INSTITUT FÜR INFORMATIK
Machine Learning
Universitätsstr. 1 D–40225 Düsseldorf

HEINRICH HEINE
UNIVERSITÄT DÜSSELDORF

Actor-Critic Reinforcement Learning With Experience Replay

Julian Robert Ullrich

Bachelorarbeit

Beginn der Arbeit:	25. Juli 2018
Abgabe der Arbeit:	25. Oktober 2018
Gutachter:	Univ.-Prof. Dr. S. Harmeling
	Univ.-Prof. Dr. M. Leuschel
Betreuer:	Julius Ramakers

Erklärung

Hiermit versichere ich, dass ich diese Bachelorarbeit selbstständig verfasst habe. Ich habe dazu keine anderen als die angegebenen Quellen und Hilfsmittel verwendet.

Düsseldorf, den 25. Oktober 2018

Julian Robert Ullrich

Abstract

Deep reinforcement learning and policy gradient methods majorly contributed to the most recent advances in the field of Artificial Intelligence. These methods enable machines to surpass human performance for Atari console games (Mnih et al., 2013), board games like Chess, Shogi (Silver et al., 2017a) or Go (Silver et al., 2017b) and most recently even complex team-based computer games (OpenAI, 2018).

As environments get more complex, the cost of simulating the environment increases and often outweighs the isolated computational cost of training the agent, making sample efficient methods necessary.

This thesis will take a look at off-policy methods and learning from previously sampled data, with the main focus being the implementation and evaluation of the "Actor-Critic with Experience Replay" (ACER) algorithm proposed by Wang et al. (2016) on the Atari 2600 console games.

Contents

1 Introduction

Sutton and Barto (2018) describe the reinforcement learning task as "learning what to do". Acting optimal within an unknown environment can be very difficult. The field within machine learning addressing this problem is called reinforcement learning.

The reinforcement problem consists of an *agent* taking *actions* within some sort of *environment*. By interacting with the *environment* the *agent* tries to find the *actions* which will yield the most *reward* in the future.

The goal of reinforcement learning is to create fast and reliable learning algorithms for the *agent* to take the optimal *actions* within the *environment*. This means, we want to achieve the maximum possible *reward* within an episode or over a period of time if an environment is continuous.

Environments pose a lot of different tasks of varying difficulty. Easy environments, like the 'Cartpole'-environment, require the agent to simply balance a pole based on 4 input values with only 2 possible actions, whereas more complex and demanding tasks might have high dimensional images as states, with many possible actions to choose from.

This thesis will work with the Atari 2600 environments offered by OpenAI (Brockman et al., 2016)

By combining deep learning techniques (Hinton and Salakhutdinov, 2006) with reinforcement learning, the problems posed by most of the Atari games can be solved nowadays.

However, complex environments like the Atari 2600 games can often be costly to simulate.

ACER (Wang et al., 2016) provides a sample efficient learning way to train the agent. This work aims at implementing and evaluating the algorithm.

To lay out the foundation for ACER, we will first discuss core components of reinforcement learning and take a closer look at policy gradient methods, specifically actor-critic methods and the *Advantage Actor Critic* (A3C) - algorithm (Mnih et al., 2016), followed by an overview of different approaches to off-policy learning from experience.

Finally, the ACER algorithm is presented, implemented and evaluated.

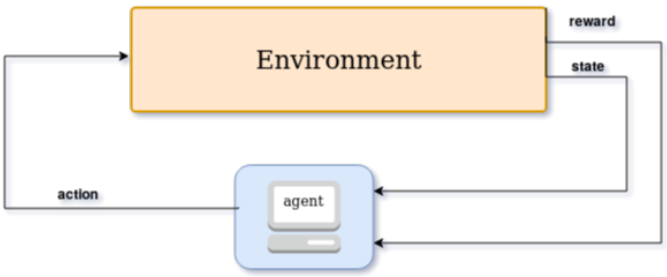

Figure 1: Agent interacting with the environment

2 Reinforcement Learning Framework

The main components of the reinforcement learning framework are the *agent* and the *environment*.

An agent interacts with the *environment* over time, by taking in the environment state, which is usually denoted as s or x. Within this work, s will be used as notation for the state. The agent evaluates the state and decides on an *action* (denoted as a), based on the current state.

A problem distinct to reinforcement learning is the one of exploration and exploitation. In order to learn the best behavior, the *agent* needs to make sure to explore the *environment*. Insufficient exploration can lead to suboptimal policies since a path that leads to a high reward might not be found.

Whenever the agent interacts with the environment, it receives a *reward* and the next state in return. Depending on whether the environment is terminal or not, an additional value denoting if the environment has terminated is received. The Atari games considered within this work terminate if the player either wins or loses the game.

2.1 Elements of Reinforcement Learning

Sutton and Barto (2018) name the following 4 other core elements of the reinforcement learning framework.

Policy

The behavior of the agent at any given time t is determined by the *policy*. A policy π can roughly be described as a mapping of states to an action or a distribution over actions. We denote the probability of an action a being taken at state s and time t under a policy π as

$$P_\pi(a \mid s_t) = \pi(a \mid s_t). \tag{1}$$

Within this work, the policy is always stochastic and the probability distribution over actions is denoted as $\pi(\bullet \mid s)$.

Reward Signal

The problem posed by an environment is defined through the reward function.

The goal of the learning agent is to receive the maximum accumulated future reward at any given time. One of the most important features of reinforcement learning is the fact, that rewards are often significantly delayed.

Connecting the delayed reward with the actions that caused it is a key task of reinforcement learning. A great example for this problem is the game Pong. The reward is caused by successfully playing the ball, however only many frames later, after the opponent missed the ball, the reward is received.

Games like chess pose an even bigger problem since every move can play an important role in winning the game. Good opening moves can strongly impact if the game is lost or won 100 moves later.

The reward received by an agent at timestep t will be denoted as r_t.

Value Function

The state-value represents the sum of the future rewards and indicates the long-term desirability of a state.

The value of a state at timestep t under a policy π is given through the expectancy of the *return* R_t , which is the possible accumulated future reward denoted as $V(s)$ and given by:

$$V^{\pi}(s) = E_{\pi}\{R_t \mid s_t = s\} = E_{\pi}\left\{\sum_{k=0}^{\infty} \gamma^k r_{t+k+1} \mid s_t = s\right\} \tag{2}$$

where γ denotes the discount factor. By discounting rewards, we can decide how important short-term rewards are in comparison to rewards in the distant future.

In accordance with this, we define the state-action value $Q(s, a)$, which is the expected return, if the action a is taken at state s.

$$Q^{\pi}(s, a) = E_{\pi}\{R_t \mid s_t = s, a_t = a\} = E_{\pi}\left\{\sum_{k=0}^{\infty} \gamma^k r_{t+k+1} \mid s_t = s, a_t = a\right\} \tag{3}$$

Additionally we define the advantage $A^{\pi}(s, a)$ for an *action* a at state s.

$$A^{\pi}(s, a) = Q^{\pi}(s, a) - V^{\pi}(s) \tag{4}$$

The advantage of an action given a state denotes how much better this action is, compared to the average action.

Environment Model

In order to solve a problem, a model of the environment can be learned and used for planning. A model can be used to predict future states and rewards before they happen. Model-based and model-free reinforcement learning methods, which explicitly learn by trial and error both play an important role in reinforcement learning.

Within this thesis, we will only look at model-free methods. Rather than learning a model, the agent learns directly through trajectories sampled from the environment.

2.2 Markov Decision Process

We can describe the sequential decision-making process of the *agent* more formally as a Markov decision process (MDP).

The sequential decision-making process is given by a sequence of states, actions and rewards:

$s_0, a_0, r_0, s_1, a_1, r_1, s_2, a_2, r_2, \ldots, s_t, a_t, r_t, s_{t+1}$

Such sequences are called *trajectories*.

Within this thesis, we assume the environment to be finite.

A state is called *Markov* or said to have *Markov property* if it only depends on its predecessor rather than the whole history.

$$P(s_{t+1} = s', r_{t+1} = r' \mid s_t, a_t, r_t, s_{t-1}, \ldots, r_1, a_0, s_0) = P(s_{t+1} = s', r_{t+1} = r' \mid s_t, a_t) \quad (5)$$

A finite discounted Markov decision process $MDP(S, A, P_a, R_a, \gamma)$ contains a finite set of states S, a finite set of actions A, the transition probablity to end up in state s' if action a is taken in state s :

$$P^a_{ss'} = Pr(s_{t+1} = s' \mid s_t = s, a_t = a) \quad (6)$$

the reward function $R^a_{ss'}$, and the discount factor $\gamma \in [0, 1)$.

2.3 Deep Reinforcement Learning

In contrast to directly mapping an input to an output value, deep learning algorithms contain so-called *hidden layers*.

Usually, a transformation or activation function is applied to the input of each hidden layer. Rectified linear units (ReLU), the tanh or the sigmoid function can be named as popular activation functions. After feeding an input into the network and calculating an error value, the weights are adjusted through backpropagation.

Convolutional neural networks (CNN) were inspired by visual neuroscience and are great tools to process image data. CNNs mainly consist of convolutional layers, pooling layers, and fully connected layers. Other relevant approaches are recurrent neural networks (RNN) or long-short-term memory networks (LSTM). Both mimic functions of a memory and have started to play bigger roles within the field of deep reinforcement learning recently.

Combining deep learning methods with reinforcement learning methods was a major breakthrough, enabling reinforcement learning methods to be successfully applied to complex problems like those posed by the Atari 2600 console (Li, 2017).

This thesis works with CNNs, consisting of convolutional layers, dense layers, and ReLU activation functions. Convolutional layers are used to extract features from image data.

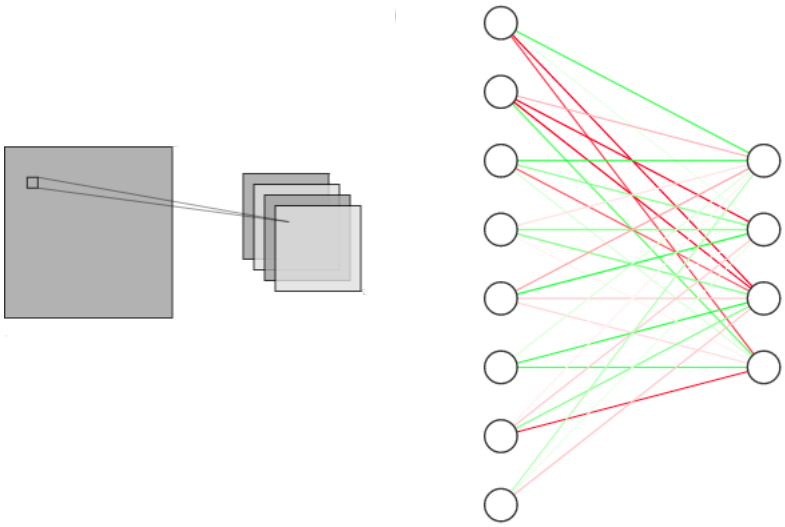

Figure 2: A convolutional (left) and fully connected (right) layer.

By moving filters over the image and calculating the weighted sum of the filter and the respective parts of the images the output is calculated.

The outputs are weighted sums of parts from the original image. The *stride* denotes how much a filter is shifted before being applied to the image again. Multiple filters are used to receive many different features.

Fully Connected Layers are used to weight the input and map it to the output space.

Each 'Neuron' or element in the output layer, holds a weighted sum of every input element (Figure 2).

The ReLU function changes all negative values to 0, while positive values remain untouched:

$$ReLU(x) = \begin{cases} x & \text{if x >0} \\ 0 & else \end{cases} \tag{7}$$

3 Actor-Critic Methods

A wide range of reinforcement learning problems and many different approaches to effectively solve them exist.

Dynamic programming methods can compute optimal policies, however, a perfect model of the environment as MDP is required.

Monte-Carlo methods on the other hand can estimate value functions and discover optimal policies by averaging over sampled trajectories without a model.

3.1 Monte-Carlo Predictions

A complete run of the environment from start to termination is called an *episode*. Monte-Carlo methods require complete episodes to learn. They estimate the value of a state by calculating the discounted returns for every encountered state and changing the current estimate slightly in the direction of the discounted return.

MC-methods learn relatively fast, however each update requires a full episode.

3.2 TD-Learning

Sutton and Barto (2018) describe *temporal difference* (TD) learning as one of the central ideas in reinforcement learning. TD learning combines dynamic programming and Monte-Carlo ideas to learn either *state-values: V(s)* or *state-action values: Q(s, a)*. Like Monte-Carlo methods, TD methods can learn directly from experience without the need of a model, but do not require finished episodes. Instead the encountered rewards are combined with an estimate of the discounted return following them. This is called *bootstrapping*.

We will take a look at the most basic TD method for learning a value function. The update step is given as:

$$V(s_t) \leftarrow V(s_t) + \alpha[r_{t+1} + \gamma V(s_{t+1}) - V(s_t)] \tag{8}$$

α denoted the learning rate. Another popular TD-Method is Q-Learning. It is used to estimate the state-action value function Q, and has a similar update step:

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha[r_{t+1} + \gamma \max_a Q(s_{t+1}, a) - Q(s_t, a_t)] \tag{9}$$

After learning a Q-function, the agent has a reliable method to estimate and maximize the return, considering the learned function resembles the *true state-value function* $Q*$ close enough. A common approach to achieve this is the use of a ϵ-greedy policy, where a gradually decreasing value ϵ is used to decide if a random action or the action assumed to be the best according to the learned function is taken.

This ensures both sufficient exploration and exploitation, as it starts with a (mostly) random policy, and ends up with a deterministic policy always choosing the action with the highest estimated return (Sutton and Barto, 2018).

Algorithm 1: Q-Learning (Sutton and Barto, 2018)

Initialize parameters for $Q(a, s)$ arbitrarily
repeat
  Initialize s
  repeat
    Choose a from s using a policy derived from Q (e.g. ϵ -greedy)
    Take action a, observe reward r and next state s'
    $Q(s,a) \leftarrow Q(s,a) + \alpha[r + \gamma \max_{a'} Q(s',a') - Q(s,a)]$ $s \leftarrow s'$
  until *s is terminal state*;
until *training finished*;

3.3 N-Step TD-Learning

The shown TD-Learning algorithms bootstrap after a single observation. This can become very inefficient with long episodes, especially if the final reward (e.g. winning or losing a game) is very important. Propagating this information along the trajectory requires many iterations. In general, TD-Learning is rather slow compared to learning from full returns.

We can approach this problem by using n-step TD-learning. Note that an '∞-step' TD update would be the same as a Monte-Carlo update.

By combining the Monte-Carlo target

$$R_t = r_{t+1} + \gamma r_{t+2} + \gamma^2 r_{t+3} + \gamma^3 r_{t+4} + \cdots + \gamma^{T-t-1} r_T \tag{10}$$

(T denotes the final step of the episode) with the TD approach, we can define the n-step TD-learning update:

$$V(s_t) = V(s_t) + \alpha(\gamma^n V(s_{t+n}) + R_{t:t+n} - V(s_t)) \tag{11}$$

where

$$R_{t:t+n} = r_{t+1} + \gamma r_{t+2} + \gamma^2 r_{t+3} + \cdots + \gamma^{n-1} r_{t+n} \tag{12}$$

denotes the discounted return for the next n steps.

3.4 Critic-Only Methods

The shown Q-learning algorithm or SARSA are popular critic-only methods.

They do not contain an explicit function for the policy, but derive it from the learned state-action values by acting greedy on the Q-Values.

Critic-only methods provide a low variance estimate of the expected returns, however the methods suffer from being biased and can be problematic in terms of convergence.

3.5 Actor-Only Methods

Unlike critic-only methods, actor-only methods do not learn any state or state-action values. Instead, they perform optimization steps directly on the policy π.

Usually the policy π is parameterized by θ, denoted as π_θ, but deterministic policy gradient methods are possible (Lillicrap et al., 2015).

Policy gradient methods like REINFORCE change the policy in order to maximize the average reward at a given timestep by performing gradient ascent (Williams, 1992). We define an objective function $J(\theta)$ which is a measurement of performance.

The learning agent tries to maximize $J(\theta)$ through gradient ascent

$$\theta_{t+1} = \theta_t + \alpha\widehat{\nabla J(\theta_t)} \tag{13}$$

where $\widehat{\nabla J(\theta_t)}$ denotes an approximated gradient of the performance measure. Methods of this schema are policy gradient methods (Sutton and Barto, 2018).

For the episodic case we define $J(\theta)$ to be the initial value of our policy. We should note that this only works under the assumption, that the environments always provides the same initial state.

$$J(\theta) \doteq v_{\pi_\theta}(s_0) \tag{14}$$

Because of the policy gradient theorem, the gradient of $J(\theta)$, $\nabla v_{\pi_\theta}(s_0)$ can be approximated as

$$\nabla J(\theta) \propto \sum_s \mu(s) \sum_a q_\pi(s,a) \nabla_\theta \pi(a \mid s, \theta). \tag{15}$$

The proof for this theorem can be found in the reinforcement learning book by Sutton and Barto (2018).

μ denotes the on-policy distribution under π.

However, if trajectories are sampled on-policy, we can rewrite this as the expectation under policy π:

$$\nabla J(\theta) \propto E_\pi \left[\sum_a q_\pi(s_t,a) \nabla_\theta \pi(a \mid s_t, \theta) \right] \tag{16}$$

Algorithm 2: REINFORCE by Williams (Sutton and Barto, 2018)

Initialize policy (π) parameters θ
repeat
  Generate episode: $s_0, a_0, r_1, \ldots, s_{T-1}, a_{T-1}, r_T$ following π
  for *step* $t \in 0, 1, \ldots, T$ **do**
    $R \leftarrow returnfromstept$
    $\theta \leftarrow \theta + \alpha\gamma^t R \nabla_\theta ln(a_t \mid s_t, \theta)$
until *forever*;

In contrast to value based approaches, policy gradient methods provide strong convergence to at least a local maximum. On top of that, actor methods are applicable on continuous action spaces (Sutton et al., 2000).

Actor-only methods however suffer from a large variance of the gradient. Compared to critic-only methods their learning process is significantly slower (Grondman et al., 2012).

3.6 Actor-Critic Methods

Actor-critic methods tackle the problem of high variance in policy gradient methods with the use of a critic.

They combine the strength of both approaches to achieve a learning agent, which has strong convergence, yet low variance.

Since actor-critic methods are still policy gradient methods at their core, they provide the possibility to work on continuous actions spaces like the actor-only approach.

The main reason for the variance in the gradient is the high variance of the return. By choosing a good baseline $b(s)$, to the objective function, we can achieve a lower variance, without creating bias. The idea behind this is to increase the probability of an action in relation to how much better it is compared to other actions at this state, rather than the full return.

$$\nabla J(\theta) \propto \sum_s \mu(s) \left[\left(\sum_a q_\pi(s,a) - b(s) \right) \nabla_\theta \pi(a \mid s, \theta) \right] \tag{17}$$

We can substitute $q_\pi(s_t, a) - b(s)$ with different terms (Schulman et al., 2015). Popular choices for actor-critic learning are the advantage function A_π (4) or the TD residual

$$r_t + V_\pi(s_{t+1}) - V_\pi(s_t) \tag{18}$$

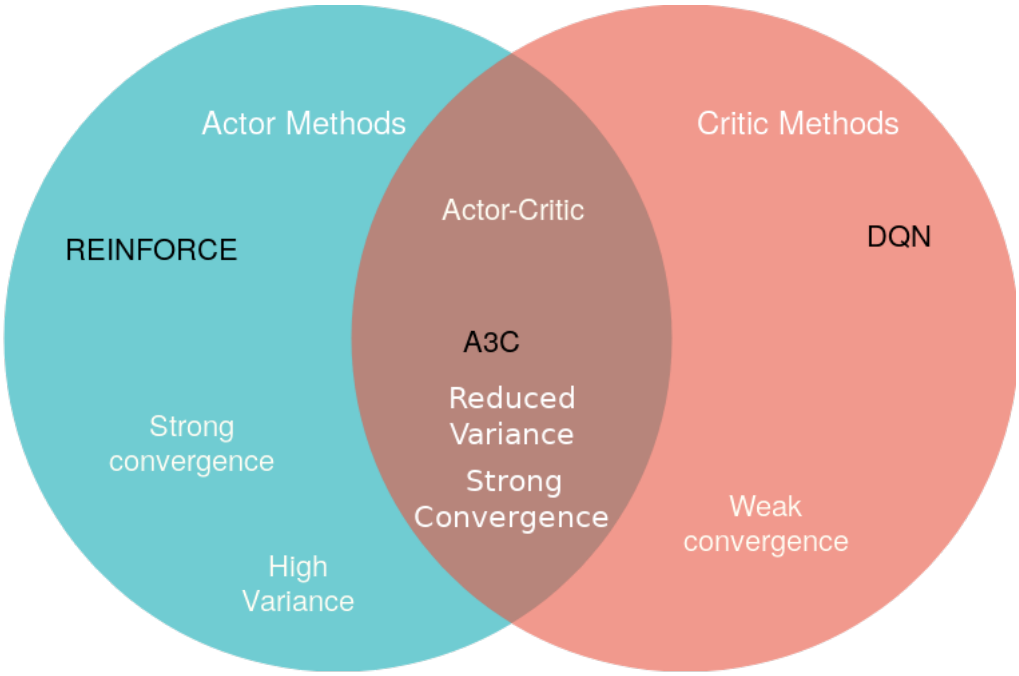

Figure 3: Actor, Critic, and Actor-Critic approach

3.7 Asynchronous Advantage Actor Critic (A3C)

In general we call an algorithm onpolicy, if the data used in the policy update was sampled under the same policy. The sequence of observed data encountered by an RL agent is strongly correlated and nonstationary (Mnih et al., 2016). This can have a negative influence on the onpolicy learning process.

A previous approach to this problem was to achieve decorrelation by using randomly selected samples from a replay memory (Mnih et al., 2013).

Training an agent comes with a high demand for computational power. To achieve feasible training times, former algorithms heavily relied on a strong GPU.

The asynchronous advantage actor critic (A3C) algorithm solves both problems, by training simultaneously on multiple environments. Each learner samples trajectories and computes gradients. Those gradients are then applied to the shared parameters. After each global update step, the local parameters are synchronized.

This method not only enables efficient CPU computation with multiple threads, rather than requiring a strong GPU, but solves the problem of correlation since every environment can be assumed to be in different states.

Usually 16 or more environments are used to ensure proper decorrelation of the samples.

Algorithm 3: A3C (Mnih et al., 2016)

// *Assume shared Parameters and counter* $\theta^{global},\ \theta_v^{global},\ T = 0$
// *with local conterparts* $\theta,\ \theta_v$
Initialize parameters for policy θ and critic θ_v and counter t=1
repeat
  Reset gradients $d\theta, d\theta_v$
  Synchronize θ, θ_v with $\theta^{global}, \theta_v^{global}$
  $t_{start} = t$
  get state s_t
  repeat
    Perform a_t according to policy $\pi(a_t \mid s_t; \theta)$
    Receive reward r_t and next state s_{t+1}
    $t \leftarrow t + 1$
    $T \leftarrow T + 1$
  until s_t ***is*** *terminal or* $(t - t_{start}) == t_{max}$*;*
  $R = \begin{cases} 0 & s_t \text{ terminal} \\ V_\theta(s_t) & s_t \text{ not terminal} \end{cases}$
  for $i \in \{t-1, \ldots, t_{start}\}$ **do**
    $R \leftarrow r_i + \gamma R$
    Accumulate gradients wrt.
    $\theta : d\theta^{global} \leftarrow d\theta^{global} + \nabla_\theta \log \pi_{\theta'}(a_i|s_i)(R - V_{\theta_v}(s_i))$
    Accumulate gradients wrt. $\theta_v : d\theta_v^{global} \leftarrow d\theta_v^{global} + \partial(R - V_{\theta_v}(s_i))^2/\partial\theta_v$
  Perform asynchronous update of θ^{global} and θ_v^{global} using dθ, dθ_v
until $T > T_{max}$;

A3C samples trajectories of length n and uses the longest possible k-return for the update step, meaning the last state uses a one-step update, the second to last a two-step update and so on, with the first state using a n-step update. The gradients are accumulated over all states within the trajectory and applied in a single gradient step.

4 Off-Policy Learning

Off-policy methods use data previously sampled under a so-called *behavior policy* denoted as $\mu(a \mid s)$ which is different to the *current/target policy* π that is optimized.

One benefit of off-policy learning is the possibility to choose a more exploratory behavior policy. Another benefit is the possibility to increase sample efficiency by reusing old data (Degris et al., 2012).

Off-policy methods can have the drawback of being divergent. Asynchronous methods like A3C are always slightly off-policy, but since the behavior policy is close enough to the target policy, the influence is neglectable.

Whenever the behavior policy differs too much from the target policy, the algorithm can no longer be viewed as *safe* without correcting for the 'off-policyness' (Munos et al., 2016).

Ensuring convergence even for an arbitrary level of 'off-policyness' is a problem addressed by a multitude of methods. We will look at the most important ones, leading up to the retrace-algorithm used for our experiments.

4.1 Importance Sampling (IS)

One of the most basic ideas is to correct for the "off-policyness" by using importance sampling (IS). It is a classic and well-known technique for estimating the value of a random variable x with distribution d if the samples were drawn from another distribution d' by using the product of the likelihood ratios.

In regards to π and μ, the importance weight is denoted as

$$p_t = \frac{\pi(a_t \mid s_t)}{\mu(a_t \mid s_t)}. \tag{19}$$

Even though this method can guarantee convergence (Munos et al., 2016) for arbitrary π and μ, it comes with the risk of high, possible infinite variance, due to the variance of the product of importance weights.

4.2 Tree-backup, TB(λ)

The tree-backup method allows off-policy corrections, without the use of importance sampling by using estimated values of untaken actions as an offset to the rewards from the off-policy sample Precup et al. (2000).

The algorithm provides low variance off-policy learning with strong convergence. However, if a sample is drawn from a policy which is close to the target policy, the algorithm unnecessarily cuts the traces. Without using the full returns, the learning process is slowed down.

4.3 Retrace(λ)

Munos et al. (2016) introduced the retrace(λ) algorithm. By combining ideas of importance sampling and tree-backup, a method with strong convergence, yet low variance was achieved that is still able to use the benefits of full returns.

Similar to how it is done in TB(λ), the traces are safely cut in case of strong "off-policyness", but without impacting the update too much, if the data was sampled under a behavior policy μ close to the target policy π.

Retrace values for a q funcion are obtained recursively by

$$Q^{ret}(x_t, a_t) = r_t + \gamma \tilde{p}_{t+1}[Q^{ret}(x_{t+1}, a_{t+1}) - Q(x_{t+1}, a_{t+1})] + \gamma V(x_{t+1}) \tag{20}$$

with $\tilde{p} = min\{c, p_t\}$ being the truncated importance weight p_t (19).

In case of a terminal state, the retrace value is equal to the final reward. Note that the formula is given considering $\lambda = 1$.

As $\lambda = 1$ performs the best for the Atari console games (Munos et al., 2016), other values were not considered within this thesis.

5 Actor-Critic with Experience Replay (ACER)

"Actor critic with experience replay" (ACER) introduced by Wang et al. (2016) was one of the first approaches to create a sample efficient, yet stable actor-critic method, that applies to both continuous and discrete action spaces.

ACER combines recent breakthroughs in the field of RL, by utilizing both the resource efficient parallel training of RL agents proposed by Mnih et al. (2016) and the retrace algorithm (Munos et al., 2016).

These approaches were combined with truncated importance sampling with bias correction and an efficient trust region policy optimization. For continuous action spaces, stochastic dueling network architectures were used.

ACER can be viewed as an off-policy extension of A3C (Mnih et al., 2016). To understand the ACER-gradient, we start from the importance weighted policy gradient, which is given by:

$$\hat{g}^{imp} = \left(\prod_{t=0}^{k} p_t\right) \sum_{t=0}^{k} \left(\sum_{i=0}^{k} \gamma^i r_{t+i}\right) \nabla_\theta \, log \, \pi(a_t \mid s_t) \tag{21}$$

A product of unbounded importance weights can cause high variance. Degris et al. (2012) approached this problem by approximating the policy gradient as

$$g^{marg} = E_{s_t \sim \beta, a_t \sim \mu} \left[p_t \nabla_\theta log \pi_\theta(a_t \mid s_t) Q^\pi(s_t, a_t) \right] \tag{22}$$

where β denotes the limiting distribution and μ the behavior policy.

To compute this gradient, knowledge about Q^π is necessary. We can use the retrace values (20) as a good estimation for Q^π.

To further reduce the variance, the importance weights are truncated.

By truncating the importance weights, bias is introduced. Finding a good tradeoff between variance and bias is essential for the success of an RL-algorithm. To counter this, ACER uses a bias correction term.

We will replace the importance weight p_t in (22) with their truncated terms

$\bar{p_t} = min\{c, p_t\}$, where c denotes the truncation value.

The introduced bias correction term is given as:

$$E_{x_t} \left[E_{a \sim \pi} \left(\left[\frac{p_t(a) - c}{p_t(a)} \right]_+ \nabla_\theta log \pi_\theta(a \mid s_t) Q_\pi(s_t, a) \right) \right] \tag{23}$$

To receive the ACER policy gradient we replace the Q terms with our approximations. For the gradient term, the retrace values are used as an approximation, which further reduces variance in comparison to using the estimated Q-function. Q_π in the correction term is estimated by our critic.

Finally, we use the approximation of the value function, which is retrieved by taking the expectation of Q_{θ_v} under π_θ as a baseline by subtracting them from the Q terms.

This leaves us with the ACER policy gradient:

$$\begin{aligned} g_t^{ACER} &= \tilde{p}_t \nabla_\theta log \pi_\theta(a_t \mid s_t) \left[Q^{ret}(s_t, a_t) - V_{\theta v}(s_t) \right] \\ &+ E_{a \sim \pi} \left(\left[\frac{p_t(a) - c}{p_t(a)} \right]_+ \nabla_\theta log \pi_\theta(a \mid s_t) \left[Q_{\theta v}(s_t, a) - V_{\theta v}(s_t) \right] \right) \end{aligned} \tag{24}$$

Note that the expectation over states is no longer necessary because it is approximated by sampling trajectories generated by μ.

The critic $Q_{\theta v}$ is trained by minimizing the mean squared error with the retrace values as the target.

The critic loss function is given as:

$$(Q^{ret}(s_t, a_t) - Q_{\theta_v}(s_t, a_t))^2 \tag{25}$$

and has the standard gradient:

$$(Q^{ret}(s_t, a_t) - Q_{\theta_v})) \nabla_{\theta_v} Q_{\theta_v}(s_t, a_t)) \tag{26}$$

An entropy term of the policy π is added to the optimization function to encourage exploration. Adding the entropy to our objective empirically performs better and can prevent early convergence to suboptimal policies.

The entropy for a policy π is given by:

$$H(\pi) = -\sum_{i=1}^{A} \pi(a_i) \log_b \pi(a_i) \tag{27}$$

where b denotes the basis for the logarithm. A common choice for the base is 2, 10 or e (Jost, n.d.).

Wang et al. (2016) proposed an efficient trust region policy optimization (TRPO), which limits the update step in a way, that the new policy doesn't deviate too much from an average policy.

Even though the TRPO method significantly improves the performance for continuous action space environments, the results presented in the paper only showed a marginal improvement for discrete action space environments. TRPO comes with a relatively high computational cost in comparison to its benefits on the discrete action space. Since only discrete action spaces were considered within this work in order to conduct more experiments within the limitations of time and computational resources, the trust region optimization was not used.

Pseudocode for the algorithm used can be found at the end of the thesis (A).

5.1 Experimental Setup

5.1.1 Environment

All experiments were made using multiple selected games from the Atari 2600 console game environments provided by Brockman et al. (2016).

In our experiments the following OpenAI-Gym environments were used:

Breakout is a game that rapidly speeds up, making it easy for machines to show "super-human" performance.

Seaquest poses a serious problem for learning agents, as they tend to get stuck on local maxima. It is especially interesting, as it provides multiple possibilities to achieve rewards. The loss of the game if the player runs out of oxygen is a game mechanic a human can grasp in an instant, but it is notoriously hard for reinforcement learning agents to 'understand'.

Space Invaders is a fast-paced game. Human and reinforcement learning approaches for DQN achieved similar scores (Mnih et al., 2015).

Riverraid is another game where humans manage to heavily outperform approaches like DQN.

The hyperparameters were adjusted to ensure a good performance for the Breakout environment. Tests on other environments were all performed with the same set of hyperparameters.

OpenAI gym environments provide the agent with a 210x160 pixel colored image. OpenAI uses a variation of the frame skipping method proposed by Mnih et al. (2015). Whenever the agent takes an action within the environment, instead of using it once and observing every state, the action is repeated over $k \in \{2, 3, 4\}$ game steps, and only every k'th observation is returned to the agent. Frame skipping reduces the workload of the agent and the length of the episodes, enabling a faster learning process.

5.1.2 Preprocessing

Using full scale colored images would come with a huge computational cost, resulting in a very slow learning process. To make efficient use of the data, each frame is preprocessed. First, the scoreboard area on top of the frame is chopped off, leaving roughly the playing area behind.

The playing area is then grayscaled and downscaled. With this method a 84x84 grayscaled pixel image is obtained.

Different grayscaling methods exist (Intensity, Luma, Lightness,...). Within this thesis, we worked with the luminance. It is given as

$$G_{Luminance} \leftarrow 0.3R + 0.59G + 0.11B, \tag{28}$$

with R, G and B denoting the red, green and blue values.

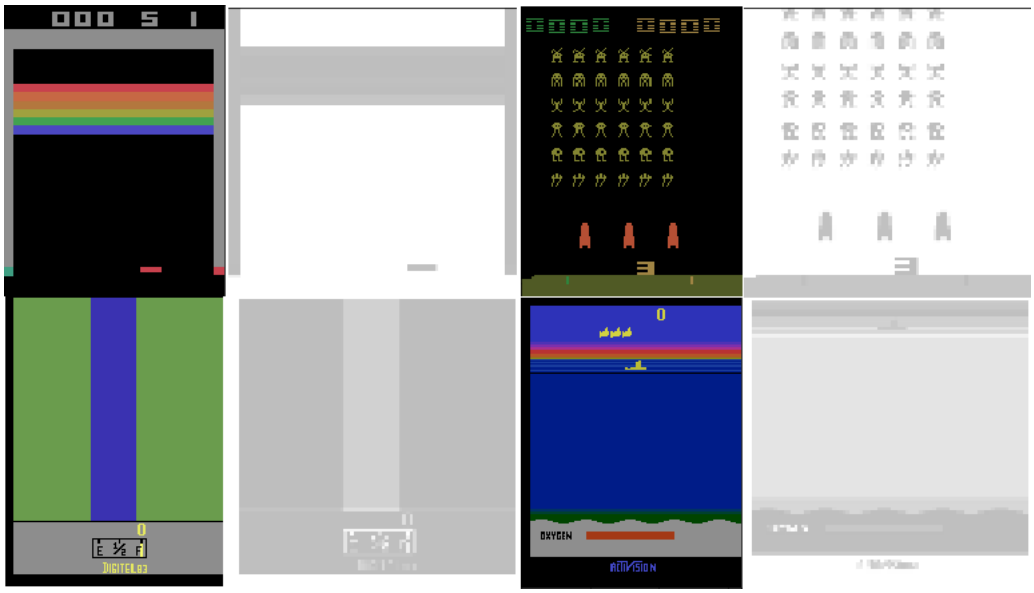

Figure 4: The Atari 2600 console games Breakout, SpaceInvaders, Riverraid and Seaquest before and after being processed.

Mnih et al. (2015) proposed to stack multiple frames. More specifically the state contains the last 4 frames received and therefore has a shape of 84x84x4.

To receive the initial state, we stack the first frame 4 times. This can convey a feeling of movement to the agent.

A high variance within the reward signal seems to negatively impact the learning of an agent. A very common method to achieve good convergence is to limit the reward. It was found to be a good method to simply keep the sign of the reward, and ignore the magnitude, so rewards are either -1, 0 or 1. This method was adopted in this paper.

5.1.3 Network Architecture

Two architectures were tested (Mnih et al., 2013; Mnih et al., 2015). Both provided good results. Since the ACER paper uses the architecture proposed in the nature paper (Mnih et al., 2015), all further experiments were done using this architecture.

Similar to the A3C (Mnih et al., 2016) network, most of the parameters are shared and used for both estimating Q^π and calculating the policy π.

The shared net consists of 3 convolutional layers. First 32 8x8 filters are applied with a stride of 4. The 2nd layer consists of 64 4x4 filters using a stride of 2. The last convolutional layer uses 64 3x3 filters with stride 1. Finally, a fully connected layer of size 512 is applied.

Each of the mentioned layers is followed by a rectifier nonlinearity (ReLU) function. ReLU is used to set all negative outputs to 0, ensuring only positive values are passed to the next layer.

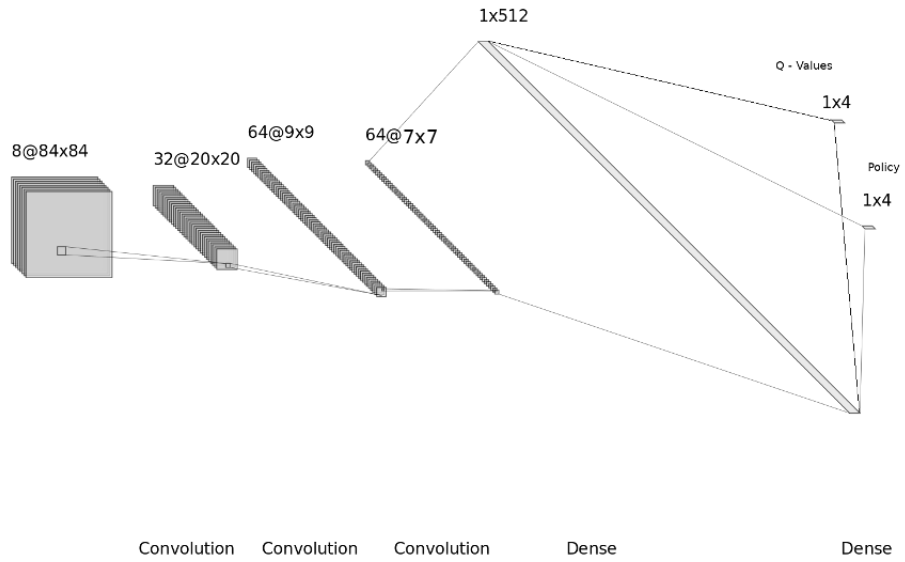

Figure 5: Shared network for policy and Q-Value approximization for an environment with 4 possible actions

A test run using tanh instead of ReLU as activation function caused a significant loss of performance.

We mapped the final shared layer to the action space twice to obtain the action scores, and the Q-values. A softmax policy is used to receive the distribution over actions. The action is then randomly sampled from the softmax probabilities.

$$P(A = a'|s) = \frac{e^{z_\theta(a')}}{\sum_{i=0}^{N} e^{z_\theta(a_i)}} \tag{29}$$

where $z_\theta(a_i)$ denotes the score assigned to the i'th action through the last network layer.

The agent is trained by using 16 learner threads running on the CPU. Each thread has its own replay buffer, which ensures a more balanced replay and reduces the risk of a single trajectory being used unreasonably often compared to a single shared replay buffer.

Updates to the network were performed every 20 steps or if the environment reached a terminal state.

Replay buffers were designed to hold up to 2500 trajectories, therefore 50.000 frames were saved for each thread and ~ 800.000 frames were stored in total, which is the same size used by DQN (Mnih et al., 2015).

If no replay is used, ACER essentially becomes a version of A3C which uses retrace and Q-values, rather than learning the value function (Mnih et al., 2016). For the offline setting, replay ratios of 1, 2 and 4 were considered, where a ratio of 4 would mean, that the net is trained on trajectories sampled from the replay buffer 4 times for every online update.

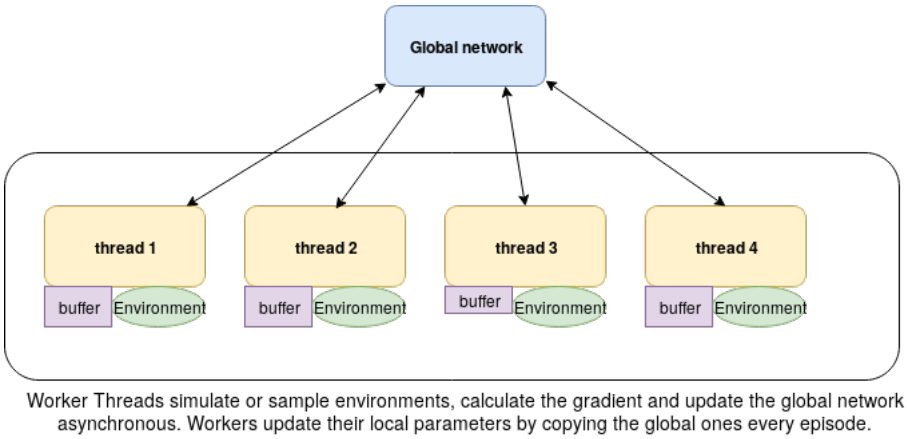

Figure 6: Example setup for 4 thread ACER

The update was performed using an RMSProp Optimizer. The RMSProp update is given by:

$$g = \alpha g + (1 - \alpha)\delta\theta^2 \tag{30}$$

$$\theta \leftarrow \theta - \eta \frac{\delta\theta}{\sqrt{g + \epsilon}} \tag{31}$$

where η denotes the learning rate and α is the momentum. ϵ is a small value to prevent dividing by 0.

RMSProp has shown to be very robust for A3C (Mnih et al., 2016).

A single optimizer is used for all threads, which gives a smoother momentum and shows better results compared to using a seperate optimizer for each thread.

Since RMSProp optimizers in tensorflow, which was used for the implementation, can only use gradient decent, we consider the objective function as a loss and use gradient decent. Instead of having seperate updates for policy, value function and entropy, a single loss is formed and optimized.

$$L_{ACER} = -L_{Policy} + w_q L_Q - \beta Entropy \tag{32}$$

with L denoting the loss. w_q and β are hyperparameters used to decide how much influence the value loss and the entropy should have on the parameters compared to the policy loss. Common values for w_q are 0.5 or 0.25, whereas the entropy is usually weighted with 0.01 or 0.001.

5.2 Hyperparameter Settings

Unless stated differently, the following hyperparameters were used.

Hyperparameters		
Parameter	Value	Explanation
LR	8e-5	learning rate
Discount	0.99	
return steps	20	
c	10	truncation value
β	0.01	weight of entropy in the loss function
w_q	0.25	weight of value loss.
RMSProp decay	0.99	
global norm	10	gradient clipping
ϵ	1e-6	clipping value denoting the distance to 0 and 1 a probability is allowed to reach

6 Results

The reward given in the graphics for the y-axis is the capped reward used to train the agent. The decision to use the capped rewards was made due to their lower variance, and because these rewards were used when training the agents. Using the capped reward to visualize the results gives a better comparison between the replay ratios, which is the main focus of this thesis.

The x-axis denotes the on-policy steps for the results on sample efficiency, and the computation time. In our experiments 100.000 on-policy updates on a CPU using 16 threads took about an hour.

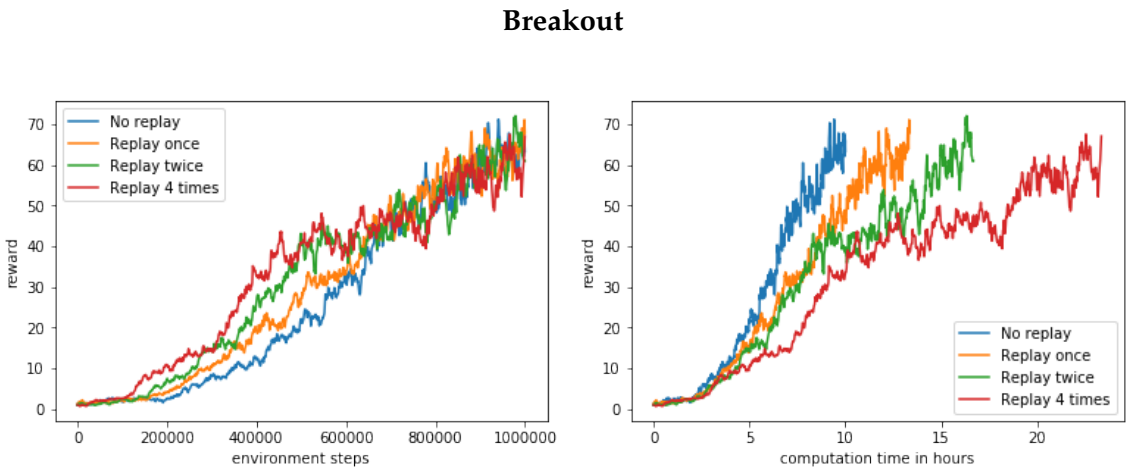

Figure 7: *ACER results for Breakout in terms of sample efficiency (left) and computation time (right).*

We can see that ACER is clearly more sample efficient with a higher replay ratio and quickly reaches better rewards. The right side shows the performance in regards to the computation time.

During our experiments, we found an online step to take about 3 times longer than an offline update. In regards to the Breakout environment, we can conclude, that using a higher replay ratio (within the tested ratios) offers better performance in terms of sample efficiency, but can not match the learning speed of the online agent.

Space Invaders

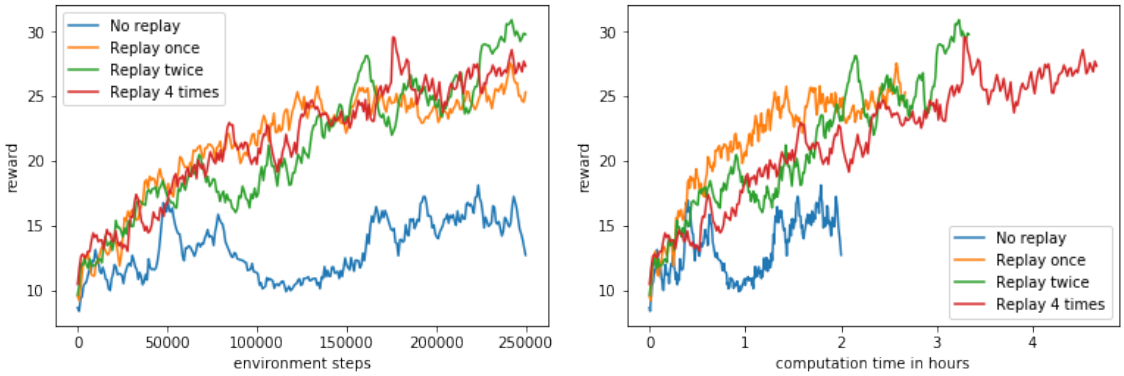

Figure 8: *Results for Space Invaders in terms of sample effciency (left) and computation time (right).*

The poor performance of the online variant on Space Invaders came unexpected. Assuming this performance was caused by an unlucky initialization, the same test was performed again, but very simlar results were observed. In comparison, the agents using replay had a much smoother performance curve. Unlike with the Breakout environment, a higher replay ratio did not result in a faster learning process.

Seaquest

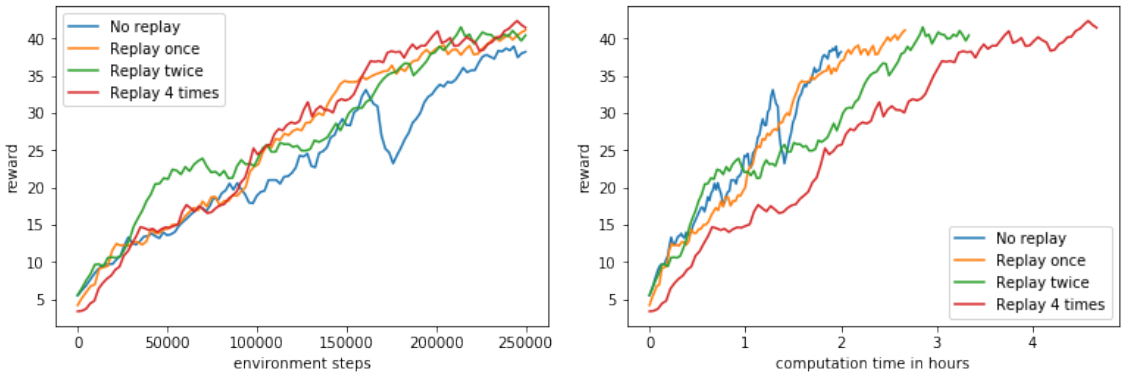

Figure 9: *ACER results for Seaquest in terms of sample efficiency (left) and computation time (right).*

No significant performance increase in terms of sample efficiency could be observed. Independent of the replay ratio, all agents learned relatively even in terms of sample efficiency. Using a replay ratio of 1 provided a smooth and efficient learning process in terms of both processing time and sample efficiency. A significant drop in performance was observed for the online agent. In comparison all the replay agents provided a smooth and stable learning process.

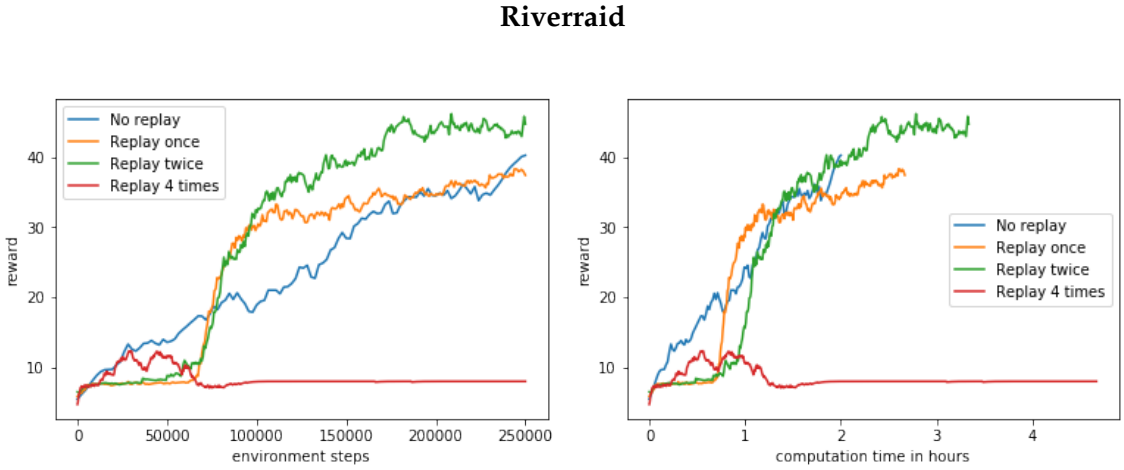

Figure 10: *ACER results on the Riverraid environment in regards to sample efficiency (left) and computation time (right).*

The experiments conducted on the Riverraid environment showed a rather slow learning in the beginning for the replay agents, with the replay ratio of 4 getting stuck on a low reward. A possible explanation could be, that the agent uses the trajectories sampled at the beginning of the learning process too often.

An attempt to fix this problem is to fill up a certain amount of the replay buffer first, and start the experience replay afterwards, to not reuse the same trajectory too often in the early learning process.

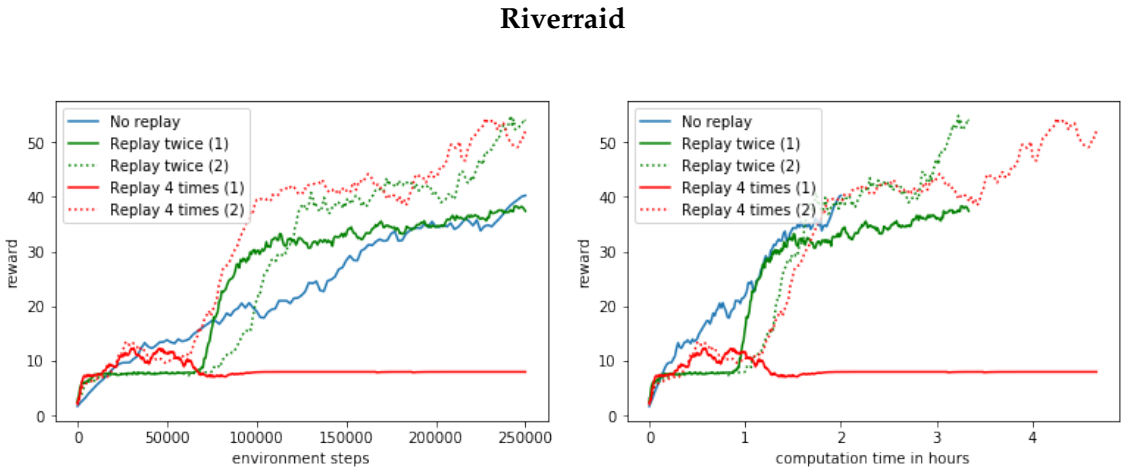

Figure 11: *ACER results on the Riverraid environment in regards to sample efficiency (left) and computation time(right). (1) Uses replay throughout the full learning process. (2) Started replay after 1000 trajectories were saved to the buffer.*

The problem of slow early learning could still be observed even with the replay starting later. To make sure both results for the replay-ratio of four were no coincidences further test are necessary. In general the performance seems to improve.

Breakout

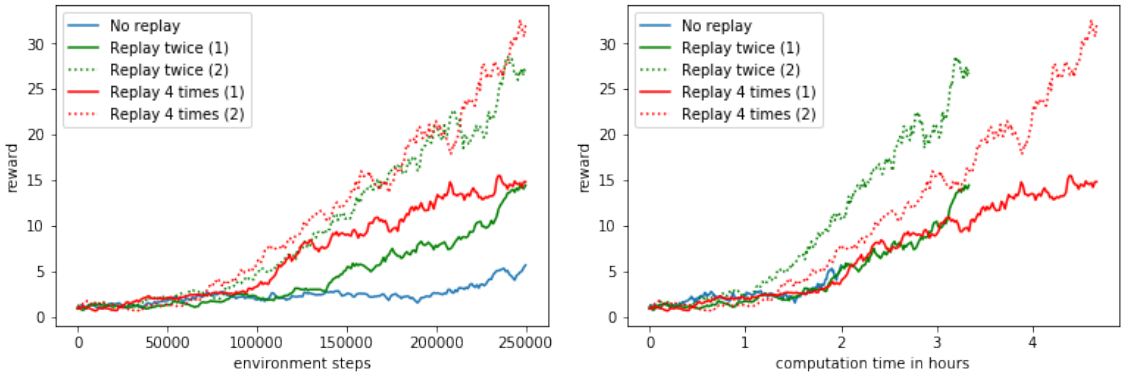

Figure 12: *ACER results on the Breakout environment in regards to sample efficiency (left) and computation time(right). (1) Uses replay throughout the full learning process. (2) Started replay after 1000 trajectories were saved to the buffer.*

The replay agents for Breakout showed much faster learning if the replay buffer was filled with samples first.

Seaquest

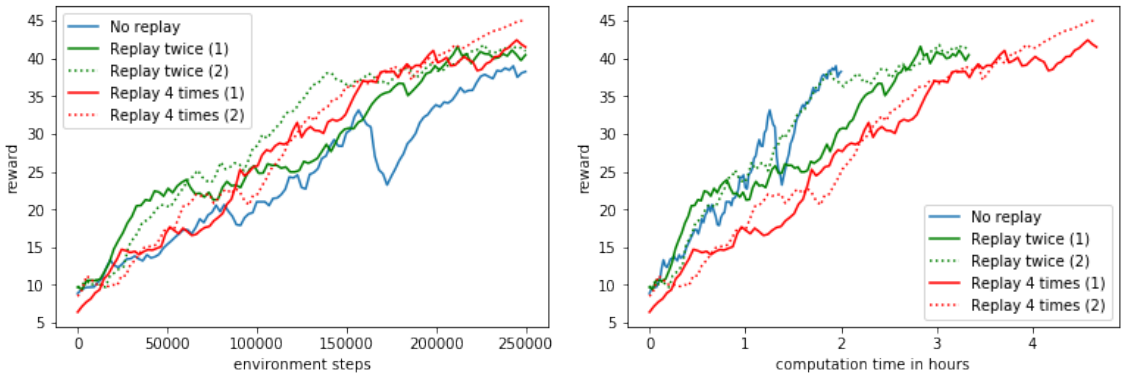

Figure 13: *ACER results on the Seaquest environment in regards to sample efficiency (left) and computation time(right). (1) Uses replay throughout the full learning process. (2) Started replay after 1000 trajectories were saved to the buffer.*

In contrary to the results for Breakout and Riverraid, no notable improvements were observed for the Seaquest environment.

Breakout

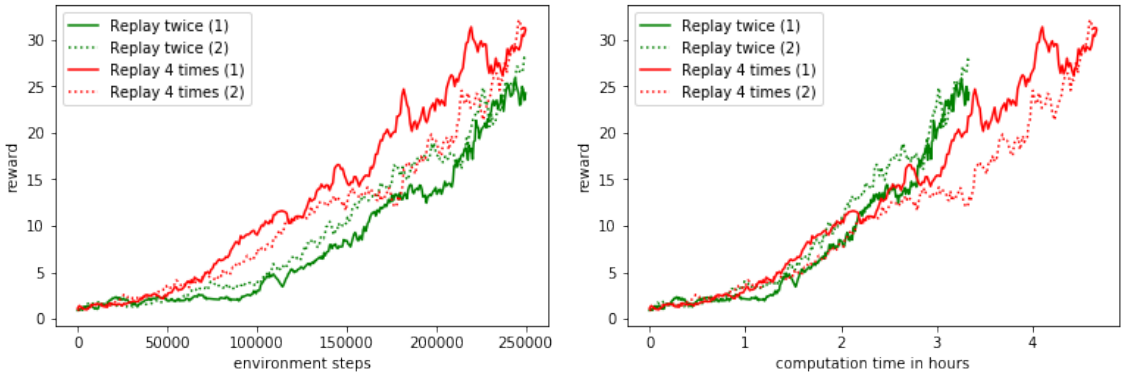

Figure 14: *ACER results on the Breakout environment in regards to sample efficiency (left) and computation time(right). (1) With bias correction and (2) without bias correction*

Space Invaders

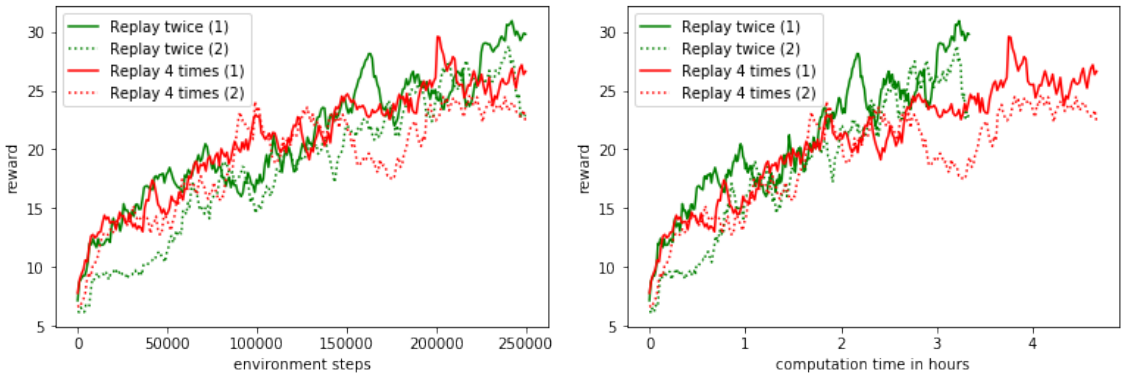

Figure 15: *ACER results on the Space Invaders environment in regards to sample efficiency (left) and computation time(right). (1) With bias correction and (2) without bias correction*

Replay agents managed to perform well on both the Breakout and the Space Invaders environment even without the use of the bias correction term for the gradient.

7 Preprocessing Atari Environments

Atari environments were specifically designed for humans. Important information might be conveyed through text messages or symbols to a human, but might be heavily underrepresented within the pixel data.

A good examples for this problem is the Sequest environment.

Seaquest often gets learning agents stuck on a local maximum because unlike a human they do not refill the oxygen. A human player managed to outperform DQN (Mnih et al., 2015) gaining four times the reward on average for Seaquest. This is in line with the observations within this thesis. Figure 9 shows the agents approaching a cap around 40 reward.

Even though future methods might overcome these problems, we can assume, that a well preprocessed environment will always outperform if compared with a poorly or non preprocessed environment. Through smart preprocessing even very basic policy gradient methods can solve environments like Pong (Karpathy, n.d.).

The core problem Seaquest has, is that the agent often cannot link the oxygen bar depleting to the end of the game. An attempt to increase the influence of the oxygen bar on the actions, could be to use space not relevant to the game to increase the size of the bar, therefore increasing its influence on the policy. In addition, the hyperparameters for the return step: $k = 50$ and $\gamma = 0.995$ were adjusted to make future rewards more attractive to the learning agent.

Figure 16: *Seaquest environment after normal (left) and individual (right) preprocessing*

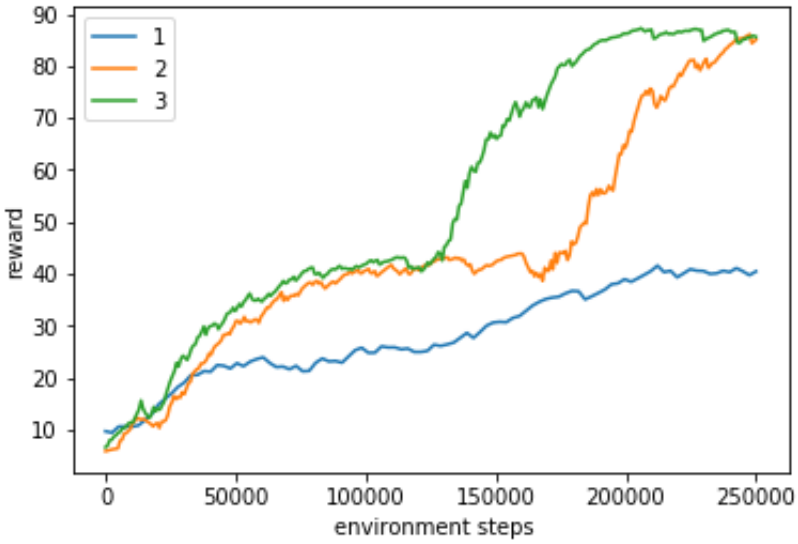

Figure 17: *Seaquest for default settings* ***(1)****, changed hyperparameters* ***(2)*** *and changed hyperparameters with custom preprocessing* ***(3)****. A replay ratio of 2 was used for all 3 runs.*

By applying individual preprocessing to the Seaquest environment, and slightly adjusting the hyperparameters, we managed to both accelerate the learning process and break through the local maximum the learning agents got stuck on in the previous tests.

Even though the results indicate an improved performance for the individually preprocessed environment, a clear statement cannot be made, because an agent learning on the normally preprocessed environment, managed to outperform the default settings and break through the local optimum in a similar way, with only the change in hyperparameters.

8 Conclusion

We were able to show, that ACER provides a sample efficient method for training agents on the Atari console by reusing experience.

Within our experiments, a replay ratio of 2 showed a good improvement in terms of sample efficiency, without increasing the computational cost too much.

Interesting findings were the fact, that starting the replay process too early might have a negative impact on the learning process. Further experiments will be necessary to determine a good starting point for the replay process.

Through preprocessing and changing the hyperparameters for the Seaquest environment, a much better learning process was achieved. This shows, that using a single set of hyperparameters for all Atari environment as a benchmark for the performance of a method might not be the ideal approach, since most environments pose unique problems.

Even without the bias correction, the agents were learning fine. More experiments on different environments would be necessary to determine if the bias correction term benefits the learning process.

A ACER-Algorithm

Algorithm 4: ACER for discrete actions (without TRPO) (Wang et al., 2016)

// *Assume shared Parameters and counter* θ^{global}, $T = 0$
// *with local conterparts* t *and* θ
Initialize parameters for policy θ and critic θ_v and counter t=0
repeat
  Reset gradient $d\theta$
  if *online* **then**
    Synchronize θ with θ^{global}
    get trajectory of length k by interacting with the environment
    save trajectory to replay buffer.
  else
    Draw trajectory of length k from replay buffer
  $\mathrm{Q}^{ret} = \begin{cases} 0 & s_k \text{ terminal} \\ V_\theta(s_k) & s_k \text{ not terminal} \end{cases}$
  for $i \in \{k-1, \ldots, 0\}$ **do**
    $Q^{ret} \leftarrow r_i + \gamma Q^{ret}$
    $\tilde{p_i} \leftarrow min\{1, \frac{\pi(s_i)}{\mu(s_i)}\}$
    $g \leftarrow \min\{c, p_i(a_i)\}\nabla_\theta log(a_i \mid s_i)(Q^{ret} - V_i)$
    $+ \sum_a \left[1 - \frac{c}{p_i(a)}\right]_+ \pi(a \mid s_i)\nabla_\theta log\pi(a \mid s_i)(Q_\theta(s_i, a_i) - V_i)$
    Accumulate gradients wrt. θ : $d\theta^{global} \leftarrow d\theta^{global} + g + \nabla_\theta(Q^{ret} - Q_\theta(s_i, a_i))^2$
    $Q^{ret} \leftarrow \tilde{p_i}(Q^{ret} - Q_\theta(s_i, a_i)) + V_\theta(s_i)$
  Perform asynchronous update of θ^{global} using dθ
  T = T+1
  t=t+1
  if *t % replay_ratio == 0* **then**
    online = True
  else
    online = False
until $T > T_{max}$;

References

Greg Brockman, Vicki Cheung, Ludwig Pettersson, Jonas Schneider, John Schulman, Jie Tang, and Wojciech Zaremba (2016). "OpenAI Gym". In: *CoRR* abs/1606.01540.

Thomas Degris, Martha White, and Richard S. Sutton (2012). "Off-Policy Actor-Critic". In: *CoRR* abs/1205.4839.

Ivo Grondman, Lucian Busoniu, Gabriel A. D. Lopes, and Robert Babuska (2012). "A Survey of Actor-Critic Reinforcement Learning: Standard and Natural Policy Gradients". In: *Trans. Sys. Man Cyber Part C* 42.6, pp. 1291–1307.

G. E. Hinton and R. R. Salakhutdinov (2006). "Reducing the Dimensionality of Data with Neural Networks". In: *Science* 313.5786, pp. 504–507. eprint: http://science.sciencemag.org/content/313/5786/504.full.pdf.

Lou Jost. "Entropy and diversity". In: *Oikos* 113.2, pp. 363–375. eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1111/j.2006.0030-1299.14714.x.

Andrej Karpathy. *Deep Reinforcement Learning: Pong from Pixels*.

Yuxi Li (2017). "Deep Reinforcement Learning: An Overview". In: *CoRR* abs/1701.07274.

Timothy P. Lillicrap, Jonathan J. Hunt, Alexander Pritzel, Nicolas Heess, Tom Erez, Yuval Tassa, David Silver, and Daan Wierstra (2015). "Continuous control with deep reinforcement learning". In: *CoRR* abs/1509.02971.

Volodymyr Mnih, Adrià Puigdomènech Badia, Mehdi Mirza, Alex Graves, Timothy P. Lillicrap, Tim Harley, David Silver, and Koray Kavukcuoglu (2016). "Asynchronous Methods for Deep Reinforcement Learning". In: *CoRR* abs/1602.01783.

Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin A. Riedmiller (2013). "Playing Atari with Deep Reinforcement Learning". In: *CoRR* abs/1312.5602. arXiv: 1312.5602.

Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A. Rusu, Joel Veness, Marc G. Bellemare, Alex Graves, Martin Riedmiller, Andreas K. Fidjeland, Georg Ostrovski, Stig Petersen, Charles Beattie, Amir Sadik, Ioannis Antonoglou, Helen King, Dharshan Kumaran, Daan Wierstra, Shane Legg, and Demis Hassabis (2015). "Human-level control through deep reinforcement learning". In: *Nature* 518.7540, pp. 529–533.

Rémi Munos, Tom Stepleton, Anna Harutyunyan, and Marc G. Bellemare (2016). "Safe and Efficient Off-Policy Reinforcement Learning". In: *CoRR* abs/1606.02647.

OpenAI (2018). *OpenAI Five*.

Doina Precup, Richard S. Sutton, and Satinder P. Singh (2000). "Eligibility Traces for Off-Policy Policy Evaluation". In: ICML '00, pp. 759–766.

John Schulman, Philipp Moritz, Sergey Levine, Michael I. Jordan, and Pieter Abbeel (2015). "High-Dimensional Continuous Control Using Generalized Advantage Estimation". In: *CoRR* abs/1506.02438.

David Silver, Thomas Hubert, Julian Schrittwieser, Ioannis Antonoglou, Matthew Lai, Arthur Guez, Marc Lanctot, Laurent Sifre, Dharshan Kumaran, Thore Graepel, Timothy P. Lillicrap, Karen Simonyan, and Demis Hassabis (2017a). "Mastering Chess and Shogi by Self-Play with a General Reinforcement Learning Algorithm". In: *CoRR* abs/1712.01815. arXiv: 1712.01815.

David Silver, Julian Schrittwieser, Karen Simonyan, Ioannis Antonoglou, Aja Huang, Arthur Guez, Thomas Hubert, Lucas Baker, Matthew Lai, Adrian Bolton, Yutian

Chen, Timothy Lillicrap, Fan Hui, Laurent Sifre, George van den Driessche, Thore Graepel, and Demis Hassabis (2017b). "Mastering the game of Go without human knowledge". In: *Nature* 550. Article.

Richard S. Sutton and Andrew G. Barto (2018). *Introduction to Reinforcement Learning*. 2nd. MIT Press.

Richard S. Sutton, David Mcallester, Satinder Singh, and Yishay Mansour (2000). "Policy gradient methods for reinforcement learning with function approximation". In: *In Advances in Neural Information Processing Systems 12*. MIT Press, pp. 1057–1063.

Ziyu Wang, Victor Bapst, Nicolas Heess, Volodymyr Mnih, Rémi Munos, Koray Kavukcuoglu, and Nando de Freitas (2016). "Sample Efficient Actor-Critic with Experience Replay". In: *CoRR* abs/1611.01224. arXiv: 1611.01224.

Ronald J. Williams (1992). "Simple statistical gradient-following algorithms for connectionist reinforcement learning". In: *Machine Learning* 8.3, pp. 229–256.

List of Figures