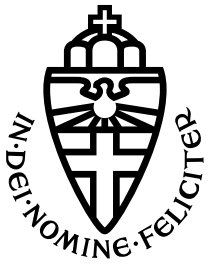
RADBOUD UNIVERSITY NIJMEGEN



FACULTY OF SOCIAL SCIENCE

IMPLEMENTING AND TESTING THE A3C ALGORITHM IN A NOVEL ENVIRONMENT

NEURAL INFORMATION PROCESSING SYSTEMS
- REINFORCEMENT LEARNING -

Daniel Anthes	**Nicole Jansen**	**Moritz Nipshagen**	**Jan Zerfowski**	**Marius Kästingschäfer**
s4767799	s1046842	s1044009	s1044709	s1047319

October 29, 2021

ABSTRACT

In the growing field of reinforcement learning, so called "Actor-Critic" models are being used with great success. In this work, we implemented a type of this model called Asynchronous Advantage Actor Critic (A3C). A3C has several training instances running in parallel but still enables sharing a network architecture, weights and weight updates across agents. We implemented A3C as well as a preprocessing and evaluation pipeline in the PyTorch framework. The algorithm in our work is applied in a novel environment, developed by the AI department at Radboud University. During training and test time, the agent is provided with high-level sensory inputs only. Additional to the implementation details and our results we provide some insight into the biological plausibility of the A3C algorithm. We found that our implementation of the algorithm effectively learned OpenAI Gym's CartPole on parameter inputs. We implemented an image preprocessing pipeline and an autoencoder structure. However, the complexity of the pixel-inputs was too high to be effectively learned by our implementation with convolutional layers.

***K*eywords** Reinforcement Learning · Neurosmash · Asynchronous Advantage Actor Critic · A3C · Biological Plausibility

1 Introduction

In the growing field of reinforcement learning, so called "Actor-Critic" [1] models are being used with great success. Actor-Critic methods aim to make use of the advantages of value-based and policy-based methods. However, when using only one critic and one actor, this process is inherently unstable due to the strong correlation of weight updates. A solution to this problem was introduced with the Asynchronous Advantage Actor-Critic agent (A3C) [2]. This particular

model has several training instances running asynchronously in parallel, while sharing a network used for evaluation. This enables a broader coverage of the state-action space and has a stabilizing effect on the learning process. A3C has been shown to converge faster than conventional algorithms, with comparable accuracy [2]. This approach is useful for training on a CPU with many cores or a distributed system with lacking graphical computing power. In the current work, we implement the A3C algorithm, as well as a preprocessing and an evaluation pipeline in the PyTorch [3] framework to run in a new environment. During training and testing, the agents are provided with high-level sensory inputs only (i.e., pixel-data). This novel environment, called "Neurosmash" is a two player boxing game implemented in the Unity Framework [4]. The environment together with the interface was developed by the AI department at Radboud University.

Our work shows a successful implementation of the A3C algorithm with the OpenAI Gym CartPole problem. However, we did not succeed in transferring our results to learning from pixel values. This made it impossible to solve the Neurosmash environment. However, this report presents our working process and gives several details of our implementation. Additionally, we provide grounding for the biological and psychological plausibility of reinforcement learning in general and actor-critic methods in particular.

2 Related Work

Within the last decade, the research in reinforcement learning (RL) has tremendously increased [5]. The areas RL intersects with, now include games [6], robotics [7], real world applications and products [8]. Several different learning strategies are currently distinguished and applied. A broad overview can be seen in Figure 1. A wide-ranging differentiation that can be made for algorithms is between model-free and model-based reinforcement learning algorithms. Model-free refers to a group of algorithms that do not rely on transition probability distributions to represent the statistical regularities of the environment. Model-based algorithms instead are able to generate a predictive model of the environment. In the following, we discuss only model-free algorithms and evolutionary strategies. A3C and ACKTR are two kinds of policy optimization methods, while Deep-Q-Networks (DQN) are a Q-learning method. Further, we justify our choice for the A3C algorithm with biological and computational arguments.

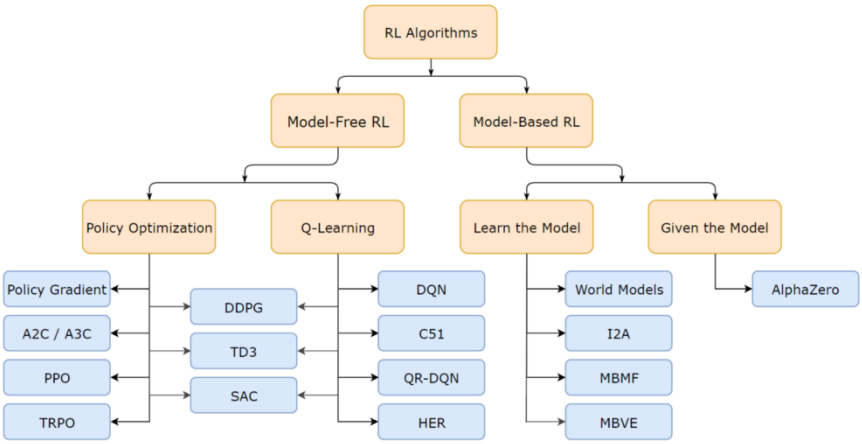


Figure 1: **Taxonomy of RL algorithms** We describe DQNs, A3Cs, ACKTR and evolutionary strategies. Evolutionary strategies are not considered to be canoncical RL methods. The taxonomy of algorithms used in reinforcement learning, adapted from [9].

Deep Q-Networks (DQN) Dynamic programming (DP) is a method for simplifying a complicated problem by dividing it into smaller sub-problems that are less complicated. The key concept in DP is a value function that assigns a value to each state, which corresponds to the expected reward if the system was started from that state [1]. If transitions between states are stochastic, the probabilities need to be known in order to calculate the value function. However, these transition probabilities are not always available for all systems. Q-learning, first described by Watkins in 1989 [10], circumvents this problem. Q-learning could be viewed as both a method of asynchronous DP and a form of model-free reinforcement learning. Q-learning is a method to approximate the expected reward of a specific action in a state, also called action-value function [1, 11]. If the state-space is very large or unknown, it is infeasible to save an expected reward for each state. To work around this issue, neural networks can be used to approximate the Q-function. These networks are then referred to as Q-networks [11].
Deep Q-learning, a variant of Q-learning, was first introduced in 2013 by DeepMind [11, 12]. Here, a deep convolutional

neural network is used to approximate the optimal action-value function, which is called a deep Q-network (DQN) [12]. Although Q-learning has shown to converge in some cases [13], it does not converge in very large or complex problems [12, 11]. Deep Q-learning addresses these instabilities. However, when a network is trained with subsequent state-action pairs as occurring in a game, the correlations between the states usually lead to unstable learning because of oscillating parameters and overfitting. These problems can be avoided, using a technique known as experience replay, where the agent's experience at each time step is stored into a replay memory, including state-action pairs and the received reward. During learning, the updates are applied using random samples or minibatches from the replay memory [11]. The agent selects and executes an action according to an ϵ-greedy policy [11, 12]. To improve learning in noisy environments and avoid overestimation of action values, two separate networks can be maintained where parameters are exchanged only periodically [12].

Asynchronous Advantage Actor Critic (A3C) Actor-critic algorithms are algorithms that have two learning units: an actor and a critic. An actor has adjustable parameters and makes decisions. A critic is a function approximator, which tries to approximate the value function of the policy that is used by the actor. Thus, both the actor and the critic interact with each other [1].
One example of an actor-critic algorithm is the asynchronous advantage actor-critic (A3C), which was first introduced in 2016 [2]. The advantage ($A(s_t, a_t)$) of action a_t in state s_t can be derived as $A(s_t, a_t) = Q(s_t, a_t) - V(s_t)$, with $Q(s_t, a_t)$ being the Q-value function and $V(s_t)$ being the value-function [2]. The Q-Value function estimates the expected future reward from a given state s_t if action a_t is taken. The value function gives a general approximation of the "goodness" of the state. Using the advantage function has the benefit of reducing variance and stabilizing the model. A3C is asynchronous by having multiple independent agents trained in parallel. Before a new episode is started, each agent copies the weights from the shared network. After a given amount of steps the agent shares its gradients with the shared network and performs a backwards pass. Using an asynchronous algorithm has the advantage of decorrelating the agents' data into a more stationary process, due to the fact that different states will be experienced by different agents at the same time. This is similar to using a replay buffer (see above), but heavily decreases the memory demand because the state-action pairs do not have to be saved. Another benefit is that no specialized hardware is needed and it could be run on a standard machine with a standard multi-core CPU [2]. Learning multiple parallel agents is also assumed to reduce training time. Furthermore, Mnih et al. showed that A3C outperformed comparable algorithms [2]. Finally, it is possible to use a different exploration-exploitation parameter (epsilon) to increase diversity between the agents. In Appendix 6.2 a pseudocode for A3C is attached, adapted from [2].
A synchronous, deterministic variant of A3C, advantage actor-critic (A2C) was proposed later. This variant was created after AI researches wondered if the asynchrony really led to an improvement in performance, or if the improvement was caused by an implementation detail allowing faster training using CPU [14]. A2C is different from A3C in that after training for the given amount of steps, each agent would wait until all agents were finished before updating the global parameters. A2C has been shown to outperform A3C [14], so this provided no evidence that asynchrony actually provides any performance benefits.

Actor-Critic using Kronecker-Factored Trust Region (ACKTR) Another actor-critic algorithm is the actor-critic using Kronecker-factored trust region (ACKTR) and was introduced by researchers from the University of Toronto and the New York University in 2017 [15]. ACKTR is a combination of actor-critic methods, trust region optimization, and distributed Kronecker factorization. Kronecker-factored approximate curvature (K-FAC) is a method for approximating natural gradient descent in neural networks, proposed by Martens and Grosse in 2015 [16]. The approximation of natural gradient updates is achieved using Kronecker-factored approximation to the Fisher matrix [15]. The trust region optimization means that updates are scaled down to modify the policy distribution by at most a specified amount. Using the ACKTR method, both the actor and the critic will be optimized using the K-FAC with trust region.
ACKTR has shown to perform better than other algorithms including A2C. Compared to A2C, ACKTR needed far less episodes to reach human level performance [15]. Furthermore, ACKTR has a better sample complexity compared to methods such as A2C and although the computation needed for ACKTR is more expensive than that for a standard gradient update, it is less expensive compared to Hessian-free optimization methods [14, 15].

Evolutionary strategies (ES) Evolutionary strategies (ES) belong to the group of black box optimization algorithms [17]. The strategy is regarded as a metaheuristic optimization algorithm since it is not based on optimizing individuals only, but on improving a population-based fitness function. The general idea of those algorithms is roughly inspired by mechanisms from natural evolution such as reproduction, mutation, recombination and especially selection. Together with genetic algorithms (GA), evolutionary strategies belong to the realm of evolutionary computing (EC). Since they do not use any (possibly biased) knowledge about the underlying fitness landscape they can be applied to a wide variety of problems. ES was shown to be a viable alternative to Q-learning and policy gradients in 2017 by OpenAI [18]. Additionally, evolutionary methods have the advantage of being easy to parallelize by evaluating fitness across multiple machines [19].

Justification of our choice We have implemented the A3C algorithm on the new environment. The choice for using the A3C algorithm is based on multiple benefits this algorithm provides. These benefits were already mentioned above, so we will summarize them here. The first benefit of using the A3C algorithm is that it has shown to outperform other comparable algorithms. Second, using the advantage function helps to reduce high variance and stabilizing the value-function. Third, because multiple independent agents are trained in parallel (instead of using a memory-consuming replay buffer), the A3C method decorrelates the agents' data to facilitate a more stable training process. Additionally, the algorithm can be run on a standard machine with a multi-core CPU [2].
We chose A3C over A2C because the algorithm and paper were more accessible and we found the available tests and implementation to be better documented. We are aware of the fact that ACKTR outperformed A3C on nearly all games with either discrete or continuous control. However, we found A3C simpler to implement, biologically more plausible and as such overall better suited for the task at hand. Further, we regarded it as an interesting challenge to implement a less common algorithm, rather than following more popular approaches to reinforcement learning problems.

3 Neurobiological grounding

Animals and especially Homo sapiens are able to weigh short term and long term benefits against each other. They are able to leverage knowledge and experiences for reward-maximization and punishment-minimization. A similar capability in machines would be an outstanding achievement. This is one of the many reasons why the use of neuroscientific and biological models was, and still is, an important source of inspiration for artificial intelligence research [20].

3.1 General inspiration

Reinforcement learning is a method to teach an agent how to maximize a numerical reward signal [21]. However, reward signals, predictions and control are not only relevant concepts in the field of reinforcement learning, but also play a vital role in psychology, neuroscience and biology. In psychology, the terms are associated with learning during classical or Pavlovian conditioning. In several animal studies it was shown that learning by trial and error is an essential part of goal-directed behavioral control. In neuroscience, approaches to explain control behavior and learning involve dopamine, a neurotransmitter strongly involved in reward processing. Cognitive neuroscience also abstracted several abilities from human agents that might be necessary for artificial agents to be human-like. The most important among them is remembering and predicting [22]. Both can be seen as two sides of the same coin. Having a compressed version of the environmental dynamics enables the agent to anticipate the future and respond in a meaningful manner. Ha and Schmidhuber utilized this idea to develop an agent able to learn relevant behaviour patterns inside its own world model [23].

3.2 A3C inspiration

To create the A3C algorithm, asynchrony was added to the original actor-critic approach. Asynchrony was added to stabilize the algorithm, however the original paper does not mention basing this decision on any (neuro-)biological processes [2]. Nevertheless, similarities between actor-critics and the structure and/or functioning of certain (networks of) brain areas have been pointed out [24]. In particular, actor-critics have been compared to the basal ganglia, which has led to the creation of actor-critic models of the basal ganglia [24, 25]. The basal ganglia refers to a group of subcortical areas in the brain, which play an important role in the organization and planning of movement and behavior [26]. To be more precise, the actor and critic components of the actor-critic algorithm have been compared to a subgroup of subcortical areas called the striatum. The dorsal subdivision of the striatum is suggested to function comparable to an actor and the ventral subdivision of the striatum is suggested to function comparable to a critic. Both the dorsal and ventral subdivision are critical for reward-based learning [21]. Another way the basal ganglia have been compared to the actor-critic algorithm is through similarities between the temporal difference (TD) error and the function of dopamine in the brain [21]. Dopamine is a neurotransmitter that plays a vital role in reward processing, which includes reward-based learning. Dopaminergic neurons exhibit phasic responses, which signal reward prediction errors, and correspond to the TD error. Similarities could be observed, for example, when a reward arrives later than expected. The reward is then unexpected and will generate a positive prediction error, which is the case for both the TD errors and responses of dopaminergic neurons [21]. Furthermore, the fact that in actor-critic algorithms the TD error is the reinforcement signal for both the actor and the critic is comparable with the fact that dopaminergic neurons target both the dorsal and ventral subdivisions of the striatum [21]. For a more in depth comparison between TD error and dopamine, we refer the reader to [21]. Even with some discrepancies between the TD error and response of dopaminergic neurons, the comparison between the two has helped improve understanding the workings of the brain's reward system [21]. An actor-critic network created by Barto et al. in 1983 was inspired by the 'hedonistic neuron' hypothesis. This hypothesis

states that individual neurons will adjust the efficacies of their synapses in order to maximize the difference between rewarding and punishing synaptic input [21]. Furthermore, the A3C algorithm, which was trained to solve a challenging cognitive task, has been shown to have a similar learning trajectory compared to human subjects [27]. The authors argue that new insights into cognitive processes could be gained by comparing such neural networks to behavior and neural responses from humans [27]. Others have adapted the standard model of reward-driven learning to a model that additionally explains observations which could not be explained by the standard model. This proposed model was based on actor-critic models and both A2C and A3C were used for simulations which showed the proposed model accounts for a diverse range of observations [28].

4 Methods

To facilitate an efficient division of tasks, we wrote an environment wrapper that enabled modularization of tasks. Further, we built an interface that enabled us to write the actual agent code agnostic of the environment. For testing our implementation, we used Open AI Gym's CartPole [29] and the Neurosmash [4] environment. We used PyTorch [3] for implementing and training the networks. Github was used for version control and to make collaboration as easy as possible.

4.1 Neurosmash Environment

We used the Neurosmash environment based on the Unity development platform-based for assessing the performance of the proposed algorithms. Neurosmash was written by U. Güçlü (2019) and specifically designed for the SOW-MKI49-2019-SEM1-V: NeurIPS course at Radboud University [4]. Neurosmash consists of two agents (red and blue), the blue agent is controlled by a naive implementation of the A*-algorithm. The red agent is controlled by the player, i.e., our network trained with the A3C algorithm. A picture of the environment can be seen in Figure 2. Both agents continuously run forward and their action space is limited to turning left or right. The game ends when one of the agent falls (or is pushed) off the platform. The reward structure of the game can be considered sparse. The player remaining on the platform receives a reward of 10 only at the end of the game, while the other one gets 0 reward. We set the environmental speed between 1 and 5, since we found higher values to result in glitches, where the agent 'jumps' outside the platform without ever losing the game. The ports of each environment framework were set manually in increasing order to be connected automatically to our environment wrapper. The image size was set to 64x64 pixels to decrease the learning time needed to acquire useful representations. We found this size large enough to discriminate the agents, but small enough in terms of number of parameters to make training the networks feasible.

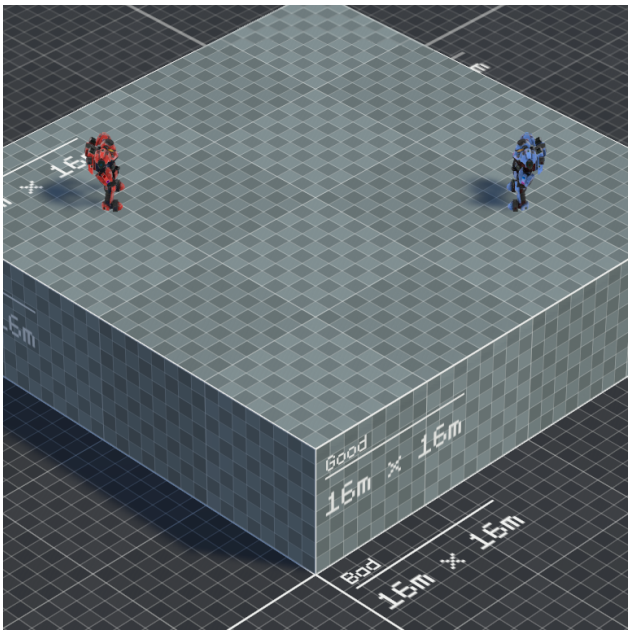


Figure 2: **Neurosmash environment** The two players (red and blue) are shown in their initial position. The walkable area is determined by the square. If the agent is outside the accessible area the agent loses the game.

4.2 Image Preprocessing

4.2.1 Screen Manipulation

To extract and use the state of the environment, we built an image preprocessing pipeline. Since the game field is viewed from an angled perspective above the ground, the first step was to transform it in such a way, that the field is represented as a flat square as if viewed from above. Since three corners are not in the visible field, the transformed square had three black corners as can be seen in Figure 4. The square was then up- or down scaled to a given size for further processing. Other manipulations that we implemented and considered were color normalization and gray-scale to increase visibility and decrease size of the image. Normalization about the three colour channels was implemented by dividing each pixel's channel by the sum of its color values.
For the conversion of the image to gray-scale, weights for each colour were manually determined to maximise contrast. Colour channel (RGB) values of 0.35, 0.4 and 0.25 were determined by trial-and-error to be optimal. The resulting gray-scale image can be seen in Figure 4. This manipulated screen was then passed to the first convolutional layer of the agent pipeline.

4.2.2 Autoencoder

We also implemented an autoencoder in our pipeline [30]. For that purpose, we used a random agent to generate over a million states of the game. All states were subsequently processed as described above to create a $64x64x3$ image only featuring the game board and the two agents. These images were then fed into an autoencoder to reduce the dimensionality of the image into a feature vector of length 128. The input was first run through a batch normalization layer, followed by two SeLU activated convolutional layers. Those feature maps where then max-pooled and run through another convolution before being flattened and brought down to size 128 with a fully connected layer. The structure can be seen in Figure 3. A decoder to reconstruct the input image from the vector was written exactly mirroring the autoencoder structure. The feature vector was first increased in size by a fully connected layer to size 1024, then reshaped to be fed into a deconvolutional layer. Since max-pooling cannot be undone, but only approximated, the PyTorch built in inverse function was used, using information from the max-pooling layer of the encoder to reconstruct. The data was then further run through two deconvolutional layers, which resulted again in a $64x64x3$ image.
In training, the input image was encoded using the structure described above and then decoded again by the decoder. The output of the decoder was finally compared to the actual input, the mean-squared-error was calculated as a loss function to propagate the changes through decoder and encoder. The network was trained for 4 epochs with a batch size of 256 and a learning rate of 0.005

While the loss saturated at around $2 * 10^{-3}$, the output images of the decoder were still unintelligible and noisy. This led us to the conclusion that the network learned insufficiently to reduce the information into the feature vector. Since trials with the autoencoder network put before the controller network were also unsuccessful, we did not consider our implementation of the autoencoder structure in following trials.

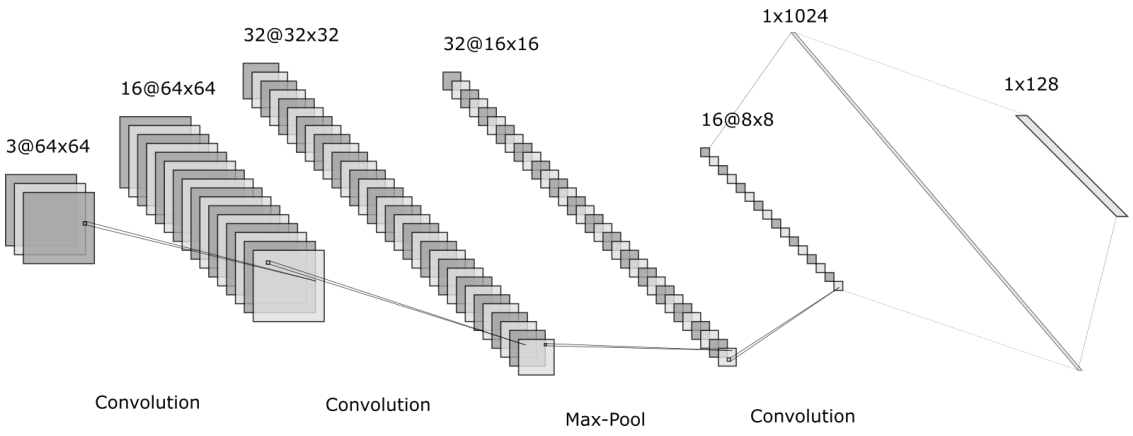


Figure 3: **The Autoencoder Structure** A batch normalization layer was set before the first convolution and before the 128 output layer. Each convolutional layer uses a SeLU as an activation function, whereas the linear layers were linearly activated. Here, only the encoder network is shown because the decoder network is an exact copy of the encoder.

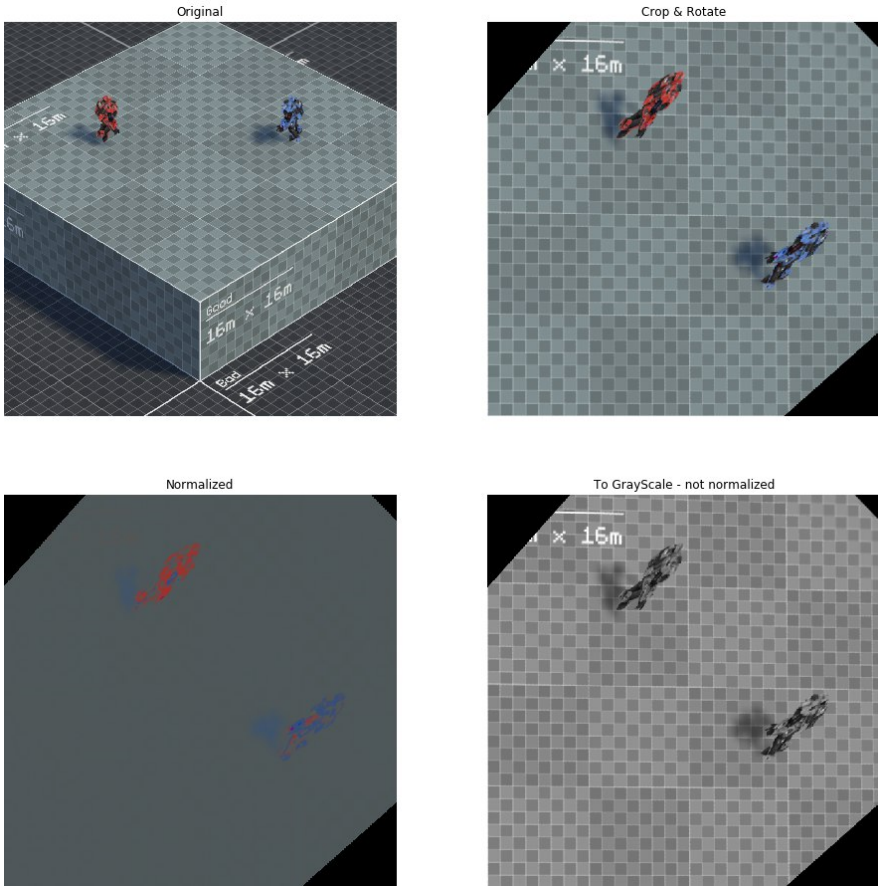


Figure 4: **Preprocessing of the image** The image in the upper left shows the unprocessed input. The image in the upper right is cropped and rotated. The lower left image is normalized. The lower right image is grayscaled.

4.3 Training procedure and agents

During the training process, we mainly struggled to find the right hyper-parameters. We started with the parameter settings we found in the original A3C paper [2]. Since we did not succeed with these settings, we started with exchanging the convolution module with a slightly different architecture. However, it turned out be non-trivial to train the combined network. We investigated the ongoing process using tensorboard, first by looking at the loss values for the policy, the value network and the reward per agent. Later, we also plotted the weights and gradients of the convolutions layers to see if our network was suffering from vanishing gradients. We used between six and up to twenty agents, which strongly decreased the overall training time.

The different hyperparameters we implemented and changed are listed in the Appendix 6.3. We tried out several combinations of all kinds of settings.

4.4 Scaling and parallelizing

To make full use of the advantages of the parallel and asynchronous capabilities of the A3C algorithm, we used the cluster from the Radboud AI department. The setup we trained on consisted of an AMD Ryzen Threadripper 2990WX with a 32-Core Processor, 128 GB ram and 3 NVIDIA GeForce RTX 2080 Ti, 3x 12 GB VRAM. This enabled us to make fast loss evaluations on multiple instances, train the convolutional part of the network efficiently and run several instances of the environment.

4.5 Algorithm implementation

The main rationale behind the A3C algorithm is that its asynchronous nature makes it possible to utilize multi-core CPUs for training. Additionally, in our case the bottleneck for training speed of the algorithm is the duration of

single games and not necessarily the computational demands of the algorithm itself. The asynchronous nature of A3C solves this problem since it allows us to run as many simulations in parallel as are needed to make optimal use of the resources we have at our disposal. To this end, we implemented the A3C algorithm as described in Mnih et al. [2]. Our implementation is written in python and makes use of the PyTorch framework [3].

4.5.1 Structure

The core of our implementation consists of an A3C class that maintains the shared network parameters that are being trained. During training, the A3C instance spawns a number of Worker instances, each of which run independently of each other in their own process such that the training can be distributed over multiple CPU cores. Each Worker instance maintains its own copy of the network as well as its own instance of the environment. Workers only need to communicate with the main A3C instance for two purposes: First, at the start of each episode the Worker synchronizes its weights with the shared network by replacing its own weights with those of the shared network. Second, after each backward pass the Worker pushes the computed gradients to the A3C instance. The A3C then uses these gradients to update the shared network. It is worth noting that weight optimization only takes place in the A3C instance. Further, gradient updates are made asynchronously and without acquiring a lock. As a result in principle it is possible that Workers override each others updates to the A3C instance. However, previous work on the Hogwild! algorithm has shown that this lock free approach still outperforms a synchronized approach [31].

4.5.2 Loss Function

We use the loss functions proposed in the original paper on A3C, namely [2]. The pseudocode can be found in figure 6.2 in the appendix below.

4.5.3 Data Efficiency

While it is easy to generate large amounts of training data when training A3C by training on multiple environments we suspect that the sample efficiency of A3C is rather low. In contrast to many implementations of DQN, A3C does not make use of a replay buffer. As a result each sample that is generated by playing in the environment is only used for training once and then discarded. A follow up experiment could investigate whether adding replay buffers to the individual worker instances would improve sample efficiency and thus allow for training the environments in fewer episodes.

4.5.4 Entropy

As suggested in the original paper on A3C we added an entropy term to the loss function of the network. However, contrary to the experimental results in the paper we observed that adding a penalty for larger entropy encouraged convergence and led to better training results. A visualisation can be seen in the appendix in Figure 12. These experimental results and the figure are obtained by training the algorithm on the CartPole environment from OpenAI Gym [29]. We trained the algorithm on the provided parameters instead of pixel data to allow for faster iteration and exploration of different hyperparameter settings.

4.5.5 Gradient Clipping

While experimenting with the hyperparameters of our algorithm we discovered that in some runs we encountered exploding gradients. As a result, the agent would never converge to a stable policy. We tried to remedy this by using gradient norm clipping. If the norm of the gradients exceeded a cutoff value the gradient vector was normalized to this cutoff value. We observed mixed results using different norm thresholds: Restricting the norm solved the exploding gradient problem, but choosing a norm that is too restrictive slowed down convergence. However, smaller norm thresholds also increased stability during training. A visualisation can be seen in the appendix in Figure 11. Results were obtained from the CartPole environment as described above.

4.5.6 Lookahead

A3Cs 'lookahead' parameter determines how many steps an agent takes before computing the gradient. Similarly to n-step DQN, credit is assigned with respect to all steps taken since the last gradient computation. Rewards are discounted, such that actions more distant in time to the reward receive less credit. We observed that a longer lookahead leads to larger cumulative gradients and faster learning. However, a shorter lookahead seems to lead to slightly more stable training and smaller losses in general. It seems reasonable that normalizing the loss with the number of steps taken to compute it could lead to more stable training while still preserving the benefits of assigning credit over multiple steps. A visualisation can be seen in the appendix in Figure 10 (data from CartPole environment as above).

4.5.7 Network Architecture

A3C learns both a policy for each state (actor) as well as a value for each state (critic). Both of these are realized through two separate, shallow, fully connected feedforward networks. Since we only have access to pixel data from the game, we additionally opted to use a convolutional network to learn a lower dimensional representation of the game that can be used as an input to the policy and value networks. By splitting the networks in three parts in this way, value and policy networks can share a common representation of the game so that we avoid training two essentially redundant convolutional networks. Further, each of the three parts of the network also has its own optimizer such that we have control over the optimization of the separate parts.

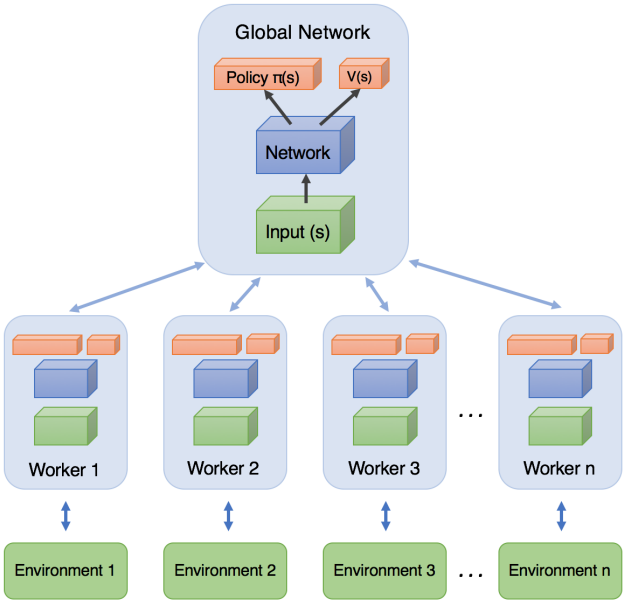


Figure 5: **A3C high-level architecture** Showing the distinction between value and policy network across multiple agents. Each agent runs in a separate instance of the environment. The diagram is reproduced from [32]

5 Conclusion and Discussion

Conclusion Our work shows a successful implementation of the A3C algorithm with the OpenAI Gym CartPole problem. We showed the effects of parameter changes in the look ahead, using or not using gradient clipping and utilizing different entropy weights. Further, we explained why reinforcement learning in general and actor-critic models in particular are biologically plausible. We also implemented an autoencoder, an image preprocessing pipeline and an interface that makes switching between environments effortless. However, we did not succeed in transferring our results to learning from pixel values. This made it impossible to solve the Neurosmash environment.

Limitations One of the largest limitations of the current work is the inability to make use of convolutional layers in a useful manner. In future versions of the Neurosmash environment, a different implementation of speeding up the time scale could decrease bugs. We found that adjusting the time frame to values above 5 sometimes led the agents to 'jump' out of the accessible area without ending the game.

Future Work In future work, it would be interesting to compare the obtained results with other algorithms in terms of implementation, training time and final performance. A higher agent diversity is assumed to increase robustness of the exploration process [2], this is something that would be interesting to look into. It would be possible to use non-random seeds, decrease the synchrony of the agents and increase diversity further by using different learning rates. Due to a lack of time we were not able to implement the A2C algorithm and compare its performance on the Neurosmash environment with the A3C algorithm. Prior research found the A2C algorithm to be more effective in solving multiple environments, it would be interesting to see whether this claim holds true for Neurosmash. Further, it might be interesting to investigate making changes to the two agents simultaneously. Especially since the complexity of the trained agent is limited by the complexity of the environment. Bansal et. al [33] used this fact by creating a

competitive multi-agent environment that contains several agents simultaneously. This does not only provide the agent with a natural learning curriculum, but with enough degrees of freedom in the action space also allows for emergent strategies. Through making it possible to play against one another, a continuous league could be created. This would allow a form of learning that DeepMind called 'population-based and multi-agent reinforcement learning' [34].

ACKNOWLEDGEMENTS

We thank Umut Güçlü, Lynn Le and Josh Ring for helpful discussions. We also thank the AI HPC cluster for providing computational resources. And the CNS and Radboud travel grant that enable us to present our work at the Interdisciplinary-College spring school.

References

[1] Vijaymohan Konda. *Actor-critic Algorithms*. PhD thesis, Massachusetts Institute of Technology, Cambridge, MA, USA, 2002. AAI0804543.

[2] Volodymyr Mnih, Adrià Puigdomènech Badia, Mehdi Mirza, Alex Graves, Timothy P. Lillicrap, Tim Harley, David Silver, and Koray Kavukcuoglu. Asynchronous methods for deep reinforcement learning. *CoRR*, abs/1602.01783, 2016.

[3] Adam Paszke, Sam Gross, Soumith Chintala, Gregory Chanan, Edward Yang, Zachary DeVito, Zeming Lin, Alban Desmaison, Luca Antiga, and Adam Lerer. Automatic differentiation in pytorch. *31st Conference on Neural Information Processing Systems*, 2017.

[4] Umut Güclü. Neurosmash. Unpublished Work, 2020.

[5] Yuxi Li. Deep reinforcement learning: An overview. *CoRR*, abs/1701.07274, 2017.

[6] David Silver, Aja Huang, Chris J. Maddison, Arthur Guez, Laurent Sifre, George van den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Veda Panneershelvam, Marc Lanctot, Sander Dieleman, Dominik Grewe, John Nham, Nal Kalchbrenner, Ilya Sutskever, Timothy Lillicrap, Madeleine Leach, Koray Kavukcuoglu, Thore Graepel, and Demis Hassabis. Mastering the game of Go with deep neural networks and tree search. *Nature*, 529(7587):484–489, January 2016.

[7] Jens Kober, J. Andrew Bagnell, and Jan Peters. Reinforcement learning in robotics: A survey. *Int. J. Rob. Res.*, 32(11):1238–1274, September 2013.

[8] Changliu Liu and Masayoshi Tomizuka. Algorithmic safety measures for intelligent industrial co-robots. In *2016 IEEE International Conference on Robotics and Automation, ICRA 2016, Stockholm, Sweden, May 16-21, 2016*, pages 3095–3102, 2016.

[9] OpenAI. https://spinningup.openai.com/, 2018. Accessed: 2019-12-12.

[10] C. J. C. H. Watkins. *Learning from Delayed Rewards*. PhD thesis, King's College, Oxford, 1989.

[11] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin A. Riedmiller. Playing atari with deep reinforcement learning. *CoRR*, abs/1312.5602, 2013.

[12] 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. Human-level control through deep reinforcement learning. *Nature*, 518(7540):529–533, February 2015.

[13] Christopher J. C. H. Watkins and Peter Dayan. Technical note: q -learning. *Mach. Learn.*, 8(3-4):279–292, May 1992.

[14] OpenAI. https://openai.com/blog/baselines-acktr-a2c/, 2017. Accessed: 2019-12-19.

[15] Yuhuai Wu, Elman Mansimov, Shun Liao, Roger B. Grosse, and Jimmy Ba. Scalable trust-region method for deep reinforcement learning using kronecker-factored approximation. *CoRR*, abs/1708.05144, 2017.

[16] James Martens and Roger B. Grosse. Optimizing neural networks with kronecker-factored approximate curvature. In *Proceedings of the 32nd International Conference on Machine Learning, ICML 2015, Lille, France, 6-11 July 2015*, pages 2408–2417, 2015.

[17] I. Rechenberg. *Evolutionsstrategie : Optimierung technischer Systeme nach Prinzipien der biologischen Evolution*. Number 15 in Problemata. Frommann-Holzboog, Stuttgart-Bad Cannstatt, 1973.

[18] Tim Salimans, Jonathan Ho, Xi Chen, Szymon Sidor, and Ilya Sutskever. Evolution Strategies as a Scalable Alternative to Reinforcement Learning, September 2017.

[19] Marco Tomassini. Parallel and distributed evolutionary algorithms: A review, 1999.

[20] D. Hassabis, D. Kumaran, C. Summerfield, and M. Botvinick. Neuroscience-inspired artificial intelligence. *Neuron Review*, 45:245–258, 2017.

[21] Richard S. Sutton and Andrew G. Barto. *Reinforcement Learning: An Introduction*. The MIT Press, second edition, 2018.

[22] Marcel van Gerven. Computational foundations of natural intelligence. *Front. Comput. Neurosci.*, 2017, 2017.

[23] David Ha and Jürgen Schmidhuber. World models. *CoRR*, abs/1803.10122, 2018.

[24] Daphna Joel, Yael Niv, and Eytan Ruppin. Actor–critic models of the basal ganglia: new anatomical and computational perspectives. *Neural Networks*, 15:535–547, 2002.

[25] Andrew G. Barto. Adaptive critics and the basal ganglia. In J. C. Houk, J. L. Davis, and D. G. Beiser, editors, *Models of Information Processing in the Basal Ganglia*, pages 215–232. MIT Press, Cambridge, MA, 1995.

[26] E. R. Kandel, J. H. Schwartz, T. M. Jessell, S. A. Siegelbaum, and Hudspeth A. J. *Principles of neural science*. McGraw-Hill Education, 5th edition, 2012.

[27] S. E. Bosch, K. Seeliger, and M. A. J. van Gerven. Modeling Cognitive Processes with Neural Reinforcement Learning. *bioRxiv*, page 084111, Oct 2016.

[28] J.X. Wang, Z. Kurth-Nelson, D. Kumaran, D. Tirumala, H. Soyer, J. Z. Leibo, D. Hassabis, and M. Botvinick. Prefrontal cortex as a meta-reinforcement learning system. *Nature Neuroscience*, 21:860–868, 2018.

[29] Greg Brockman, Vicki Cheung, Ludwig Pettersson, Jonas Schneider, John Schulman, Jie Tang, and Wojciech Zaremba. Openai gym, 2016.

[30] Pierre Baldi. Autoencoders, unsupervised learning, and deep architectures. In *Proceedings of ICML workshop on unsupervised and transfer learning*, pages 37–49, 2012.

[31] Benjamin Recht, Christopher Re, Stephen Wright, and Feng Niu. Hogwild: A lock-free approach to parallelizing stochastic gradient descent. In J. Shawe-Taylor, R. S. Zemel, P. L. Bartlett, F. Pereira, and K. Q. Weinberger, editors, *Advances in Neural Information Processing Systems 24*, pages 693–701. Curran Associates, Inc., 2011.

[32] Arthur Juliani. Simple reinforcement learning with tensorflow part 8: Asynchronous actor-critic agents (a3c), Jun 2017.

[33] Trapit Bansal, Jakub Pachocki, Szymon Sidor, Ilya Sutskever, and Igor Mordatch. Emergent complexity via multi-agent competition. *CoRR*, abs/1710.03748, 2017.

[34] Oriol Vinyals, Igor Babuschkin, Junyoung Chung, Michael Mathieu, Max Jaderberg, Wojtek Czarnecki, Andrew Dudzik, Aja Huang, Petko Georgiev, Richard Powell, Timo Ewalds, Dan Horgan, Manuel Kroiss, Ivo Danihelka, John Agapiou, Junhyuk Oh, Valentin Dalibard, David Choi, Laurent Sifre, Yury Sulsky, Sasha Vezhnevets, James Molloy, Trevor Cai, David Budden, Tom Paine, Caglar Gulcehre, Ziyu Wang, Tobias Pfaff, Toby Pohlen, Dani Yogatama, Julia Cohen, Katrina McKinney, Oliver Smith, Tom Schaul, Timothy Lillicrap, Chris Apps, Koray Kavukcuoglu, Demis Hassabis, and David Silver. AlphaStar: Mastering the Real-Time Strategy Game StarCraft II. `https://deepmind.com/blog/alphastar-mastering-real-time-strategy-game-starcraft-ii/`, 2019.

6 Appendix

6.1 Perception action cycle

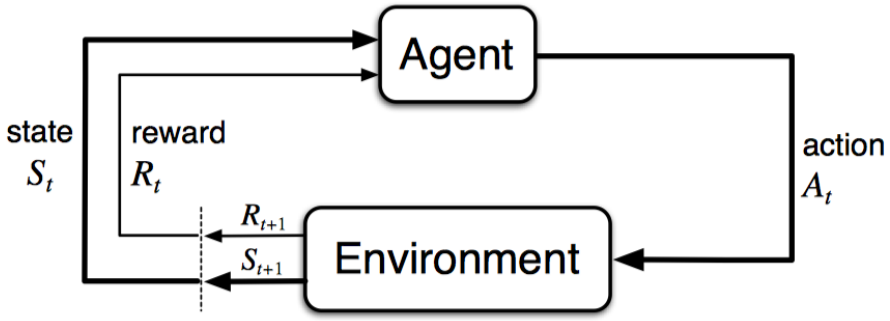


Figure 6: Showing the canonical agent-environment feedback loop [21]

6.2 Algorithms

Algorithm 1 Deep Q-learning with Experience Replay

Initialize replay memory $\mathcal{D}$ to capacity N
Initialize action-value function Q with random weights
for episode $= 1, M$ **do**
  Initialise sequence $s_1 = \{x_1\}$ and preprocessed sequenced $\phi_1 = \phi(s_1)$
  for $t = 1, T$ **do**
    With probability ϵ select a random action a_t
    otherwise select $a_t = \max_a Q^*(\phi(s_t), a; \theta)$
    Execute action a_t in emulator and observe reward r_t and image x_{t+1}
    Set $s_{t+1} = s_t, a_t, x_{t+1}$ and preprocess $\phi_{t+1} = \phi(s_{t+1})$
    Store transition $(\phi_t, a_t, r_t, \phi_{t+1})$ in $\mathcal{D}$
    Sample random minibatch of transitions $(\phi_j, a_j, r_j, \phi_{j+1})$ from $\mathcal{D}$
    Set $y_j = \begin{cases} r_j & \text{for terminal } \phi_{j+1} \\ r_j + \gamma \max_{a'} Q(\phi_{j+1}, a'; \theta) & \text{for non-terminal } \phi_{j+1} \end{cases}$
    Perform a gradient descent step on $(y_j - Q(\phi_j, a_j; \theta))^2$ according to equation 3
  end for
end for

Figure 7: Showing the algorithm we implemented for the training of the DQN agent. The implementations follows [12]

Algorithm S3 Asynchronous advantage actor-critic - pseudocode for each actor-learner thread.

// Assume global shared parameter vectors θ and θ_v and global shared counter $T = 0$
// Assume thread-specific parameter vectors θ' and θ'_v
Initialize thread step counter $t \leftarrow 1$
repeat
  Reset gradients: $d\theta \leftarrow 0$ and $d\theta_v \leftarrow 0$.
  Synchronize thread-specific parameters $\theta' = \theta$ and $\theta'_v = \theta_v$
  $t_{start} = t$
  Get state s_t
  repeat
    Perform a_t according to policy $\pi(a_t|s_t;\theta')$
    Receive reward r_t and new state s_{t+1}
    $t \leftarrow t + 1$
    $T \leftarrow T + 1$
  until terminal s_t **or** $t - t_{start} == t_{max}$
  $R = \begin{cases} 0 & \text{for terminal } s_t \\ V(s_t, \theta'_v) & \text{for non-terminal } s_t \text{// Bootstrap from last state} \end{cases}$
  for $i \in \{t-1, \ldots, t_{start}\}$ **do**
    $R \leftarrow r_i + \gamma R$
    Accumulate gradients wrt θ': $d\theta \leftarrow d\theta + \nabla_{\theta'} \log \pi(a_i|s_i;\theta')(R - V(s_i;\theta'_v))$
    Accumulate gradients wrt θ'_v: $d\theta_v \leftarrow d\theta_v + \partial \left(R - V(s_i;\theta'_v)\right)^2 / \partial \theta'_v$
  end for
  Perform asynchronous update of θ using $d\theta$ and of θ_v using $d\theta_v$.
until $T > T_{max}$

Figure 8: Showing the algorithm we implemented for the training of the A3C agent. The implementations follows [2].

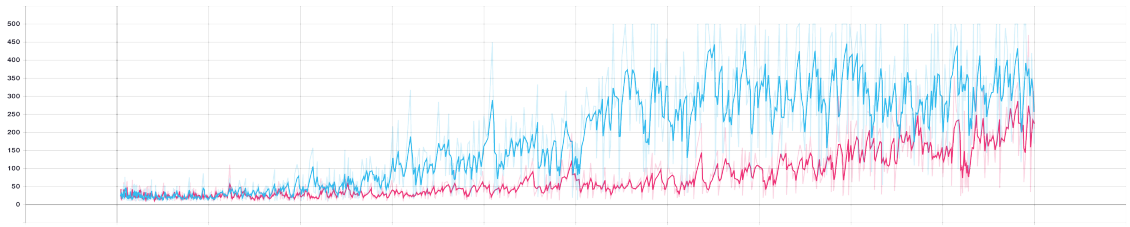
Algorithm 1 Evolution Strategies

1: **Input:** Learning rate α, noise standard deviation σ, initial policy parameters θ_0
2: **for** $t = 0, 1, 2, \ldots$ **do**
3: Sample $\epsilon_1, \ldots \epsilon_n \sim \mathcal{N}(0, I)$
4: Compute returns $F_i = F(\theta_t + \sigma\epsilon_i)$ for $i = 1, \ldots, n$
5: Set $\theta_{t+1} \leftarrow \theta_t + \alpha \frac{1}{n\sigma} \sum_{i=1}^{n} F_i \epsilon_i$
6: **end for**

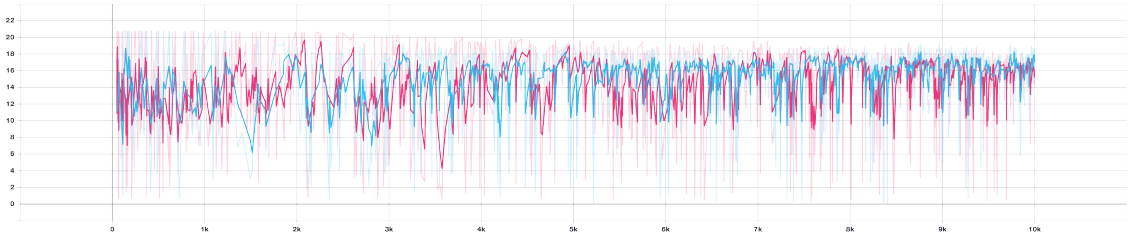
Figure 9: Showing the algorithm we implemented for the training of the ES agent. The implementations follows [18]

6.3 Parameter configurations

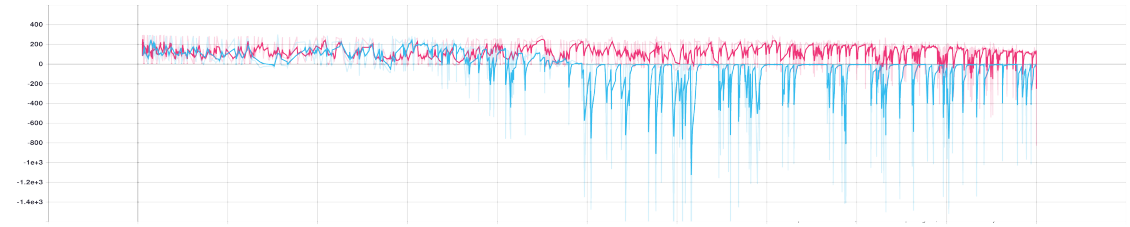
Number of runs to train: 2.000-30.000
Number of workers (agents): set between 6-12
Look ahead between 30-100
Epsilon, different exploration-exploitation parameters. We used three different annealing schemes, namely fast-annealing, slow-annealing and linear annealing. We adjusted the epsilon value starting from one and annealed it to a final value of 0.1-0.05.
We used different learning rates and loss functions for the policy, the value and convolution part of the network. We set the learning rate (lr.) between 0.01-0.0001
The optimizers we used were RMSProp, SGD and Adam
We also used a weight decay for the policy and value optimizer of 0.0001
During the evaluation we looked at between 10 and 80 episodes
We used gradient clipping to get an upper bound for value and the policy network
To discount future rewards we used a gamma factor of 0.99 (a value we found in the original A3C paper)
We put a weighted entropy value in the loss function to increase exploration behaviour during low epsilon values



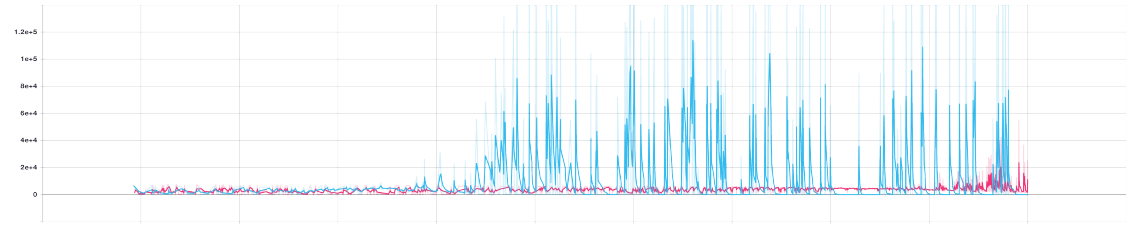
(a)



(b)

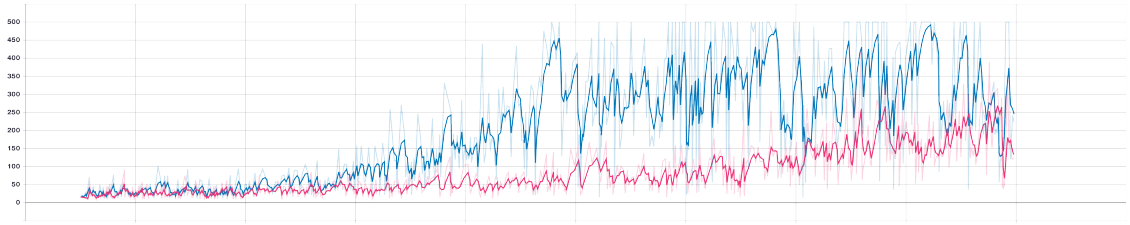


(c)

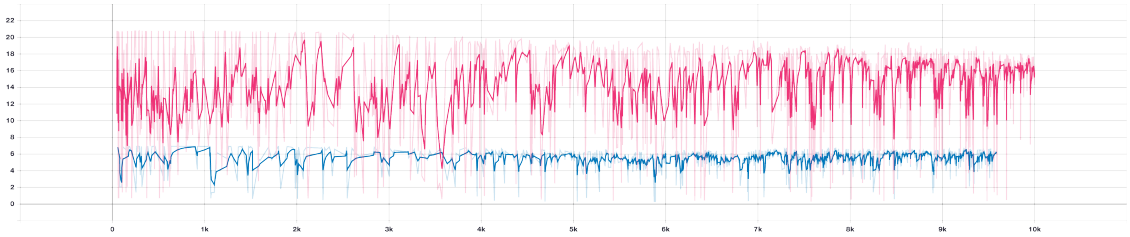


(d)

Figure 10: In this figure we changed the look ahead from 30 to 10 steps. (a) Shows the reward obtained by agent (b) shows the entropy (c) shows the policy loss (d) shows the value loss.

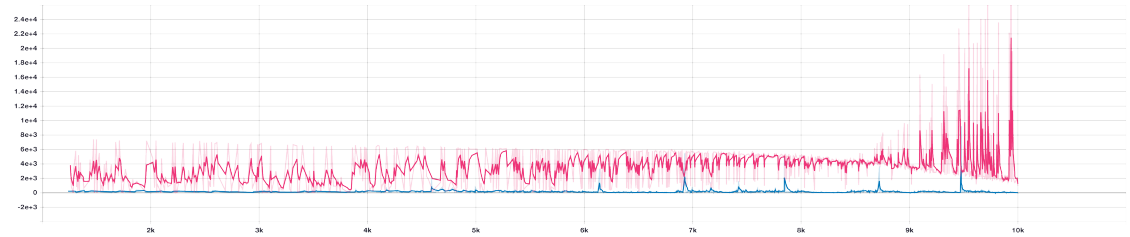


(a)



(b)

(c)



(d)

Figure 11: In this figure we switched gradient clipping on and off. (a) Shows the reward obtained by agent (b) shows the entropy (c) shows the policy loss (d) shows the value loss.

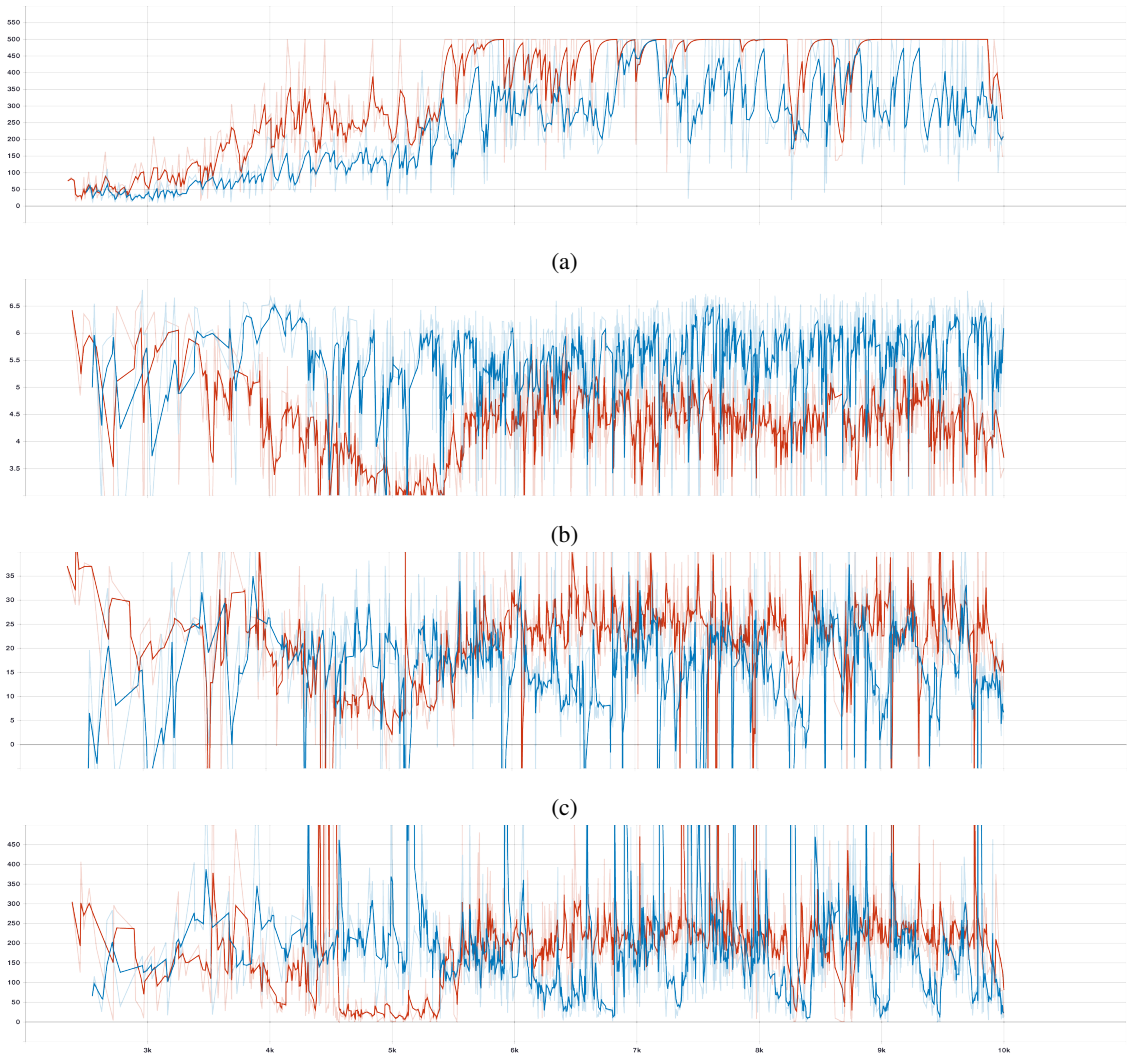


Figure 12: In this run we changed the entropy weight from -0.1 to 1. (a) Shows the reward obtained by agent (b) shows the entropy (c) shows the policy loss (d) shows the value loss.