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 enabled machines to surpass human performance for Atari console games (Mnih et al., 2013), boardgames 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 outweights the isolated computational cost of training the agent, making sample efficient methods nessecary.

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) describes the reinforcement learning task as "learning what to do". Acting optimal within an unknown environment can be very difficult. The field within machine learning adressing this problem is called reinforcement learning.

The reinforcement problem consist 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* withing the *environment*, This simply means, we want to achieve the maximum possible *reward* within an episode or over time if an environment is continuous.

Environments can pose a range of tasks, with very different difficulty. The 'Cartpole'-environment for example requires 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 environment like the Atari 2600 games can often be costly to simulate.

ACER (Wang et al., 2016) provides a sample efficient learning 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.

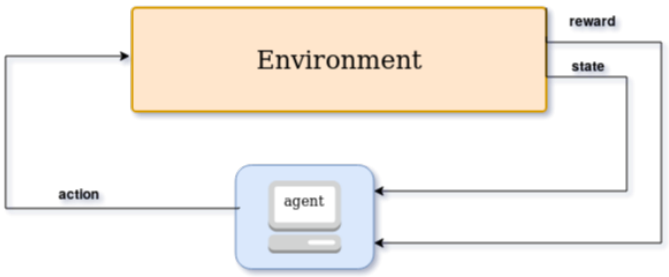


Figure 1: Agent interacting with the environment

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, we will always use s 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 behaviour the *agent* needs to make sure to explore the *environment*. Insufficient exploration can lead to suboptimal policies.

Whenever the agent interact with the environment, a*reward* and the next state are given to him 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 all terminate eather if the player has lost or won the game.

2.1 Elements of Reinforcement Learning

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

Policy

The behaviour 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 at time t under a policy π as $\pi(a \mid s_t)$ 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 reinfocement learning is the fact, that rewards are often very delayed. Connecting the delayed reward with the actions that caused it is a key task of reinforcement learning. Good examples are games like Pong or Breakout,

where the reward is caused by successfully playing a ball, but only received many frames later.

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 value of a state describes how much more reward can be earned from this state onwards. Values represent the sum of the future rewards, and indicate the long term desirability of states.

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

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

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

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

This is a good place to additionally define the advantage $A^{\pi}(s, a)$ of s given a is taken.

$$A(s, a) = Q(s, a) - V(s, a) \tag{3}$$

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

The sequential decision making process of the *agent* can be more formally described as a Markov decision process (MDP).

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

$S_0, A_0, R_0, S_1, A_1, R_1, S_2, A_2, R_2, \ldots, S_t, A_t, R_t, S_{t+1}$

Within this thesis a finite environment is assumed. We call a state *Markov* or say it has *Markov property* if it only depends on it's predecessor rather than the whole history.

$$P(s_{t+1} = s', r_{t+1} = r' \mid s_t, a_t, r_t, s_{t-1}, \ldots, r_1, a_0, s_0) = P(s_{t+1} = s', r_{t+1} = r' \mid s_t, a_t) \quad (4)$$

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 :

$$P^a_{ss'} = Pr(s_{t+1} = s' \mid s_t = s, a_t = a) \quad (5)$$

the reward function $R^a_{ss'}$, and the discount factor $\gamma \in [0, 1)$, used to define the importance of immediate reward in contrast to future reward.

2.3 Deep Reinforcement Learning

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

Usually a transformation or activation function is used on the input. Rectified linear units (ReLU), the tanh or the sigmoid function can be named as popular activaton functions. After feeding an input into the network and calculation 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 usually contain 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, enabeling reinforcement learning methods to be successfully applied to complex problems like those posed by the Atari 2600 console. (Li, 2017)

Within this thesis, we will only work with CNNs, consisting of convolutional layers, dense layers and ReLU activation functions.

We use convolutional data to extract features from images. By moving a filters over the image, the weight values of the filter are multiplied with the pixel data provided by the part of the image.

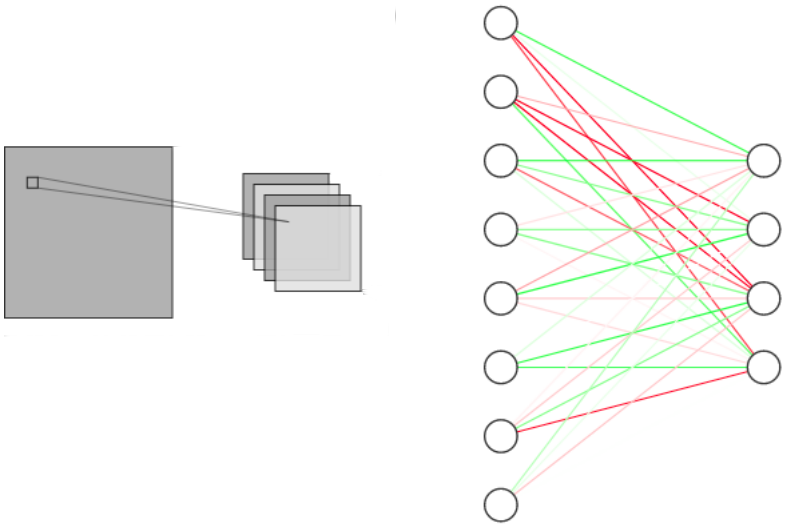


Figure 2: A convolutional (left) and fully connected (right) layer.

The output is a weighted sum of a part of the original image. The *stride* denotes how much a filter is shifted before being applied to the image again. Usually many filters are used, to receive many different features.

Fully Connected Layers are used to weight the input data and map it to the output space. Each 'Neuron' or Element in the output layer, hold a weighted sum of every input element.

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

$$ReLU(x) = \begin{cases} x & \text{if x >0} \\ 0 & else \end{cases} \tag{6}$$

3 Actor-Critic Methods

Many different approaches to different kind of reinforcement learning problems 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.

3.1 Monte-Carlo Predictions

We call a complete run of the environment from start to termination an *episode*. Monte-Carlo predictions learn from complete episodes. They can be used to estimate the value of a state by calculating the discounted returns for every encountered state and changing the estimated value of the states 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 describes *temporal difference* (TD) learning as one of the central ideas in reinforcement learning. TD learning combines dynamic programming and Monte-Carlo ideas to learn eather *state-values: V(s)* or *state-action values: Q(s, a)* Just like Monte-Carlo methods, TD-learning can learn directly from experience, while being able to learn without the need of finishing an episode through bootstrapping.

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

$$V(s_t) \leftarrow V(s_t) + \alpha[r_{t+1} + \gamma V(s_{t+1}) - V(s_t)] \tag{7}$$

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

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha[r_{t+1} + \gamma \max_a Q(s_{t+1}, a) - Q(s_t, a_t)] \tag{8}$$

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 by 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)

Initialize $Q(s, a)$ arbitrarily
Repeat (for each episode):
  Initialize s
  Repeat (for each step of episode):
    Choose a from s using policy derived from Q (e.g., ε-greedy)
    Take action a, observe r, s'
    $Q(s, a) \leftarrow Q(s, a) + \alpha\left[r + \gamma \max_{a'} Q(s', a') - Q(s, a)\right]$
    $s \leftarrow s'$;
  until s is terminal

Q-learning algorithm taken from (Sutton and Barto, 2018)

3.3 N-Step TD-Learning

The TD-Learning algorithm shown only learns from a single step. Often episodes and be long, and the final reward (e.g. winning or losing a game) can be very important. 1-step learning can take a very long time to propagate this final reward into earlier states. 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 'inf-step' TD update would be the same as a Monte-Carlo updates.

By combining the Monte-Carlo target

$$R_t = r_{t+1} + \gamma r_{t+2} + \gamma^2 r_{t+3} + \gamma^3 r_{t+4} + + \gamma^T - t - 1 r_T \tag{9}$$

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

$$V(s_t) = V(s_t) + \alpha(\gamma^n V(s_{t+n} + R_{t:t+n} - V(s_t) \tag{10}$$

where

$$R_{t:t+n} = r_{t+1} + \gamma r_{t+2} + \gamma^2 r_{t+3} + \cdots + \gamma^{n-1} r_{t+n} \tag{11}$$

denotes the discounted return for the next n steps.

3.4 Critic-Only Methods

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

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

Critic-olny methods provice 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

Onlike critic-only methods, actor only methods do not learn any state or state-action values. Instead they perform optimization directly on the policy. Usually a stochastic and parameterized policy π_θ is used.

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 seeks to maximize $J(\theta)$ through gradient ascent

$$\theta_{t+1} = \theta_t + \alpha\widehat{\nabla J(\theta_t)} \tag{12}$$

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.

$$J(\theta) \doteq v_{\pi_\theta}(s_0) \tag{13}$$

We can approximate the gradient of $J(\theta)$, $\nabla v_{\pi_\theta}(s_0)$ through

$$\nabla J(\theta) \propto \sum_s \mu(s) \sum_a q_\pi(s,a) \nabla_\theta \pi(a \mid s, \theta) \tag{14}$$

thanks to the policy gradient theorem. 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 simply rewrite this as the expectation under policy π:

$$\nabla J(\theta) \propto E_\pi \left[\sum_a q_\pi(s_t,a) \nabla_\theta \pi(a \mid s_t, \theta) \right] \tag{15}$$

Algorithm 1: REINFORCE provided by Sutton and Barto (2018)

Initialize parameters $\theta \; for \pi$
Repeat forever:
  Generate episode: $S_0, A_0, R_1, \ldots, S_{T-1}, A_{T-1}, R_T, following \; \pi$
  For each step t =0,1,... in T:
    G ←return from step t
    $\theta \leftarrow \theta + \alpha\gamma^t G \nabla_\theta ln(A_t \mid S_t, \theta)$

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 continuus 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 slowed down. (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 strenght 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 just like the actor-only approach.

The main reason for the variance in the gradient is the high variance of the return. By introducing a 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.

$$\nabla J(\theta) \propto \sum_s \mu(s) \left[\left(\sum_a q_\pi(s,a) - b(s) \right) \nabla_\theta \pi(a \mid s, \theta) \right] \tag{16}$$

The term $q_\pi(s_t, a) - b(s)$ can be replaced by different terms (Schulman et al., 2015). Popular choices for actor-critic learning are the advantage function A_π (3) or the TD residual

$$r_t + V_\pi(s_{t+1}) - V_\pi(s_t) \tag{17}$$

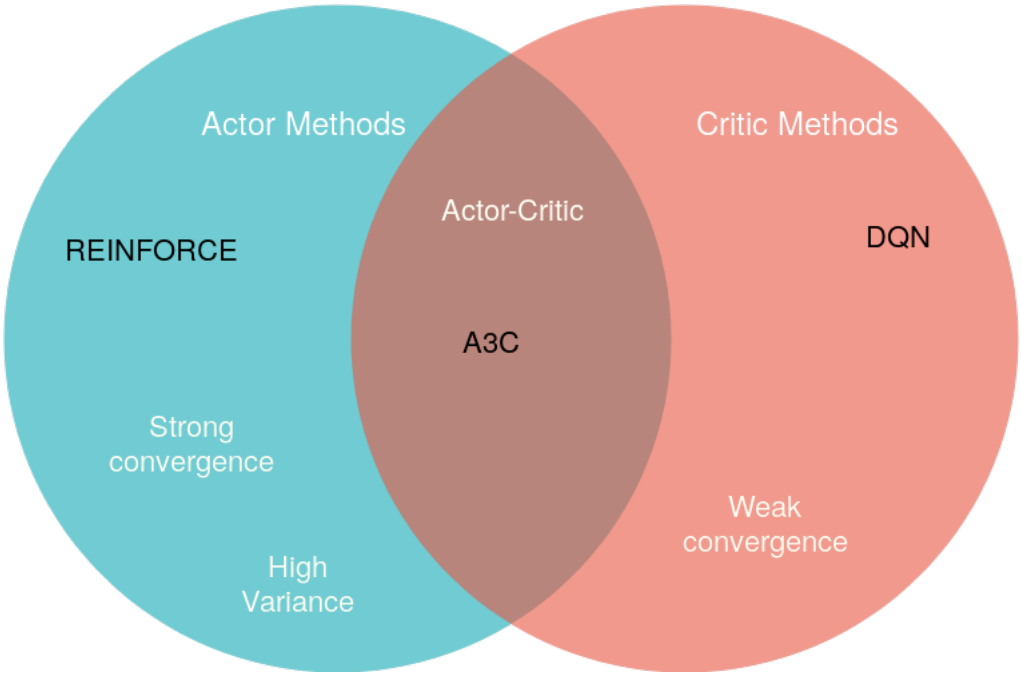


Figure 3: Actor, Critic, and Actor-Critic approach

3.7 Asynchronous Advantage Actor Critic (A3C)

In general we call an algorithm on policy, 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 non-stationary (Mnih et al., 2016). This can have a negative influence on the learning process.

Previous methods usually approached this problem 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 environment. 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 32 environments are used, to ensure the decorrelation of the samples.

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 an n-step update. The gradients are accumulated over all state withing the trajectory and applied in a single gradient step.

Algorithm 1: A3C algorithm by 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. θ : $d\theta^{global} \leftarrow d\theta^{global} + \nabla_\theta \log \pi_\theta(a_i|s_i)(R - V_{\theta_v}(s_i))$
    Accumulate gradients wrt. θ_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 $T > T_{max}$;

Figure 4: A3C by Mnih et al. (2016)

4 Off-Policy Learning

Off-policy methods use data sampled from a so called *behavior policy* we will denote as $\mu(a \mid s)$ to optimize the agents *current/target policy* π.

One benefit of off-policy learning is the possibility to chose a more exploratory behaviour 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. The class of Asynchronous Methods A3C belongs to is always slightly off-policy is only slightly off-policy, which doesn't impact the convergence.

If the behaviour policy can be very different from the target policy, the algorithm can no longer be viewed as *safe*. (Munos et al., 2016)

This problem is adressed by many different Methods, in order to ensure convergence, even for arbitraty "off-policyness" of sampled data.

4.1 Importance Sampling (IS)

One of the most basic ideas is to correct for the "off-policyness" by using Importance sampling. It is a classic 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

$$p_t = \frac{\pi(a_t \mid s_t)}{\mu(a_t \mid s_t)} \tag{18}$$

Even though this method can guarantee convergence (Munos et al., 2016), 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 imporance sampling by using the expectation and values under the target policy π of every untaken action. 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 unnessecarly 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, low varience with strong convergence was achieved while keeping the benefits of full returns.

Like TB(λ) the traces are safely cut in case off strong "off-policyness", without impacting the update too much, if the data was sampled under a behaviour policy μ close to the target policy π.

Retrace values for a q funcion are obtained recursively by

$$Q^{ret}(x_t, a_t) = r_t + \gamma\tilde{p}^{t+1}[Q^{ret}(x_{t+1}, a_{t+a}) - Q(x_{t+a}, a_{t+1})] + \gamma V(x_{t+1}) \tag{19}$$

with $\tilde{p} = min\{c, p_t\}$ being the truncated importance weight p_t (18).

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 contiuous and discrete action spaces.

ACER combines recent breakthroughs in the field of RL, by utilizing both the ressource 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.

The importance weighted policy gradient is given by:

$$\hat{g}^{imp} = \left(\prod_{t=0}^{k} p_t\right) \sum_{t=0}^{k} \left(\sum_{i=0}^{k} \gamma^i r_{t+i}\right) \nabla_\theta \, log \, \pi(a_t \mid s_t) \tag{20}$$

The unbounded importance weights can cause massive variance. Degris et al. (2012) approached this problem by approximating the policy gradient as

$$g^{marg} = E_{s_t \sim \beta, a_t \sim \mu} \left[p_t \nabla_\theta log \pi_\theta(a_t \mid s_t) Q^\pi(s_t, a_t) \right] \tag{21}$$

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

To compute this gradient, knowledge about Q^π is nessecary. The retrace values (19) present a good estimation for Q^π.

To further reduce the variance, the importance weights are truncated. A core problem of actor-critic methods is the tradeoff between bias and variance.

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 21 with their truncated terms = min { c,p_t}, $wherecdenotedthetruncationvalue.$

The introduced bias correction term is given as:

$$E_{x_t} \left[E_{a \sim \pi} \left(\left[\frac{p_t(a) - c}{p_t(a)} \right]_+ \nabla_\theta log \pi_\theta(a \mid s_t) Q_\pi(s_t, a) \right) \right] \tag{22}$$

To receive the ACER policy gradient we simply 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 simply using the estimated Q-function. Q_π in the correction term is simply estimated by our critic.

Finally we use the approximation of the value function, which is simply 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:

$$\begin{aligned} g_t^{ACER} &= \tilde{p}_t \nabla_\theta log \pi_\theta(a_t \mid s_t) \left[Q^{ret}(s_t, a_t) - V_{\theta v}(s_t) \right] \\ +E_{a \sim \pi} &\left(\left[\frac{p_t(a) - c}{p_t(a)} \right]_+ \nabla_\theta log \pi_\theta(a \mid s_t) \left[Q_{\theta v}(s_t, a) - V_{\theta v}(s_t) \right] \right) \end{aligned} \tag{23}$$

Note that the expectation over states is no longer necessary, because it is approximated by the 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:

$$(Q^{ret}(s_t, a_t) - Q_{\theta_v}(s_t, a_t))^2 \tag{24}$$

and has the standard gradient:

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

An entropy term of the policy π is added to the loss function. Using entropy encourages exploration. Adding the entropy to the loss has shown to prevent early convergence to suboptimal policies.

The Entropy for a policy π is given by:

$$H(\pi) = -\sum_{i=1}^{A} \pi(a_i) \log_b \pi(a_i) \tag{26}$$

where b denotes the basis for the logarithm. A common choice for the base are 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 doesnt 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 it's 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 ressources, the trust region optimization was not used.

5.1 Experimental Setup

5.1.1 Environment

All experiments were made using multiple selected games from the Atari 2600 console game environments providede 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 "superhuman" performance.

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

Scoring well on this environment can be considered a great achievement.

Space Invaders is a fast paced game. Human and reinforcement learning approaches usually have similar scores. (Mnih et al., 2013)

The hyperparameters were adjusted to ensure a good performance for the Breakout environment. Test 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, enabeling a faster learning process.

5.1.2 Preprocessing

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

The playing area is grayscaled and resized the frame to a. 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

$$G_{Luminance} \leftarrow 0.3R + 0.59G + 0.11B, \tag{27}$$

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

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.

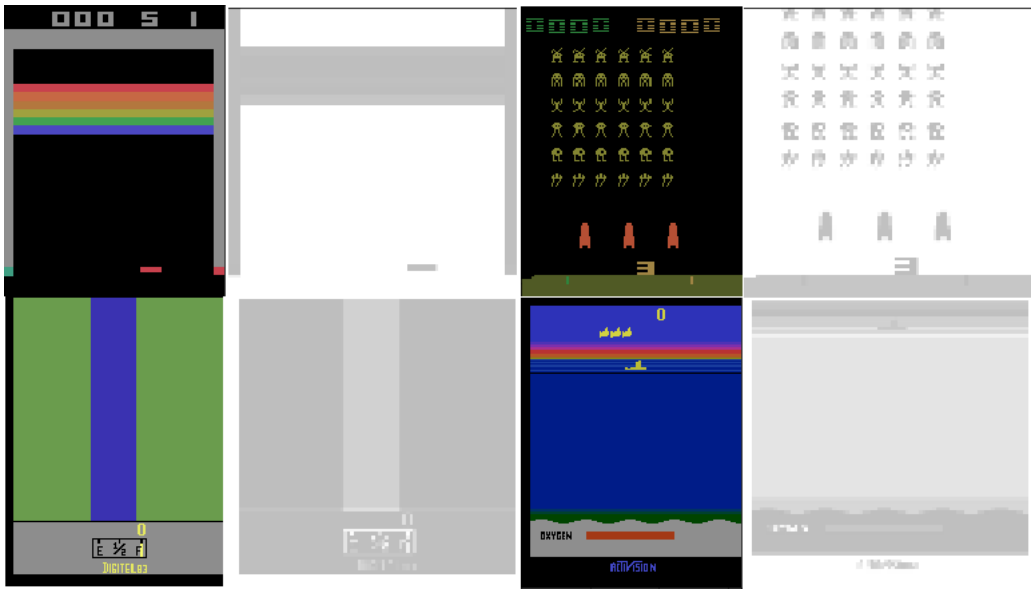


Figure 5: The Atari 2600 console games Breakout, SpaceInvaders, Riverraid and Seaquest before and after being processed.

To receive the initial state, we simply stack the first frame 4 times.

This can convey a feeling of movement to the agent.

A high variances within the rewards 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 eather -1, 0 or 1. This method was adapted 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 paper uses the architecture proposed in the nature paper (Mnih et al., 2015), all further experiments were done using this architecture.

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

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. Even though other activation/transformation functions exist, the ReLU has empirically done really well.

A test run using tanh instead of ReLU caused a strong loss of performance.

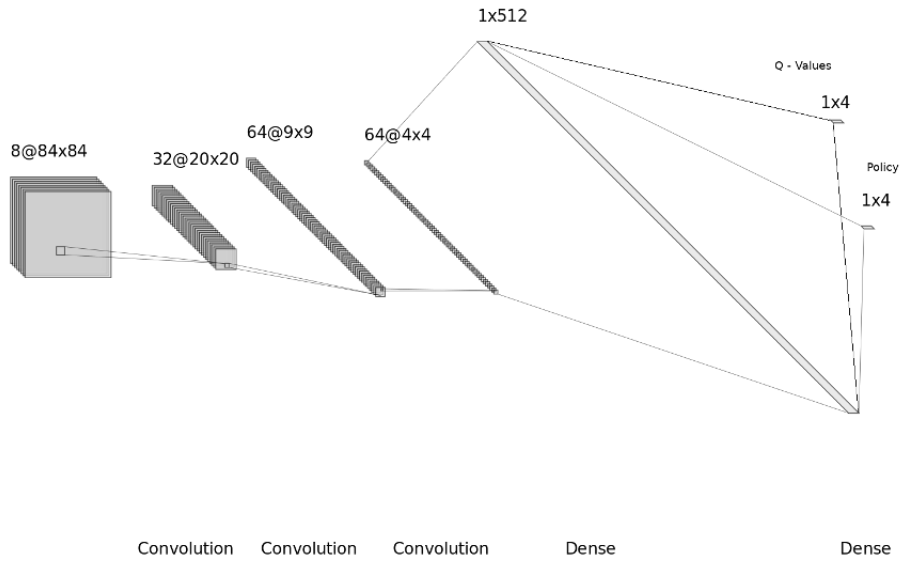


Figure 6: Shared network for policy and Q-Value approximization for an environment with 4 possible actions

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

$$P(A = a'|s) = \frac{e^{z_\theta(a')}}{\sum_{i=0}^{N} e^{z_\theta(a_i)}} \tag{28}$$

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

within our

The agent is trained by using 16 learner threads running on the CPU. Each Thread has it's 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.

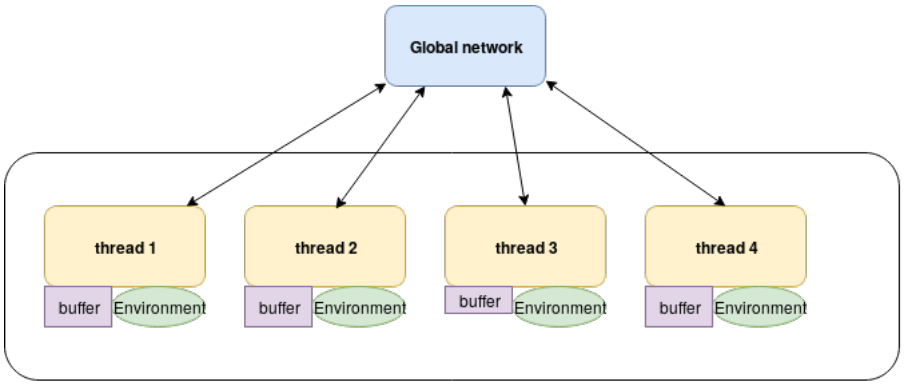
Updates to the network were performed every 20 steps or if the environment reached a terminal state.

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.

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



Worker Threads simulate or sample environments, calculate the gradient and update the global network asynchronous. Workers update their local parameters by copying the global ones every episode.

by:

$$g = \alpha g + (1 - \alpha)\delta\theta^2 \tag{29}$$

$$\theta \leftarrow \theta - \eta \frac{\delta\theta}{\sqrt{g + \epsilon}} \tag{30}$$

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 Asynchrounous Actor critic Methods. (Mnih et al., 2016)

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

Since RMSPropOptimizer in tensorflow, which was used for the implementation, can only use gradient decent, the following Loss-Function was minimized. Instead of having seperate updates for policy, value function and entropy, a single loss is formed and optmized.

$$L_{ACER} = -L_{Policy} + w_q L_Q + \beta Entropy \tag{31}$$

with L denoting the loss. w_q and β are hyperparameters used to decide how much influece 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 said to be else, 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	weigth of entropy in the loss function
w_q	0.25	weigth 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 is the received and capped reward in comparison to the actual reward given by the environment we will take a look at afterward. The decision to use the capped rewards was made due to their lower variance, and because these rewards were used when optimizing the function.

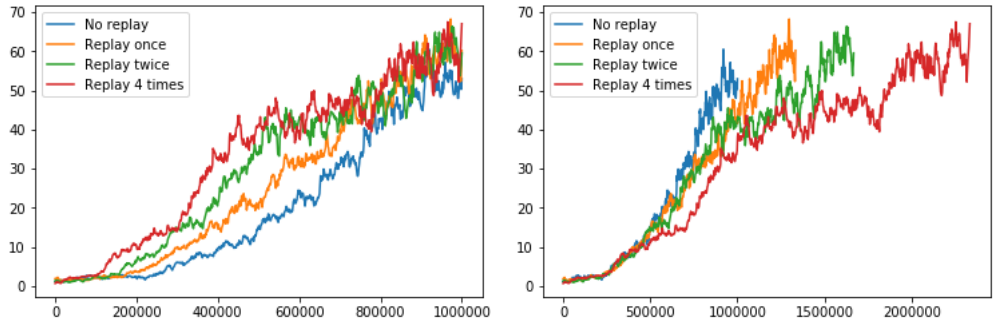


Figure 7: ACER results on the Breakout environment in regards to sample efficiency (left) and computation time(right).

Breakout

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. Instead of looking at actual time, we will instead consider the relation towards the online steps, since this does not depend on the performance of the machine. 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) offered better performance in terms of sample efficiency, but could not match the learning speed of the online agent.

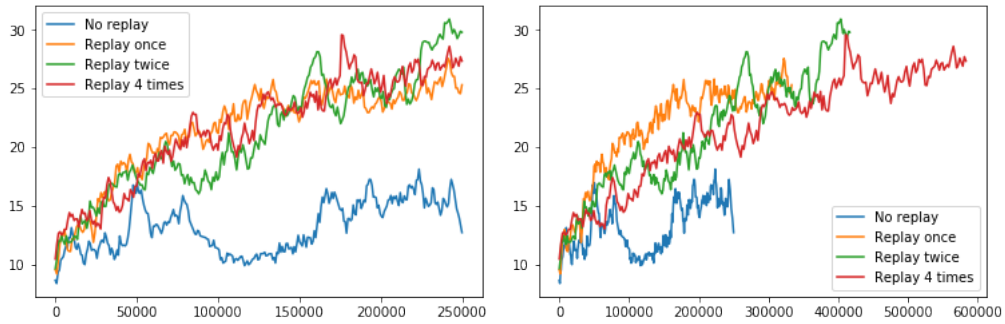


Figure 8: Results on SpaceInvaders in terms of sample effciency (left) and comutation time(right)

Space Invaders

The poor performance of the online variant on SpaceInvaders came as a big surprise. Assuming this performance to be caused by e.g. a very unlucky initialization, the same

test was used 2 more times, without achieving better results. In comparison the agents using replay had a much smoother performance curve. All replay ratios showed similar sample efficiency and.

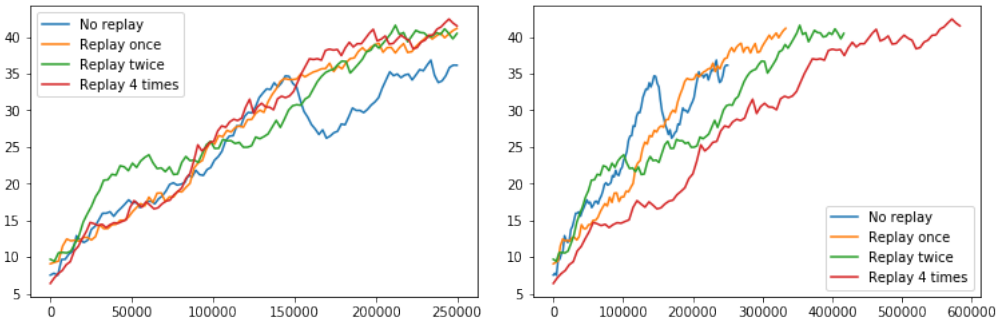


Figure 9: ACER results on the Breakout environment in regards to sample efficiency (left) and computation time(right).

Seaquest

Similar to the Results in Breakout, we can see that the replay agents are more sample efficient. We should note, that all replay agents, no matter the ratio learn relatively even and start to hit a cap in the end. Using a replay ratio of 1 showed the benefits of a smoothed learning process in comparison to the online agent, while achieving similar results in terms of processing time which is however mainly caused by the drop in performance for the online agent, which is similar to the results on Spaceinvaders.

Even though the main purpose of ACER is to provide a sample efficient algorithm, it additionally seems to smoothen the learning process.

7 Preprocessing Atari Environments

Atari Environments were specifically designed for Humans to be able to play them. Very important information might be conveyed through text messaged or symbols to a human, but might be heavily underpresented within the pixel data.

Good examples for this problem are environments like Sequest or Montezuma's revenge.

Seaquest often gets learning agents stuck on local maxima, because unlike a human they do not refill the oxygen. A human player managed to outperform DQN (Mnih et al., 2015) gaining 4x the reward on average for Seaquest. In Montezumas Revenge, DQN did not manage to receive any reward at all. This is in lign with the observations within this thesis. Figure 9 already shows the agents approaching a cap 40.

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 can not link the oxygen bar depleting to the end of the game. An attempt to increace the influence of the oxygen bar on the actions, could be to simply to use space not relevant to the game to increase the size of the bar, therefore increasing it's 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 10: Seaquest environment after normal (left) and individual (right) preprocessing

By applying a minimal individual preprocessing to the Seaquest environment, and slightly adjusted 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.

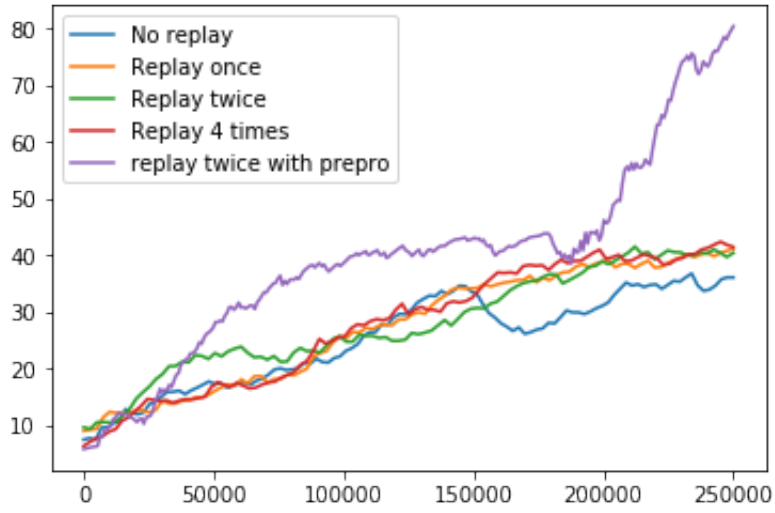


Figure 11: Results of preprocessed Environment

—- Placeholder: Comparison to normal seaquest environment with changed hyperparameters. —-

8 Adversial Attacks

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.

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

List of Tables