_{t+1} = \phi(s_{t+1})$
        Store transition $(\phi_t, a_t, r_t, \phi_{t+1})$ in $\mathcal{D}$
        Sample random minibatch of transitions $(\phi_j, a_j, r_j, \phi_{j+1})$ from $\mathcal{D}$
        $$y_i = \begin{cases} r_j & \text{for terminal } \phi_{j+1} \\ r_j + \gamma \text{argmax}_{a'} Q(\phi_{j+1}, a'; \theta) & \text{for non-terminal } \phi_{j+1} \end{cases}$$
        Perform a gradient descent step on $(y_i - Q(\phi_j, a_j, \theta))^2$
        if $t \equiv 0 \ mod \ \tau$ **then**
            Copy weights into target network $\theta_{target} \leftarrow \theta$
        else
            Let weights of target network fixed
        end if
    end for
end for

2.3.4 Double DQN

In this report DDQN is used, which is a modification to DQN. As described in a previous chapter, double learning breaks up the determination of the maximizing action and the estimation of its value to overcome the problem of overestmiations of values . In the calculation of the loss function, the term y_i is changed as follows: $y_i^{DQN} = \mathbb{E}_{S_t,A_t\sim\rho()}[r + \gamma Q(s', \underset{a'}{\text{argmax}}\ Q(s', a'; \theta_i), \theta_{i-1}^-)|s, a]$. As it can be seen, the action is chosen by the policy network with weights θ and its Q-value is evaluated by the target network with weights θ^-. [1] Algorithm 1 in line 12 changes as shown in Algorithm 2.

Algorithm 2 Double DQN (extract) by van Hasselt et al.in [1]

12: $y_i = \begin{cases} r_j & \text{for terminal } \phi_{j+1} \\ r_j + \gamma Q(\phi_j, \text{argmax}_{a'} Q(\phi_{j+1}, a'; \theta), \theta^-) & \text{for non-terminal } \phi_{j+1} \end{cases}$

2.4 Double DQN with prioritized replay

This chapter describes how scientists from DeepMind combined a bunch of techniques to further improve Q-learning. First experience replay by Lin [4] is used to break the temporal correlations of observations by mixing more and less recent experience for the updates and rare experience will be used more than just a single update. These inventions have been used in DQN by Mnih et al. in [5] and [6]. Van Hasselt et al. further proposed an algorithm called DDQN in [1] which combines double learning with DQN to address the problem of overestimation of action values. In both, DQN and DDQN the experience is sampled uniformly at random from the experience replay memory. Schaul et al. showed in [7], how experience replay can be made more efficient, by prioritizing which transactions are replayed, than if all transitions are replayed uniformly. They say in their paper “Experience replay liberates online learning agents from processing transitions in the exact order they are experienced. Prioritized replay further liberates agents from considering transitions with the same frequency that they are experienced.”

2.4.1 Prioritizing with TD-error

Schaul et al. describe, that the central component of prioritized replay is the criterion by which the importance of transitions is measured. The expected learning progress would be an ideal criterion, which is not directly accessible. A reasonable proxy is the magnitude of a transition's TD-error δ, which by default is suitable for online RL algorithms such as SARSA or Q-learning, that already compute the TD-error and update the parameters in proportion to δ. A greedy TD-error prioritization would store the TD-error alongside with every experience. The transition with the largest absolute TD-error will be replayed from the memory. New transitions do not have a TD-error, so they get the highest priority to ensure, that they will be seen at least once. [7]

2.4.2 Proportional Stochastic Prioritization

Greedy TD-error comes with three issues. First, to avoid expensive sweeps over the entire replay memory, TD-errors are only updated for the transitions that are replayed which causes, that transitions which enter the replay memory with a low TD-error, will not be replayed for a long time or even never. Moreover it is noise-sensitive, which also can be worsen by bootstrapping, where approximation errors are a source of noise. Third, because greedy prioritization focuses on a small subset of the experience, the errors will shrink slowly, because the loss will be minimized by every update, but the update steps getting smaller and smaller because the loss gets smaller. Since the TD-errors shrink slowly, the transitions will be used for updates over and over which leads to a lack of diversity and can make the system prone to over-fitting. [7]

Stochastic sampling overcomes these problems by interpolating between pure greedy prioritization and uniform random sampling. It is ensured that the probability of being sampled is monotonic in a transition's priority, while guaranteeing a non-zero probability even for the lowest priority transition. The probability of sampling transition i is defined in Equation 5,

$$P(i) = \frac{p_i^\alpha}{\sum_k p_k^\alpha} \tag{5}$$

where $p_i > 0$ is the priority of transition i. With p_i being defined as the magnitude of the TD-error $|\delta_i|$. The exponent α is constant and determines how much priori-

tization is used, with $\alpha = 0$ corresponding to the uniform case. The prioritization factor has to be determined by a hyperparamter search. The variable k is to be set equal to the sample batch size.

In this report a variant called Proportional Stochastic Prioritization is used, where $p_i = |\delta_i| + \epsilon$, where ϵ is a small positive constant that prevents the edge-case of transitions not being revisited once their error is zero.

2.4.3 Annealing the Bias

Estimations and updates need to have the same distribution. Prioritized replay changes this distribution and therefore changes the solution that the estimates will converge to. Schaul et al. correct this bias by using Importance Sampling (IS) weights as shown in Equation 6.

$$\omega_i = \left(\frac{1}{N} \cdot \frac{1}{P(i)}\right)^{\beta} \tag{6}$$

The non-uniform probabilities will be fully compensated with $\beta = 1$. This is a hyperparameter that has to be determined with a hyperparameter search. The resulting weights can be used for Q-learning updates by using $\omega_i \delta_i$ instead of δ_i. β is annealed over time, starting with β_0, ending at $\beta_{final} \leq 1$. The annealing is practically realized by a beta annealing fraction. The resulting algorithm can be seen in Algorithm 3. [7]

2.5 Ape-X DQN

Ape-X was proposed by Horgan et al. [2] in 2018. It describes a distributed architecture for deep RL. By using that architecture, agents are enabled to learn more efficient from orders of magnitudes by decoupling acting from learning. The key idea behind this is, that a learner can learn faster, if there are more experiences to learn from. This is realized by prioritizing like in DDQN and mass parallelization of the actors. This idea stands in contrast to parallelization of computation of gradients, but does not exclude each other.

In detail, there are *parallel* agents, stepping through their *own* environments, eval-

Algorithm 3 Double DQN with proportional prioritization by Schaul et al. in [7]

Input: minibatch k, step-size η, replay period K
Initialize replay memory $\mathcal{H} = \varnothing, \Delta = 0, p_1 = 0$
Observe S_0 and choose $A_0 \sim \pi_\theta(S_0)$
for $t = 1$ to T **do**
    Observe S_t, R_t, γ_t
    Store trans. $(S_{t-1}, A_{t-1}, R_t, \gamma_t, S_t)$ in $\mathcal{H}$ with max. priority $p_t = max_{i<t} p_i$
    if $t \equiv 0 \; mod \; K$ **then**
        for $j = 1$ to k **do**
            Sample transition $j \sim P(j) = p_j^\alpha / \sum_i p_i^\alpha$
            Compute importance-sampling weight $\omega_j = (N \cdot P(j))^{-\beta} / max_i \; \omega_i$
            Compute TD-error $\delta_i = R_j + \gamma_i Q_{target}(S_j, argmax_a \; Q(S_j, a)) - Q(S_{j-1}, A_{j-1})$
            Update transition priority $p_j \leftarrow |\delta_j|$
            Accumulate weight-change $\Delta \leftarrow \Delta + \omega_j \cdot \delta_j \cdot \nabla_\theta Q(S_{j-1}, A_{j-1})$
        end for
        Update weights $\theta \leftarrow +\eta \cdot \Delta, reset \Delta = 0$
        if $t \equiv 0 \; mod \; \tau$ **then**
            Copy weights into target network $\theta_{target} \leftarrow \theta$
        else
            Let weights of target network fixed
        end if
    end if
    Choose action $A_t \sim \pi_\theta(S_t)$
end for

uating their *own* policies which are implemented as neural networks. They store their experiences in a *shared* replay buffer. The other part is a *single* learner, which samples batches of experience from the replay buffer to update his *single* policy parameters. The actors periodically send remote calls to the learner, to obtain the latest network parameters. This way, the actor's policy networks will be synchronized periodically. This procedure is illustrated in Figure 8. [2]

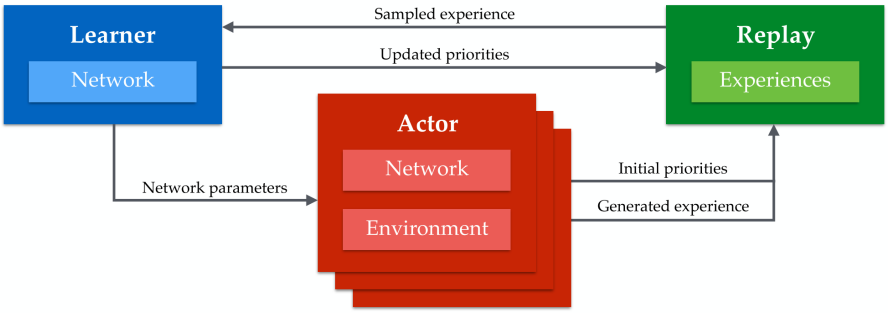


Figure 8: Ape-X architecture in a nutshell [2]

This architecture is implemented as follows: The actors run on own CPUs, the learner runs on a single GPU. The replay buffer is centralized and runs in the GPU's memory. The learner is also responsible for deleting old experiences, when the buffer gets full. In contrast to DDQN with prioritized replay, the new experiences that are added to the buffer, are not prioritized to the maximum, because it does not scale well, since waiting for the learner to update priorities would result in a myopic focus on the most recent data. Instead Horgan et al. use the fact, that in Ape-X the actors already compute priorities with their local copies of the policy. This makes the priorities more accurate but does not extra cost computation time of the learner. One advantage of sharing experiences compared to shared gradients is, that experience data outdates slower that gradients. In Ape-X the focus lies on higher throughput, which leads to a small latency in the experience, that can be neglected. Another advantage is, that it is possible to combine data from distributed actors with different exploration policies, which broadening the experience they jointly encounter.

Since Ape-X is a general framework, it can be combined with different learning algorithms. Horgan et al. described in [2] experiments with Ape-X DQN, which uses DDQN with prioritized replay and Ape-X DPG which uses DDPG. Since in this report Ape-X DQN has been used, it will be shortly described in the following.

Ape-X DQN uses DDQN with multi-step bootstrap targets as the learning algorithm and a dueling network architecture as the function approximator. This

results a slightly different loss computation, as it can be seen in Equation 7.

$$l_t(\theta) = \frac{1}{2}(G_t - q(S_t, A_t, \theta))^2 \tag{7}$$

The cumulative reward is defined by a multi-step return with double-Q bootstrap values, as it can be seen in Equation 8.

$$G_t = R_{t+1} + \gamma R_{t+2} + ... + \gamma^{n-1} R_{t+n} + \gamma^n q(S_{t+n}, \underset{a}{\text{argmax}}\ q(S_{t+n}, a, \theta), \theta^-) \tag{8}$$

In practice, the actors use ϵ-greedy policies with different values of ϵ, since a low value allows to explore deeper in the environment and higher values prevent over-fitting.

3 Experimental assembly

3.1 StarCraft II

StarCraft II is a video game of the popular genre Real Time Strategy (RTS) . RTS games are high demanding video games in which the player has to control many units (e. g. soldiers) and buildings and has to manage several concurrent processes in a strategic manner in real time (especially in contrast to round based games) while playing against one ore more opponents.

In SC2 two opponents (in 1v1) or more spawn on a map which contains resources (minerals and vespene gas) and map specific elements like ramps, plateaus, bridges or passages. To win a game a player must gather resources, construct a base, build an army and eliminate all enemy buildings. Therefore the player has to choose units from a very big technology tree and to compose its army from flying (air) and ground units, armored, melee, ranged, bio (humanoid) and mechanic units. Moreover the player has the possibility to research so called upgrades to enhance the abilities or properties of units.

The player has to build faster and in a better composition, here the principle of the rock - paper - scissors game is comparable, but in a lot more complex matter. To build specific units or research specific upgrades, there are dependencies. In especially there have to be some building present to enable a upgrade or to unlock following buildings.

Some reasons for using video games and especially SC2 for research in reinforcement learning are summarized by Vinyals et al. in [9]:

1. they provide a clear objective measure of success

2. output a lot observable data

3. are *externally* defined to be difficult problems, so developers do not tend to change the problem in order to make it easier to their algorithms

4. are designed to be run anywhere the same way and same interface, which makes it easy to share challenges and results

5. there is a pool of human players, making it possible to benchmark against highly skilled individuals

6. they can be scaled well (parallelization, speed up)

3.2 StarCraft II Learning Environment

Vinyals et al. proposed in [9] the StarCraft II Learning Environment. It is a framework for reinforcement learning on StarCraft II. The components are shown in Figure 9. As it can be seen the SC2 Binary internals are exposed by an Application Programming Interface (API) which provides the possibility to start a game, get observations, take actions and review replays. PySC2 is a Python environment that wraps the SC2 API to provide convenience functions for taking actions or getting spatial observations. It also provides a random agent and a handful rule-based agents as examples. There are also mini-games (which are used in this report) and visualisation tools to understand what the agent can sense. The observation contains non-spatial features like resources, available actions and build queues. It also contains screen features and mini map features. The agent does not receive

the RGB pixels that a human would see, but abstracted features as it is implied in the Figure. The agent also recieves a reward signal. In the SC2LE it can receive a ternary signal (-1, 0, 1) at the end of the game or can be fed while playing with the Blizzard Score, that human players only can see at the end of a game. The agent processes the input and takes an action which is then sent to the SC2 binary to be processed. [9]

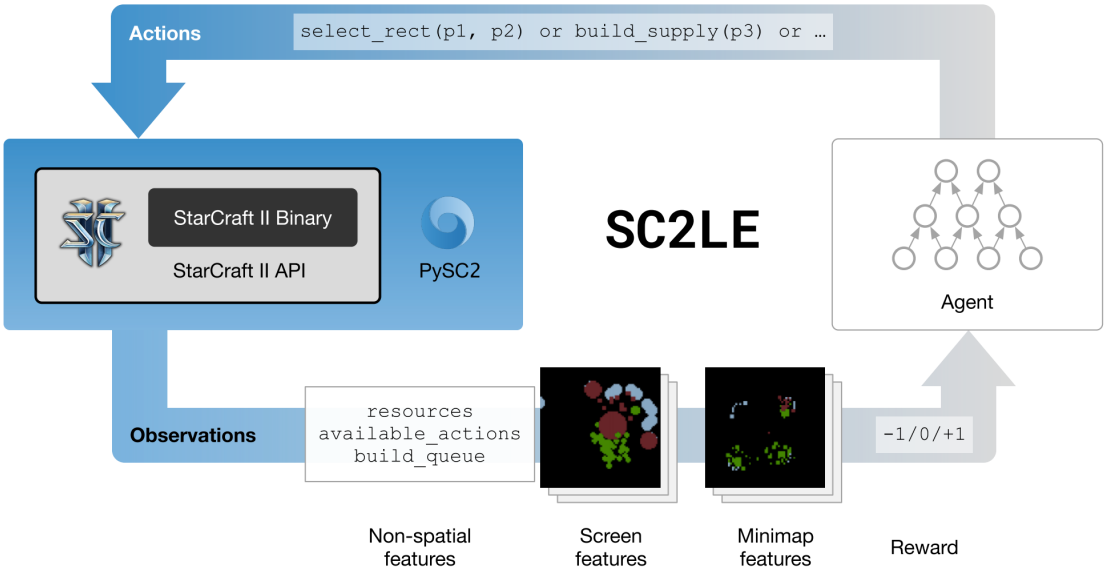


Figure 9: StarCraft II Learning Environment as proposed by Vinyals et al. in [9]

3.3 Modifications

I implemented our learning environment with Thomas Schick, who also did a research project in this field. In our environment we left out possibilities of the PySC2 framework. Instead we used a framework called PythonSC2. We used the SC2 binary and the API too. But instead of using the spatial features, we only used information given by the RAW API. The RAW API provides access to every unit (buildings, moving units) with unique identifiers and all attributes of these units. Further it provides a set of convenience functions for selecting, grouping and moving units. In SC2 many basic actions are composed of a set of actions to be done. For example building a supply depot, requires to select a worker, select the construction menue, choosing the building supply depot, find visually an adequate position to place, place. In PythonSC2 all these actions can be done by one atomic step without any visual feature processing required.

We decided to left out the visual features in order to reduce computation time. We instead only processed non-spatial features. To make that possible, we had to help

the agent with positioning of the buildings. We also managed worker selection and control.

3.4 Ray/RLLib

This reinforcement learning agent is also realized with RLlib, a framework for distributed reinforcement learning by Liang et al. which is presented in [3]. This framework offers several built-in reinforcement learning algorithms and the possibility to split computation across many Central Prozessing Units (CPUs), GPUs and computers in a cluster. It is designed to handle algorithms, that are highly irregular in the computation patterns they create and it follows the philosophy of reusing parallelizable components. It provides a logical cenralized and hierarchical control and gives the possibility to use, add and combine components and tune parameters in a centralized configuration file. This makes it on the one hand a very powerful tool for parallelized reinforcement learning but with the cost of high complexity.

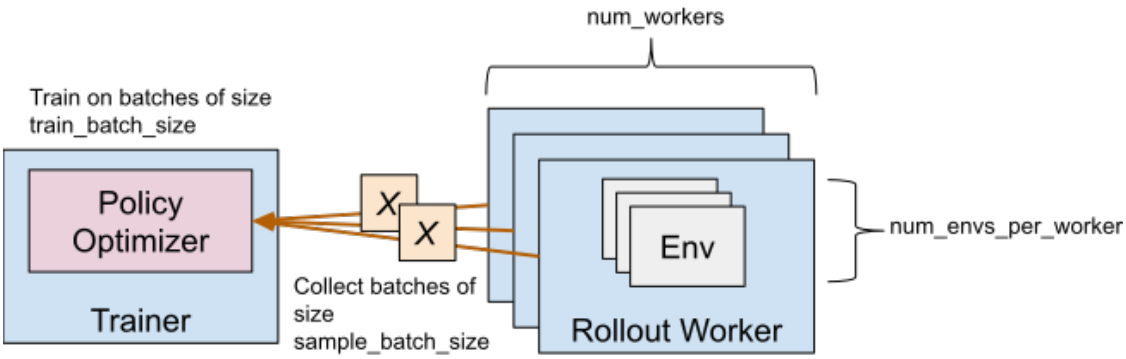


Figure 10: Illustration of a RLlib configuration[a]

[a] `https://ray.readthedocs.io/en/latest/rllib-training.html`

In this report, experiments are made with a configuration as shown in Figure 10. There is a centralized trainer, which runs on a single GPU that holds the policy network and does the training. In RLlib it is possible to assign workers. A single worker can host multiple agents with their own environments. In the experiments I used 3 workers, hosting 8 agents each. This makes a total number of 24 games of SC2 running in parallel. Every agent holds its own replay buffer with 2,000,000 experiences. The trainer fetches batches of size 4 (*sample_batch_size)* and trains on batches of size 256 (*train_batch_size)*. The workers do not own a policy network, they send their observations to the trainer's policy network to decide how to behave. The communication is realized over HTTP by a REST API. Note that

this architecture is different from Ape-X. Here, every Worker has its own replay buffer. In Ape-X only one centralized replay buffer exists.

3.5 Used hardware and software

In my experiments I used a single machine of TH Cologne with a CPU Intel(R) Xeon(R) CPU E5-2680 v4 @ 2.40GHz with 56 cores, 256 GB Random Access Memory (RAM) and a Ubuntu Linux 16.04.6 LTS. Table 1 shows the used software.

software	**version**
anaconda	4.6.11
python	3.6.8
python-sc2	0.11.1
ray/rllib	0.7.0.dev2
StarCraft II binary	B70326 (SC2.2018Season4)
s2clientprotocol	4.9.1.74456.0
tensorflow	1.12.0

Table 1: Used software components in the project

3.6 Reinforcement Learning Scenario

In this report I used a smaller problem than the entire game. The SC2LE contains special scenarios called mini-games to provide environments with clear learning tasks and reasonable rewarding. Figure 11 shows the map with the units and buildings, that are available in the scenario.

3.6.1 Scenario

In this scenario the agent shall learn to train marines. Marines can be trained in a military building called barracks. In order to be able to train marines the agent must perform preliminary steps. First it must gather minerals, which are the basic resources in SC2. If enough resources are gathered, the agent is able to build supply depots. Supply depots have two purposes. First, they raise the supply cap, a population limit for moving units. Every moving unit *costs* a specific amount of supply. Second, they unlock the availability to build a barracks. Buildings, units and researches are locked and need to be unlocked by other buildings or preliminary

Figure 11: The Reinforcement Learning Scenario *BuildMarines*

researches. After that, one or more barracks can be built. If a barracks is present, the agent is able to produce marines as long as the supply cap is not reached and enough resources are stocked.

In the following the agent has to make sure, that 1. enough resources are present to produce more marines, 2. that enough supply depots are built, to guarantee that enough supply is left for training a marine, 3. ideally non-stop marines are being trained.

3.6.2 Rewarding

There is an ideal number of SCVs (the unit that can construct buildings and gather resources), supply depots and barracks to achieve a maximum of reward. The reward function gives back a reward of one for every trained marine. The cumulative reward is therefore the sum of trained marines. The difficulties for the agent are the delayed reward (every training takes some time, only after finishing the training, the agent earns the reward) and the tech-lock, which forbids to build a barracks without a supply depot. A simulation takes 15 minutes before it terminates and restarts. One further point is, that it is possible to queue orders. This could hamper the learning additionally, because the agent can receive delayed rewards, that seem totally uncorrelated to the actions, it performed recently.

3.6.3 Observations, Actions and neural network architecture

The agent gets non-spatial features as observation. It gets information about the amount of gathered minerals, the supply that is left until reaching the actual supply cap, the already total used supply (for moving units), the actual frame (a real time second is considered to be 16 frames). Beside this, the agent can also sense the amount of built supply depots, barracks, SCVs and marines. The available actions, that a SCV can perform are *build_supply* and *build_barracks*. The command center can perform *build_worker* and the barracks can perform *build_marine*. The agent needs to learn, that it needs special buildings to perform these actions but does not need to learn to select those. There is also a *do_nothing* action, to enable the agent to wait. The observations and available actions are summarized in Tables 2 and 3.

feature	**description**
minerals	basic resource
supply_left	Left supply until a new supply depot has to be built
supply_used	Total amount of used supply for moving units
frame	actual frame number (16 frames are supposed to be one second)
supply_depots	number of built supply depots
barracks	number of built barracks
SCVs	number of trained SCVs
marines	number of trained marines

Table 2: Overview of feature vector

action	**description**
build_worker	Train a SCV in the command center
build_marine	Train a marine on a barracks
build_supply	Construct a supply depot
build_barracks	Construct a barracks
do_nothing	Do nothing in this time step

Table 3: Overview of available actions

The neural network is composed of four fully-connected hidden layers with a input layer in front and a output layer at the end. The hidden layers have 256 neurons each, the input layer has eight neurons (number of features) and the output layer has a one-hot encoded neuron. Depending on the value (from 0 to 4), we select the action to be performed.

3.6.4 Simplifications

As described by Vinyals et al. in [9], this is a challenging problem for reinforcement learning algorithms. To make the problem easier and to save computation time, I decided to automate the control of gathering SCVs. If a SCV is trained or finished a construction job, it will be automatically assigned and sent to a mineral field. Moreover the agent does not need to process spatial inputs. If the agent constructs a building, the selection of a worker and the positioning of the building is automated.

4 Experiments

4.1 Building a non-AI bot

In order to get familiar with PythonSC2 and to test the capabilities of the framework, we implemented in a group a bot to play against the built-in artificial player in our research team in the full game. We divided the bot into four sub-agents. The first had to manage the control of the workers, the second had to realize a build order and produce units, another sub-agent that did the scouting to gather information about the opponent and the fourth sub-agent had to control the army movement. We were able to defeat the easy and medium artificial opponent. After that we split up to do our own special research projects. The agent for controlling workers and the agent for constructing buildings has been adopted for the experiments of this report.

4.2 Parameter Search for DQN

After splitting into teams, I performed an academic grid search on common parameters for reinforcement learning for the DQN algorithm. The results are presented in the following. For every parameter value a total amount of 10 measurements over at least 3 hours have been taken. With some reference measurements, I found out, that 3 hours of computation are enough to see the maximum of most graphs. The ten measurements represent the mean values of 24 parallel environments. To make the results comparable, a mean of these 10 measurements have been calculated. The particular measurements can be viewed in detail in Appendix A.

4.2.1 Training batch size

DQN uses a replay buffer to store experience, during playing the game. From this replay buffer a batch is taken to perform trainings. In order find the ideal training batch size, I assumed the discount factor γ and the learning rate to be constant and varied the batch size in the interval $\{64, 256, 1024, 2048\}$. Table 4 shows the parameter values and Figure 12 shows the mean values for the training batch size. As it can be seen from Figure 12 a training batch size of 64 has a slightly higher

parameter	**value**
discount factor	0.99
learning rate	0.0001
training batch size	$\{64, 256, 1024, 2048\}$

Table 4: Parameter set for training batch size grid search

slope than a size of 256 but it only reaches a maximum of 63 whereas the graph with a size of 256 reaches a maximum of 65 and has a clearly smaller negative slope afterwards. The other graphs with sizes of 1024 and 2048 have clearly smaller slopes and lower maxima. Since these results favored a size of 256, I chose this size for further measurements.

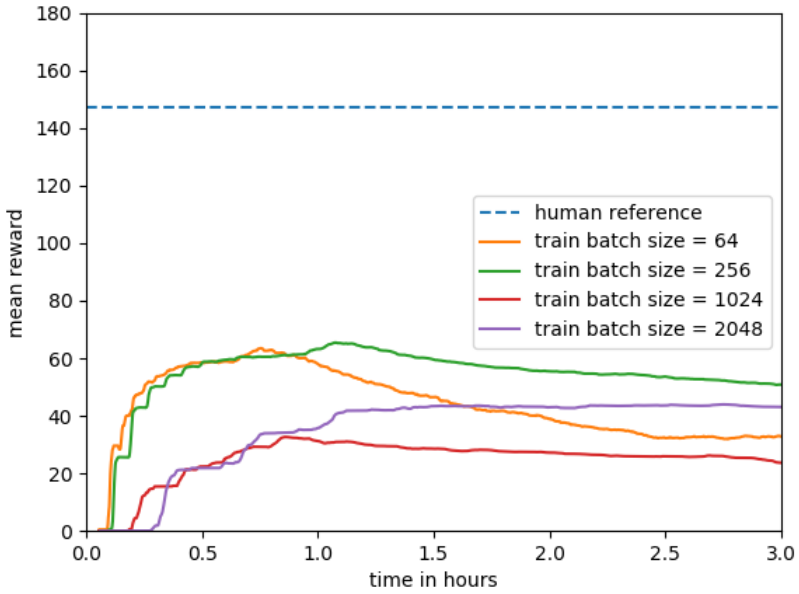


Figure 12: Mean graphs of parameter search for traininig batch size

4.2.2 Discount factor

The second grid search treats the discount factor. In theory a discount factor near to 0 discounts future rewards, so that only the immediate reward has influence on the expected reward. A discount factor near to 1 does not discount future rewards and the agent takes into account, that future rewards are important. Thus, in this experiment I supposed, that a high discount factor should lead to better results, due to the fact, that better results can be achieved, if in the beginning a hand full workers are trained, to raise the mining rate of resources. More resources lead to a potential higher training rate of marines, although in the beginning a lower reward is received. The results of the grid search can be seen in Figure 13. In this

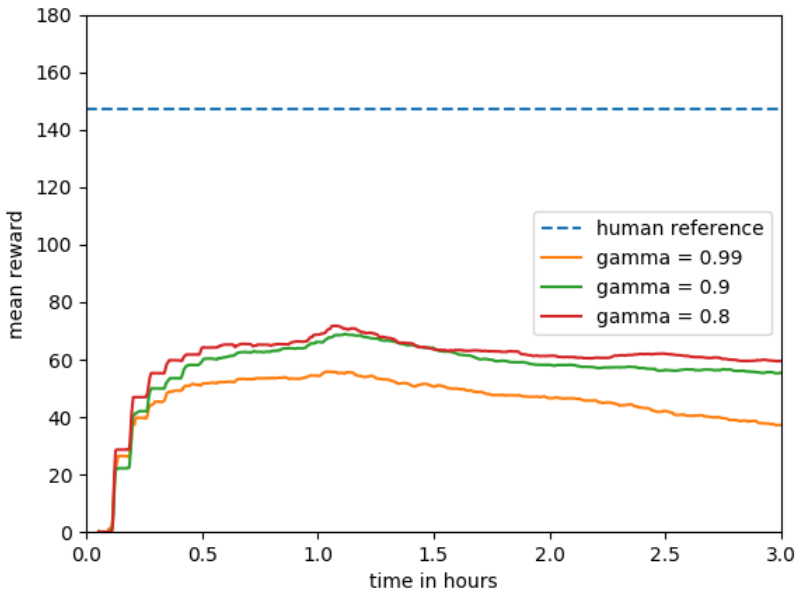


Figure 13: Mean graphs of parameter search for discount factor

grid search the result from the first grid search has been taken into account. Thus a training batch size of 256 has been chosen, the other parameters remained the same. The parameters can be seen in Table 5.

parameter	**value**
discount factor	$\{0.8, 0.9, 0.99\}$
learning rate	0.0001
training batch size	256

Table 5: Parameter set for discount factor grid search

Surprisingly, a discount factor of 0.8 gives the best results. The discount factor of 0.99 only gives a maximum of 55 and falls faster than the other graphs. With a

maximum of 72, γ with 0.8 has a littler greater cumulative reward (4 points) than the factor of 0.9. I assume, that the agent has problems to link the reward for trained marines with the action of training marines, since it takes 18 seconds until the reward is given. The review of a replay shows, that the agent has problems to connect easy correlations between the supply cap and the number of supply depots, because it constructs those as soon as it has enough resources. The barracks are strongly delayed, because they cost more resources, which are often spent before. Since the agent cannot connect these correlations, it seems, that a discount factor of 0.8 is adequate, because the agent will preferably learn, that training marines will somehow give more reward than training SCVs or construct supply depots.

4.2.3 Learning rate

The third tuned hyperparameter in this report is the learning rate. Since it is common known, that a high learning rate causes faster learning but produces higher variance and low learning rate causes slow progress, but finer optimization, a grid search for the optimal learning rate has been performed. Therefore the parameter set of Table 6 has been used. From grid search 1 and 2 a training batch size of 256 and a discount factor of 0.8 has been taken. The learning rate has been varied in the interval $\{0.0001, 0.001, 0.01\}$.

parameter	**value**
discount factor	0.8
learning rate	$\{0.0001, 0.001, 0.01\}$
training batch size	256

Table 6: Parameter set for learning rate grid search

The results are shown in Figure 14. As it can be seen all evaluated learning rates perform equally well until they reach one hour of training. Afterwards they split up and a learning rate of 0.01 gives the best mean results with on average 15 reward more than the second best graph. Thus, a learning rate of 0.01 is chosen fur the upcoming experiments.

4.3 Performance comparison

In the following section, a performance comparison is executed. Therefore the tuned version of DQN will be compared to the untuned versions of DDQN with experience

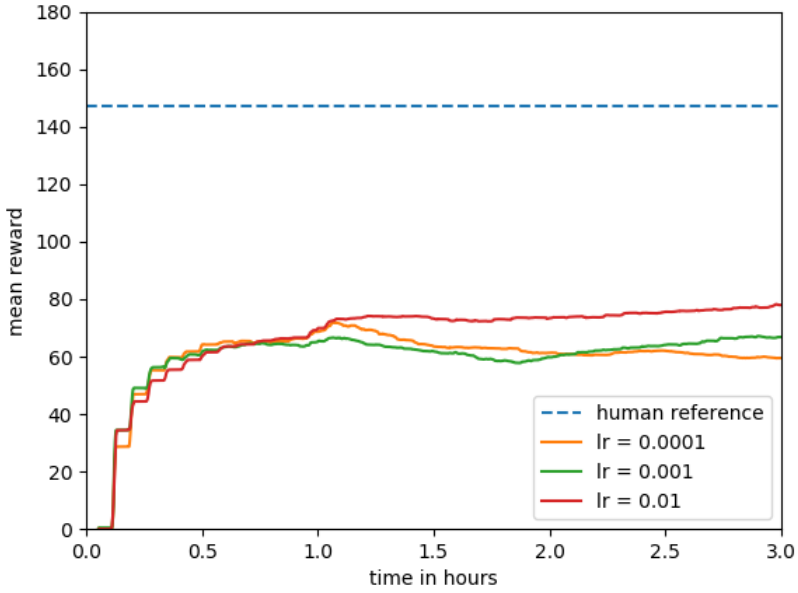


Figure 14: Mean graphs of parameter search for learning rate

prioritization and Ape-X DQN. For DDQN and Ape-X DQN the parameter set of the tuned DQN version will be used. Further parameters, that are introduced with DDQN or Ape-X DQN will not be tuned, but set to "standard" values. In especially values will be used, that are known to perform well in the Atari emulator related experiments of Schaul et. al in [7] (DDQN) and Horgan et al. in [2] (Ape-X DQN).

4.3.1 DQN

Figure 15 shows the mean graph of all measurements done with the tuned hyper-parameters, as found in the previous sections, and the best performed graph in all experiments. As it can be seen, the agents perform with an average reward of 75-80. This is explainable with the fact, that the 24 clients, that run in parallel have a equally distributed ϵ between 0.4 and 0. Thus some clients have a relative high probability to do a random action instead of following the policy. The maximum graph instead ranges between a reward of 110 and 122. These results are comparably well, if it is compared with a good human player, that reached a cumulative reward of 147.

In fact, by reviewing a replay, that uses that policy, it seems as the agent has problems to connect the amount of supply depots and the available supply, since

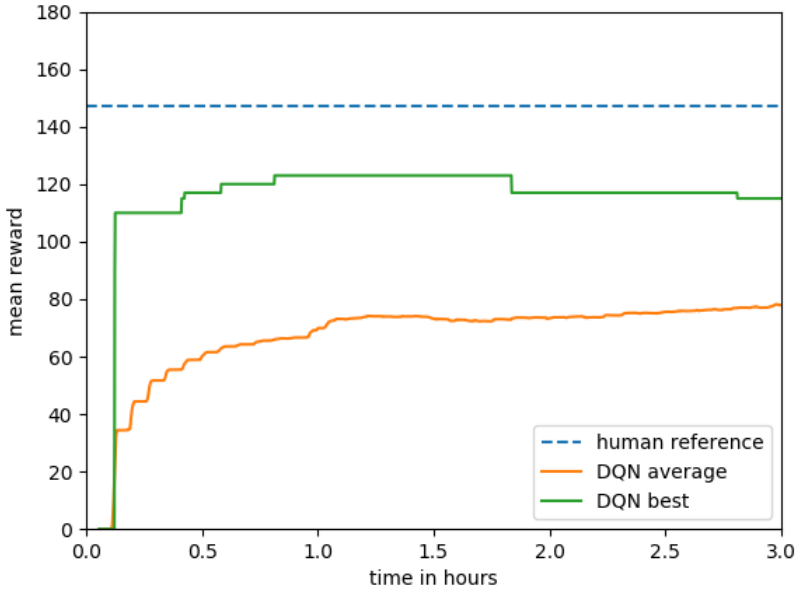


Figure 15: Average graph and best graph of tuned DQN

the agent constructs too many supply depots while playing. Moreover it seems, as the agent can somehow connect the action "build_marine" and the delayed reward of trained marines, but not the correlation between the necessity of first constructing a supply depot to unlock barracks. This can be seen in the replay, when the agent builds three or four supply depots in a row in the beginning of the game and using all available resources. It does not save resources to construct a barracks. Nevertheless, after constructing some barracks it produces perpetual marines.

4.3.2 DDQN

After tuning DQN, the hyperparameters have been adopted to the DDQN setting. DDQN-specific hyperparameters have been adopted from [7], as it can be seen in Table 7.

The results are shown in Figure 16. As it can be seen, the average cumulative reward is between 10 and 18. This clearly shows, that is not sufficient, to tune for the DQN setting and adopt it to the DDQN setting. Since DDQN uses prioritization, dueling and double learning it is clearly supposed to achieve at least equally good results, if not better. Nevertheless if we look at the maximum graph, it can be seen, that the maximum cumulative reward ranges from 95 to 103. I assume, that

parameter	value
discount factor	0.8
learning rate	0.01
training batch size	256
prioritization α	0.6
prioritization β_0	0.4
prioritization β_{final}	0.4
prioritization ϵ	1^{-6}

Table 7: Parameter set for DDQN

the algorithm can still achieve good results, because of the fact, it uses importance sampling. With adequate hyperparamter tuning, an improvement is presumable.

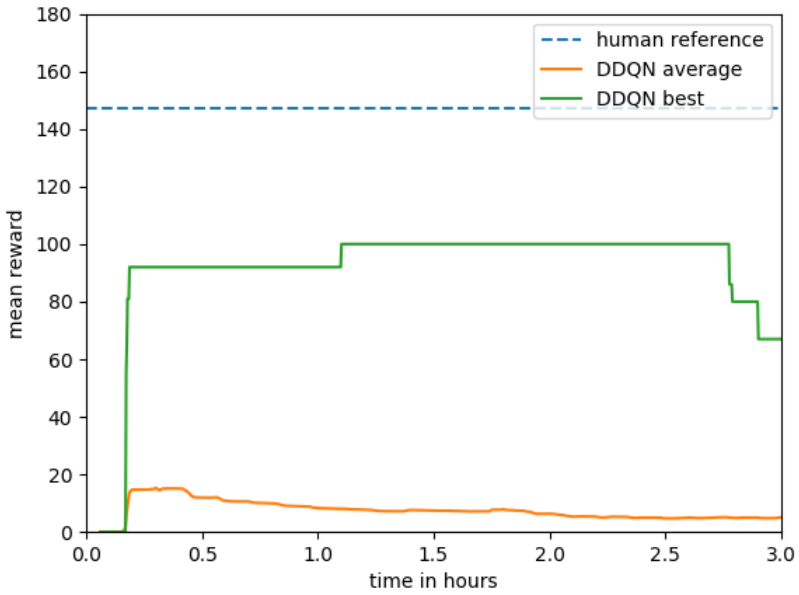


Figure 16: Average graph and best graph of untuned DDQN

4.3.3 Ape-X DQN

Equally to DDQN, in the measurements for Ape-X DQN, the hyperparameters of the DQN setting has been adopted. As explained earlier, all agents feed their experiences into one shared replay memory, controlled by a central learner. I assumed, that Ape-X DQN will perform better than DDQN, because the prioritized experiences should be used more efficient than in single replay memories, distributed to the agents. Figure 17 shows the resulting graphs. As it can be seen, the average graph ranges between 10 and 18. This is very similar to the average graph of DDQN. Thus, my assumption is disproved. It seems as, the tuned DQN hyperparameters

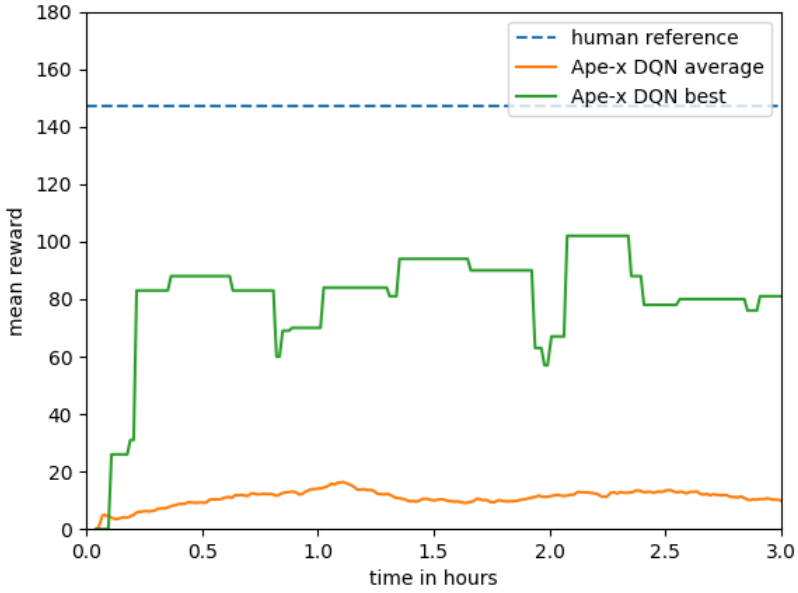


Figure 17: Average graph and best graph of untuned Ape-X DQN

are not sufficient for the distributed setting of Ape-X. Here a hyperarameter search should produce much better results. The maximum graph ranges between a cumulative reward of 75 to 105. I assume the high variance comes from the high learning rate of 0.01. With a smaller or decreasing learning rate better results should be possible. Moreover it has to be mentioned, that the measurements have been taken with only 24 parallel agents, as in the prior settings. The strength of Ape-X is the high scalability, which means the results should getting better with a higher number of parallel agents (e. g. 256 or 512 agents in parallel). Since for this report there was not enough calculation power available, this border case could not be investigated.

4.3.4 Comparison

In the following the maximum graphs of vanilla DQN, DDQN and Ape-X DQN will be shown and discussed. The graphs can be seen in Figure 18. The tuned variant of DQN gives the highest cumulated rewards. This shows on the one hand, that hyperparameter tuning is very important to get good results. On the other hand it clearly shows, that the optimized variant DDQN and the distributed setting Ape-X cannot generally be fed with the tuned hyperparameters of DQN to get equal or better results. Thus, a hyperparameter search for the successor of DQN has to be performed. The third conclusion to infer from this comparison is,

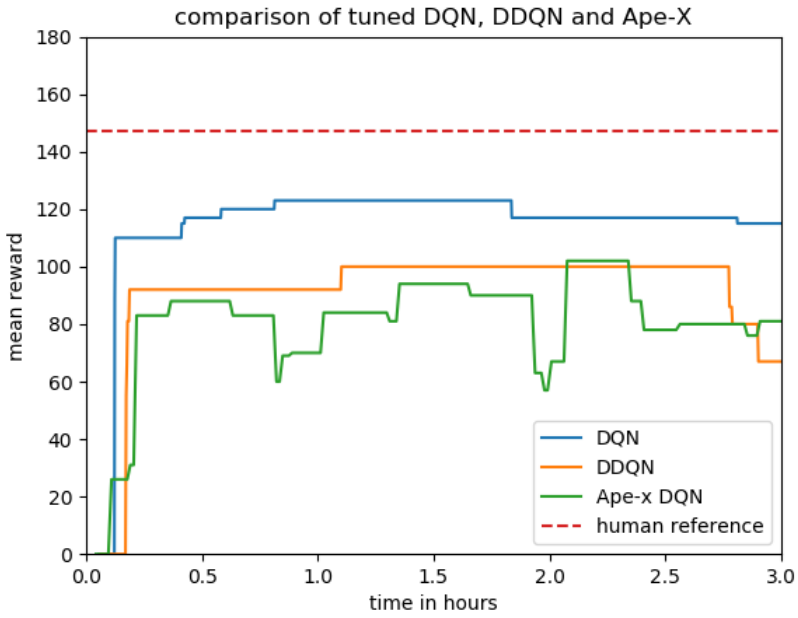


Figure 18: Comparison of the best results of DQN, DDQN and Ape-X DQN

that these algorithms can perform sufficiently well, with non-ideal hyperparameters. All algorithms could not reach the experienced player, that is plotted in all Figures. This can have several reasons: The delayed reward could make it impossible for the agent to connect the action with the reward. Additionally orders can be queued which brings extra difficulties to learn this connection. Here Long short-term memorys (LSTMs) could have helped, but they were not available in RLlib in combination with DQN-related algorithms. Those could be subject of upcoming projects in this field.

5 Conclusion

In the following the results will be summarized and a personal conclusion is drawn.

5.1 Results

In this report, a RL environment to play SC2 with DQN has been built. It has been shown, that DQN can give reasonable results, if trained on a mini game called *build_marines*. To achieve that, a hyperparameter search has been performed and an agent has been sucessfully trained with non-spatial features. It has been also

shown that the successor algorithm DDQN and the distributed setting with Ape-X DQN can be applied to this problem and that they can perform sufficient well with the tuned hyperparamters of DQN. Since this is an improved algorithm, it should perform better with a own hyperparamter search.

Since the learning problem has been artificially made easier, because the spatial observations has been cut away and the agent also did not have to learn the more complicated action steps, that a human would have to perform, this work cannot be compared with the work of other studies, e. g. like Mnih et al. in [5] or Vinyals et. al in [9]. To make them comparable, spatial features have to be implemented, which could be part of further projects.

Moreover the agents seem to have problems to connect simple dependency chains, like the necessity of constructing a special building first, to unlock another or sparing resources to be able to construct a building. These problems would also occur in a setting with spatial features. Here LSTMs could help.

5.2 Lessons learned

Since this was my first research project in the field of machine learning, there were a lot of pitfalls.

When we started our research, we implemented in a group of four a scripted bot to play the whole game against computer opponents. We did this in order to get to know the PythonSC2 framework and its capabilities. We invested to much time into this, so that we somehow wasted time.

After we split up, Thomas Schick and I implemented our learning environment. We had several problems by implementing our environment. After setting everything up, we tried to run the training, but we did not get it working for a long time. Reason for that was, that we did not know enough about the algorithms we used. One thing I have learning from this is, that it is very important to invest some time to understand the algorithms that I use, before trying to use it.

Another point, related to that is, that we used RLlib, which on the one hand is a very powerful tool to use but at the other hand very complicated to use and not enough documented for a quick start, if not everything of the used algorithm has

been understood correctly. I recommend to implement an algorithm yourself before using a framework.

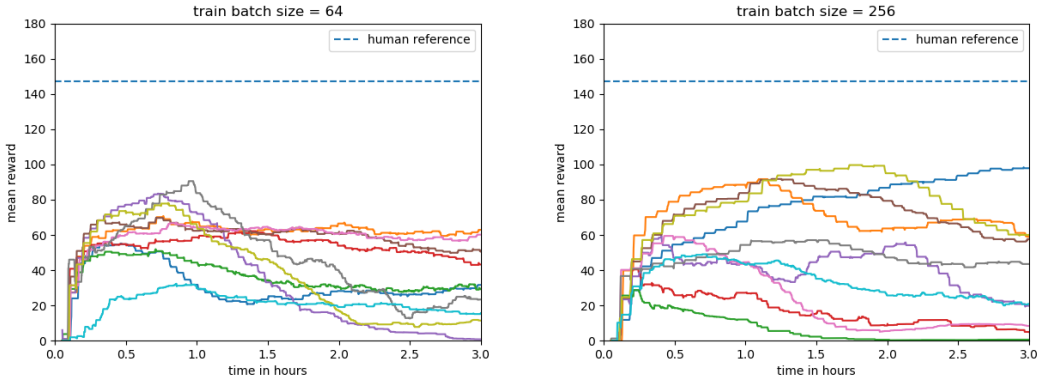
Further it needs some time to find the topic of the concrete research project. We started with the idea of using an algorithm called Asynchronous Advantage Actor Critic (A3C) and an architecture called auto-encoder to find out, if it possible to speed up training with encoded non-spatial features. We ended in using DQN without using auto-encoders. This took some time, that we did not calculate beforehand.

There are a lot of technical problems that occur while doing a project like this, that consume a lot of time. It is advisable to plan extra time for troubleshooting, debugging or creating graphs.

A All Measurements

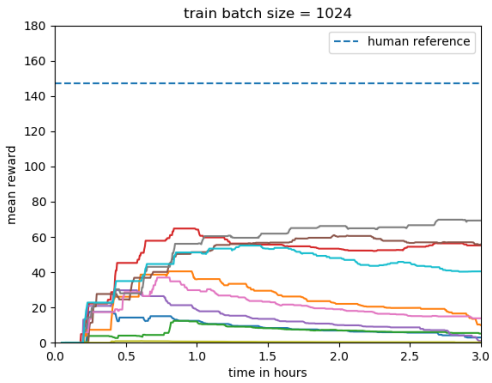
DQN Grid search

For every hyperparameter grid search a total amount of ten measurements per value have been performed. The mean graphs can be seen in Figures 19 to 21.

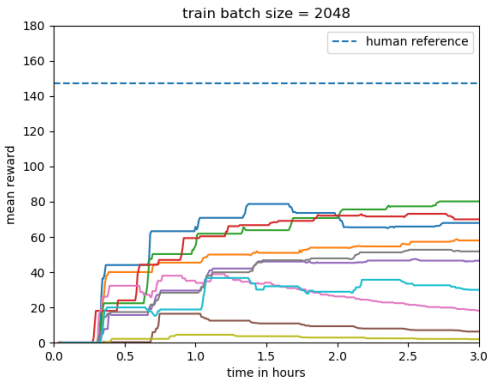


(a) training batch size = 64

(b) training batch size = 256

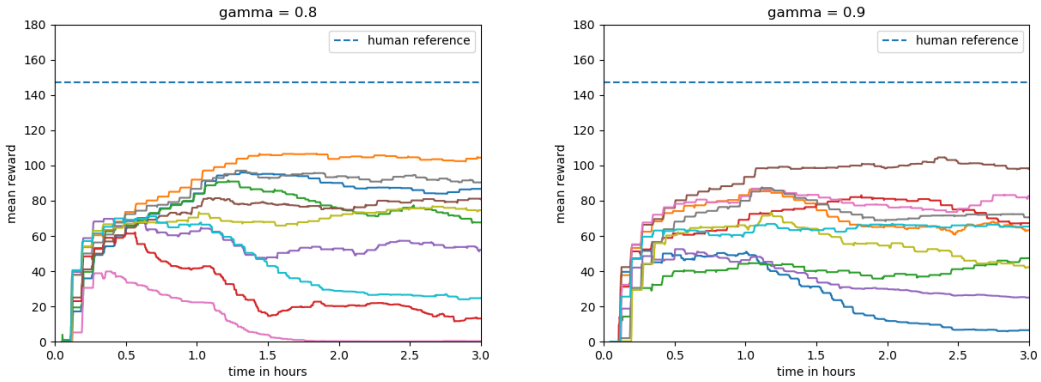


(c) training batch size = 1024

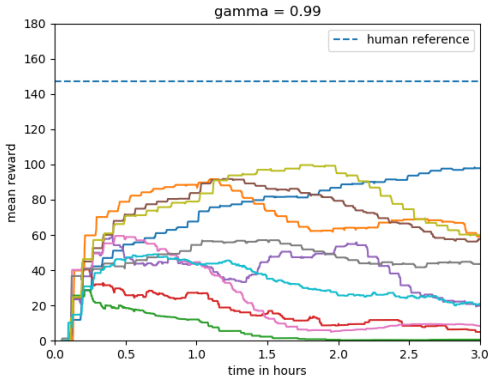


(d) training batch size = 2048

Figure 19: Collocation of experiments performed for finding the value of the training batch size: 19a to 19d show the range of the evaluated values. 19b turned out to have the highest mean reward, as described in experiments section.

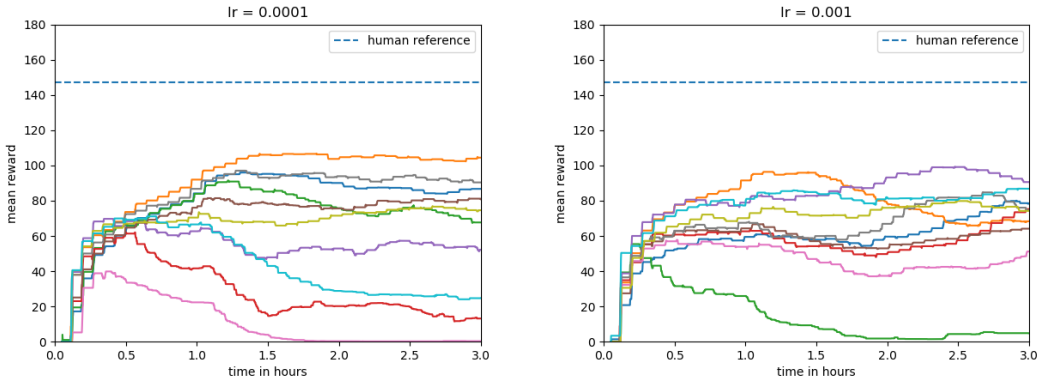


(a) $\gamma = 0.8$ (b) $\gamma = 0.9$

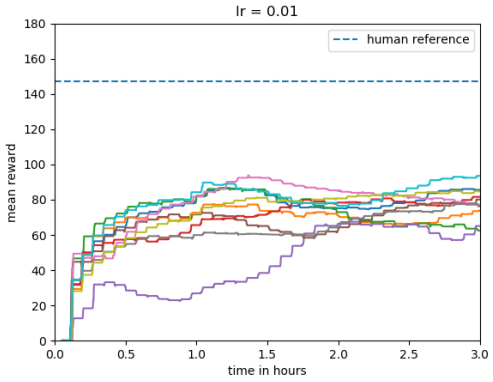


(c) $\gamma = 0.99$

Figure 20: Collocation of experiments performed for finding the value of the discount factor: 20a to 20c show the range of the evaluated values. 20a turned out to have the highest mean reward, as described in experiments section.



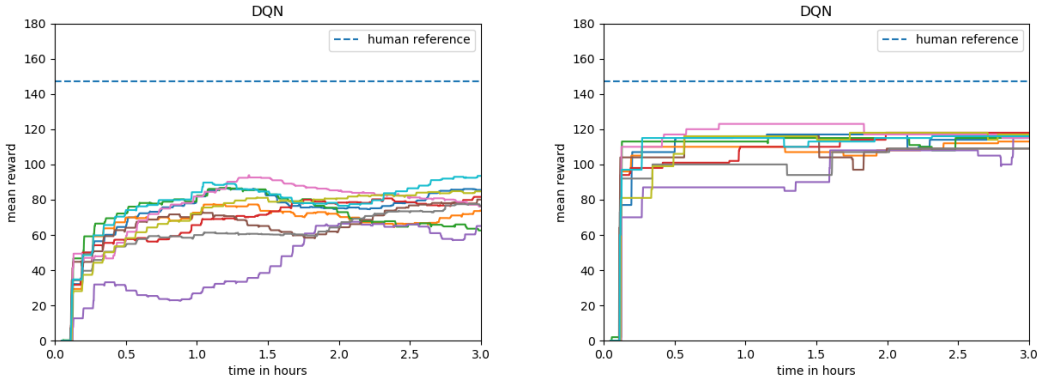
(a) $\alpha = 0.0001$ (b) $\alpha = 0.001$



(c) $\alpha = 0.01$

Figure 21: Collocation of experiments performed for finding the value of the learning rate: 21a to 21c show the range of the evaluated values. 21c turned out to have the highest mean reward, as described in experiments section.

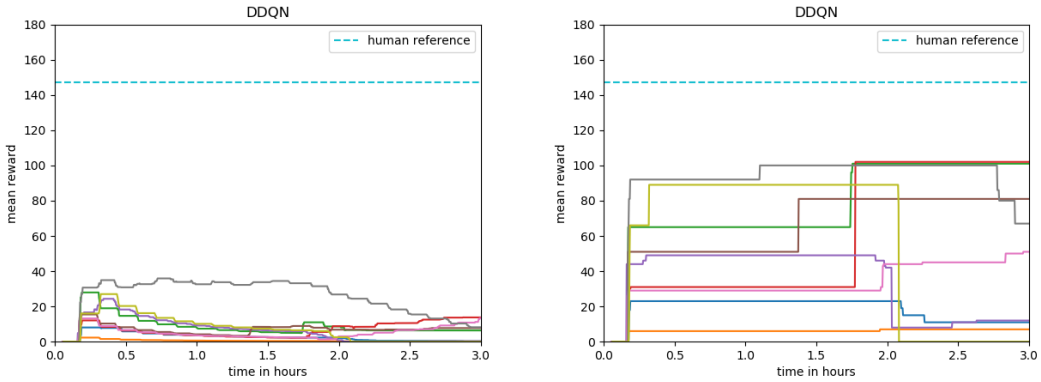
DQN mean and max reward graphs



(a) DQN mean graphs (b) DQN maximum graphs

Figure 22: Collocation of experiments performed for examining DQN performance with tuned hyperparameters: 22a shows the mean graphs of the experiments, 22b shows the maximum reward graphs.

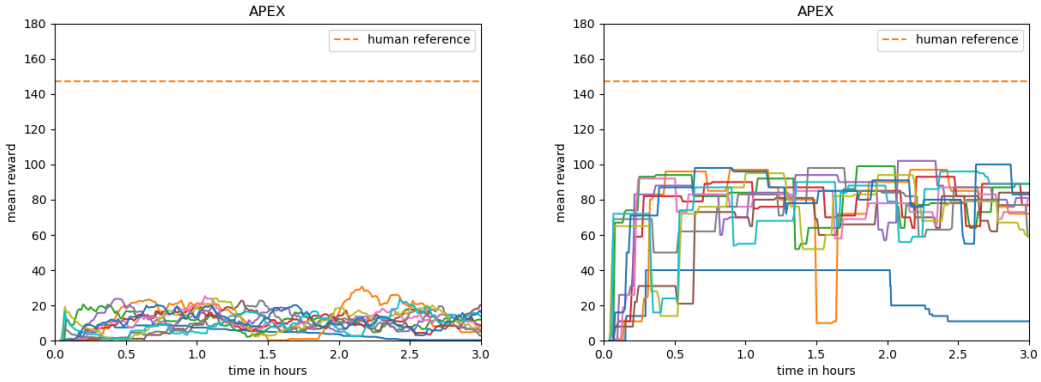
DDQN mean and max reward graphs



(a) DDQN mean graphs (b) DDQN maximum graphs

Figure 23: Collocation of experiments performed for examining DDQN performance with tuned hyperparameters for vanilla DQN: 23a shows the mean graphs of the experiments, 23b shows the maximum reward graphs.

Ape-x mean and max reward graphs



(a) Ape-X DQN mean graphs

(b) Ape-X DQN maximum graphs

Figure 24: Collocation of experiments performed for examining Ape-X performance with tuned hyperparameters for vanilla DQN: 24a shows the mean graphs of the experiments, 24b shows the maximum reward graphs.

Literature

[1] Hasselt, H. van ; Guez, A. ; Silver, D. : Deep Reinforcement Learning with Double Q-learning. (2015). http://arxiv.org/abs/1509.06461

[2] Horgan, D. ; Quan, J. ; Budden, D. ; Barth-Maron, G. ; Hessel, M. ; Hasselt, H. van ; Silver, D. : Distributed Prioritized Experience Replay. (2018), 1–19. http://arxiv.org/abs/1803.00933

[3] Liang, E. ; Liaw, R. ; Moritz, P. ; Nishihara, R. ; Fox, R. ; Goldberg, K. ; Gonzalez, J. E. ; Jordan, M. I. ; Stoica, I. : RLlib : Abstractions for Distributed Reinforcement Learning. (2014)

[4] Lin, L.-J. : Reinforcement learning for robots using neural networks. In: *ProQuest Dissertations and Theses* (1993), 160. https://search.proquest.com/docview/303995826?accountid=12063{%}0Ahttp://fg2fy8yh7d.search.serialssolutions.com/directLink?{&}atitle=Reinforcement+learning+for+robots+using+neural+networks{&}author=Lin{%}2C+Long-Ji{&}issn={&}title=Reinforcement+learning+for+robots+us

[5] Mnih, V. ; Kavukcuoglu, K. ; Silver, D. ; Graves, A. ; Antonoglou, I. ; Wierstra, D. ; Riedmiller, M. : Playing Atari with Deep Reinforcement Learning. (2013), 1–9. http://dx.doi.org/10.1038/nature14236. – DOI 10.1038/nature14236. – ISBN 1476–4687 (Electronic) 0028–0836 (Linking)

[6] Mnih, V. ; Kavukcuoglu, K. ; Silver, D. ; Rusu, A. A. ; Veness, J. ; Bellemare, M. G. ; Graves, A. ; Riedmiller, M. ; Fidjeland, A. K. ; Ostrovski, G. ; Petersen, S. ; Beattie, C. ; Sadik, A. ; Antonoglou, I. ; King, H. ; Kumaran, D. ; Wierstra, D. ; Legg, S. ; Hassabis, D. : Human-level control through deep reinforcement learning. In: *Nature* 518 (2015), Nr. 7540, 529–33. http://dx.doi.org/10.1038/nature14236. – DOI 10.1038/nature14236. – ISSN 1476–4687

[7] Schaul, T. ; Quan, J. ; Antonoglou, I. ; Silver, D. : Prioritized Experience Replay. (2015), 1–21. http://arxiv.org/abs/1511.05952

[8] Sutton, R. S. ; Barto, A. G.: *Reinforcement Learning: An Introduction, Second edition in progress*. 2017. – 360 S. http://dx.doi.org/10.1016/S1364-6613(99)01331-5. http://dx.doi.org/10.1016/S1364-6613(99)01331-5. ISSN 13646613

[9] Vinyals, O. ; Ewalds, T. ; Bartunov, S. ; Georgiev, P. ; Vezhnevets, A. S. ; Yeo, M. ; Makhzani, A. ; Küttler, H. ; Agapiou, J. ; Schrittwieser, J. ; Quan, J. ; Gaffney, S. ; Petersen, S. ; Simonyan, K. ; Schaul, T. ; Hasselt, H. van ; Silver, D. ; Lillicrap, T. ; Calderone, K. ; Keet, P. ; Brunasso, A. ; Lawrence, D. ; Ekermo, A. ; Repp, J. ; Tsing, R. : StarCraft II: A New Challenge for Reinforcement Learning. (2017). http://dx.doi.org/https://deepmind.com/documents/110/sc2le.pdf. – DOI https://deepmind.com/documents/110/sc2le.pdf. – ISBN 0962–8436, 0962–8436