University of Stavanger

FACULTY OF SCIENCE AND TECHNOLOGY

MASTER'S THESIS

Study programme/specialisation: Master of Science in Applied Data Science	Spring/ ~~Autumn~~ semester, 2021 Open / ~~Confidential~~
Author: Andreas Nesse	
Programme coordinator: Antorweep Chakravorty Supervisor(s):	
Title of master's thesis: Implementing an Adaptive Genetic Algorithm in the Atari Environment	
Credits: 30	
Keywords: Evolutionary Algorithm, Genetic Algorithm, Deep Q-Network, DQN, ACROMUSE, Atari, Neural Network, Machine Learning	Number of pages: ……44…… + supplemental material/other: ……14…… Stavanger, …4 / 6 / 2021…. date/year

Implementing an Adaptive Genetic Algorithm in the Atari Environment

Andreas Nesse

Abstract. This thesis attempts to implement a genetic algorithm for training agents within the Atari game environments. The training is performed on hardware of a widely available character, and so the results give an indication of how well these models perform on relatively inexpensive equipment available to many people. The Atari environment Space Invaders was chosen to train and test the models in. As a baseline, a Deep Q-Network (DQN) algorithm is implemented within TensorFlow's TF-Agents framework. The DQN is a popular model that has inspired many new algorithms and is often used as a comparison to alternative approaches. An adaptive genetic algorithm called ACROMUSE was implemented and compared with the performance of the DQN within the environment. This algorithm adaptively determines crossover rates, mutation rates and tournament selection size. Using measures for diversity and fitness, two subpopulations are maintained to avoid converging toward a local optimum. Based on the results found here, the algorithm did not seem to converge or produce high-performing agents, and importantly performed worse than the DQN approach. The reasons for why this algorithm fails and why other genetic algorithms have succeeded are discussed. The large number of weight parameters present in the network seem to be a barrier to good performance. It is suggested that a parallel training approach is necessary to reach the number of agents and generations where a good solution could be found. It is also shown how the number of frames skipped in the environment had a significant impact on the performance of the baseline DQN model.

Table of Contents

List of Figures

List of Tables

Preface

This thesis marks the end of my master's degree in Applied Data Science at the University of Stavanger. I hope it can also mark the beginning of a new, interesting time in my life. While it has been hard work at times, and things have not always gone to plan, it feels good to finish.

I want to thank my supervisor Antorweep Chakravorty for advice, direction and feedback during the process. I also want to thank Nikita Rajendra Karandikar, and the other students in the group, Daiana, Mohammed, Fredrik and Magnus, for the shared progress update meetings.

Finally, I want to thank my father Svein Olaf Nesse, along with the rest of my family, for moral support and interest in my thesis and progress.

Andreas Nesse
June 4, 2021

1 Introduction

As people want to create agents for data processing, image processing, autonomous cars, robotics and the like, there is a need for different algorithms to train agents to handle the challenges faced in the real world. How does one go about proposing and testing new models and training regiments?

Having a standard, varied set of tools to test agents on is very useful, and in the last years, the Atari 2600 library of games has become a popular option. In particular, the Arcade Learning Environment (ALE)[1], and later toolkits built on it, has made these environments easily available and usable.

Using these simple video games, which provide a variety of challenges the agent must adapt to, can be a valuable indication of how well such an agent could adapt to real-world challenges. The combination of short-term and long-term goals for the agent to pursue, makes for an interesting way to refine the already available approaches, and for building all-new algorithms as well.

The Deep Q-Network (DQN)[2][3] approach used as a baseline in this thesis has inspired many new variations, and have often been used as a comparison for new methods. This is the reason why the DQN approach is central to this thesis and is used to test the performance of the genetic algorithm. The genetic algorithm implemented here is called ACROMUSE[4]. It is adaptive, which means it controls certain hyperparameters depending on how training is progressing and the fitness of the agents. This, along with its goal of maintaining a diverse population, is why it was chosen. Although, as always, there are trade-offs, and new parameters that must be set appropriately beforehand.

1.1 Problem statement

The goal of this thesis can be put as follows:

"Implement and test an ACROMUSE genetic algorithm for use with an Atari environment. Compare this model with a baseline DQN reinforcement agent implementation. The training and testing will be performed on hardware of a widely available nature, to provide an indication of how useful these tools are when high-performance equipment is lacking."

2 Previous Work

Many papers have made use of the Atari 2600 games to test different machine learning algorithms, due to their complexity and the varying challenges they pose. A commonly used framework for running these Atari games is the Arcade Learning Environment (ALE) introduced in the paper 'The Arcade Learning Environment: An Evaluation Platform for General Agents'[1]. This is an open-source framework for using and interacting with Atari 2600 environments, based on the Stella emulator. Open AI's Gym toolkit[5][6] is built upon the ALE to allow easier installation and interaction with the Atari environments. This platform was used in [7]. M. C. Machado, M. G. Bellemare, E. Talvitie, *et al.* discuss the recent widespread use of the ALE environment, how stochasticity has been introduced in different ways, and the need for standardizing evaluation, in their paper 'Revisiting the Arcade Learning Environment: Evaluation Protocols and Open Problems for General Agents'[8]. The short paper [9] presents a case study evaluating different ways of introducing randomness into the Atari environment, to prevent agents from simply 'memorizing' an action sequence.

Q-Learning is a reinforcement learning method introduced over three decades ago[10][11] and has been applied to many problems within machine learning. Because it is impractical to run this full algorithm in more complex environments, an estimation function is often used. To estimate the Q-function, which is used to determine actions given a state in some environment, methods may use linear approximation function, as e.g. [12][1], or non-linear functions, such as a neural network, in [2][3][13], among others. Experience replay was discussed in [14] and can speed up training by saving previous experience in a buffer and using them repeatedly in training. Google's TensorFlow has a specific toolkit developed for working with reinforcement learning called TF-Agents[15][16].

Two very influential papers introduced using the Deep Q-Learning Network (DQN) method for training agents within the Atari environment. The first is the 2013 paper 'Playing Atari with Deep Reinforcement Learning'[2]. The DQN agent collects experience in the environment using an epsilon-greedy policy, based on a continuously updated Convolutional Neural Network (CNN) called the Q-network. An experience replay buffer of fixed size is used to continuously save the most recent experienced transitions. When training, a batch of transitions is sampled from the buffer and used to perform a gradient descent step. For efficiency, the paper uses frame skipping as well as stacking frames together before feeding through the network. A target network is periodically updated with the weights from the main Q-network, and acts to evaluate the actions taken by the main Q-network. This paper[2] uses seven of the available Atari games to test and evaluate their model against other previous work and shows that the proposed agent outperforms these on most games, even outperforming a human player in some cases. Unfortunately, many of the parameters and the code used to run the training in this paper is not widely available.

The second paper[3] was published in 2015 by many of the same authors as [2] in addition to some new participants. The DQN approach is further optimized, and a slightly larger Q-network structure is used. This paper focuses on

comparing this model to the performance of a professional human tester and previously proposed reinforcement learning[1] and contingency awareness methods[12]. This paper extends the experiments to include 49 Atari games. The agent is shown to outperform both of the previous methods being compared in most games, and is at or above human level in many of the games. The authors have made the parameters and code available for review.

Further work has later built on this DQN method such as H. van Hasselt, A. Guez, and D. Silver describing the Double-DQN model[13], which uses the continuously updated Q-network in a greedy policy to pick an action for a given state in the environment, but uses the lagging target network to estimate the Q-value of this action. This is implemented to deal with the DQN method's tendency to overestimate the Q-value for a given action. The target network is still used to evaluate the action taken by the epsilon-greedy policy and is periodically updated as in DQN. In [17] T. Schaul, J. Quan, I. Antonoglou, *et al.* introduces prioritizing to the experience replay. They propose using the TD error found during a learning step to assign a probability to the transition. The TD error acts as an estimate of how 'surprising' the transition is, and the experience transitions that deviate the most from the policy's predicted action have a higher probability of being sampled to correct this deviation. The papers [18] and [19] suggest ways to distribute the training in a DQN model. M. Hausknecht and P. Stone discuss how games requiring a memory larger than the typical stack of frames, behave like a Partially-Observable Markov Decision Process and propose adding a Long Short-Term Memory (LSTM) recurrent network in place of the first convolutional layer in the traditional DQN[3]. They concluded that this was viable but did not present a significant improvement for most Atari games. An asynchronous n-step DQN, accumulating rewards and applying a combined gradient after n steps, is introduced in [21] along with the well-known Asynchronous Advantage Actor-Critic (A3C) algorithm. Many other papers have been written proposing similar methods either building on the DQN model or other reinforcement learning methods and testing them within the Atari environments[22][23]. This shows how useful the Atari games have been in developing these agents, and how influential the initial DQN papers[2][3] have been.

Genetic Algorithms (GA) are evolutionary algorithms that roughly mimic the genetic change seen in a population of organisms in nature. K. De Jong describes three necessary elements of GAs[24], which include some measure of an agent's fitness, a way of creating offspring agents and a method of basing an offspring's genes on the parent's genetic material. Several papers have been written using evolutionary and genetic algorithms for training agents to perform in the Atari game environments. F. P. Such, V. Madhavan, E. Conti, *et al.* in their paper 'Deep Neuroevolution: Genetic Algorithms Are a Competitive Alternative for Training Deep Neural Networks for Reinforcement Learning'[25] apply a genetic algorithm using only simple mutation, a large population size of 1000 agents, elitism and truncated selection. The specific implementation shares many features with the second DQN paper mentioned earlier[3], such as network structure. This method produced similar or better results on many of the Atari

games as compared to the previous methods considered in the paper. They use a special type of encoding to compress the size of the networks, which allows for distributed running, but also makes applying more complex forms of genetic algorithms less straight forward. Because their implementation makes use of distributed running when evaluating the agents and utilizes the capabilities of GPUs to speed up forward-passes through the network, they are able to train capable agents very quickly. The authors claim that this training can be done in $\sim$4 hours on a modern desktop computer, although it should be mentioned that the desktop computer described in their paper seems to be of a purpose-built high-performance specification. Other evolutionary algorithms are also applied, such as in the paper 'A Neuroevolution Approach to General Atari Game Playing'[26], which builds on the earlier paper [27], where several evolutionary algorithms are applied to evolve agents within the Atari environments. Only the method called HyperNEAT is deemed appropriate for using the raw pixel input from the game as input to its network. HyperNEAT is a method proposed in the paper [28] and uses indirect encoding of an artificial neural network. The indirect encoding is carried out by small Connective Compositional Pattern-Producing Networks (CPPNs) which encode the connections in the neural network. These CPPNs are evolved using the NeuroEvolution of Augmenting Topologies (NEAT) algorithm[29]. Applying this HyperNEAT method to training agents for playing Atari games[26], M. Hausknecht, J. Lehman, R. Miikkulainen, *et al.* find it works well for many games, but that interestingly, different types of algorithms seem best suited for different types of games. Yet another method is used in the paper 'Evolution Strategies as a Scalable Alternative to Reinforcement Learning'[7], where an Evolution Strategies (ES) approach is used in the Atari environments. Due to its ability to run well in parallel, it can run in a short amount of time given powerful and plentiful enough hardware, and it produces capable agents in many of the games when compared to other methods.

A challenge for genetic algorithms, which they share with practically all other machine learning methods, is choosing the right training hyperparameters. An option is to tune the hyperparameters by analyzing the algorithm's performance using different parameter values and choosing the best parameters from there. A framework for this is presented in [30]. A different option is to control the parameters in some other way. A. E. Eiben, R. Hinterding, and Z. Michalewicz discussed the terminology and the contemporary methods of hyperparameter control in their 1999 paper [31]. In 2015 an updated paper, 'Parameter Control in Evolutionary Algorithms: Trends and Challenges'[32], gives an overview of more recent innovations in this area. Different forms of control and ways of controlling parameters such as population size, mutation rate and selection pressure are discussed. In section V of [32], the authors discuss Control Ensembles, which are methods that combine several control mechanisms for different parameters at the same time. Mentioned here, among others, is R. Hinterding, Z. Michalewicz, and T. C. Peachey's[33] method for self-adaptation of mutation strength with an adaptive population size. F. Herrera and M. Lozano[34] propose a method using two fuzzy logic controllers to maintain a good balance between exploration

and exploitation. Also described is the ACROMUSE ensemble, discussed in the following paragraph.

In their paper 'Maintaining Healthy Population Diversity Using Adaptive Crossover, Mutation, and Selection'[4], B. McGinley, J. Maher, C. O'Riordan, *et al.* introduce the adaptive genetic algorithm which they name ACROMUSE. The goal of this algorithm is to prevent the agents from converging to a local optimum, and instead maintain a diverse population of high-performing agents spread throughout the search space. ACROMUSE uses two measures to achieve this. The first measure is the Standard Population Diversity (SPD) which is strictly a measure of the variation in the agents in the population with no regard to the fitness of the agents. The contribution of an agent to the SPD depends on its Euclidean distance to the average individual. It is mentioned that several previous papers, such as [35] and [36], have used similar approaches to ensure diversity in the population while training with a genetic algorithm. However, a second measure is proposed to ensure that the agents are not only diverse, but also high-performing. This is called the Healthy Population Diversity (HPD). How much an agent contributes to the HPD depends both on its Euclidean distance to a fitness-weighted average individual, and its own fitness. ACROMUSE effectively maintains two populations, one for exploitation, containing agents created using crossover and a low mutation rate, and one for exploration, containing agents created by mutating a single parent. The size of these populations is determined by the crossover rate. In both cases the parent(s) are chosen through tournament selection where the HPD contribution of the participating agents determines which are selected for procreation. Several operators determine the crossover rate, the mutation rate, and the size of the tournament for each generation based on the SPD and HPD values. Elitism is also employed to carry over the best performing agent in each generation. ACROMUSE as implemented in the paper uses a purposefully small population size of 40 agents to avoid the method becoming too resource intensive. The paper[4] describes that ACROMUSE outperforms other algorithms in the evaluations performed using multimodal function optimization benchmarks both with respect to fitness and diversity.

ACROMUSE was used in [37] in combination and comparison with other adaptive methods to optimize early building designs, and this paper also suggests using variable SPD and HPD maximum values. These are set according to the highest observed values at any point while running the genetic algorithm.

3 Code Availability

The code is available on GitHub, from:

`https://github.com/a-nesse/acromuse_atari`

Corrections or additions to the repository and aesthetic changes to the code might occur, but there is no guarantee that the code will be maintained going forward. See appendix C for reference to specific commit/code version used for training.

4 Deep Q-Network Algorithm

The Deep Q-Network (DQN) algorithm is a reinforcement algorithm based on the Q-learning method[10][11] which selects actions based on the policy formed by equation (1).

$$Q^*(s,a) = \max_{\pi} \mathbb{E}[r_t + \gamma r_{t+1} + \gamma^2 r_{t+2} + \cdots | s_t = s, a_t = a, \pi] \tag{1}$$

This function simply states the policy, π, should select the action with the greatest potential total reward. The future rewards are discounted using γ, to reduce the influence of potential rewards far in the future. Since it is not practically possible to map all states and actions in the Atari environments, this function is approximated using some sort of function, often linear. DQN uses a neural network[3] which is a non-linear function. In [3] it is explained how non-linear functions will generally fail at this task. However, using experience replay with random sampling, and letting the target network only be updated at regular intervals, DQN removes the correlation that could cause the neural network to fail to converge. The convolutional neural network used in DQN is called the Q-network in the model, where the target network has the same structure.

The DQN model implemented in this thesis follows the structure in [2] and [3] as closely as is possible using TensorFlow's TF-Agents toolkit[15][16] and within the time frame available. With a few exceptions, the parameters used in [3] is used, since the paper is of a later date and both the parameters and code are available. The exceptions are the network structure, the epoch length, target network update frequency, the amount of steps used for evaluation and the total steps used for training, which follow [2], due to limitations in time and hardware resources. Because of this, the final DQN implementation will be compared to [2], to check that it produces similar results. In addition, the size of the experience replay buffer is reduced and a slightly different loss function is used. The Q-network structure used in the model is described in section 6 and follows the structure used in [2].

DQN uses separate training and evaluation environments. They are described in section 7.1. The training environment is used for collecting experience and the evaluation environment is strictly used to evaluate the model at the set interval.

4.1 Differences from previous papers

The DQN model in [3] clips the error term if the element-wise loss falls outside the range $[-1, 1]$. The element-wise error used in the loss function takes form as is shown in equation (2).

$$x_i = \begin{cases} (y_j - Q(\phi_j, a_j, \theta_i))^2 & \text{if } |x_i| \leq 1 \\ |y_j - Q(\phi_j, a_j, \theta_i)| & \text{if } |x_i| > 1 \end{cases} \tag{2}$$

However, the DQN model implemented in this thesis uses the Huber loss[38] that is implemented in TensorFlow. This gives us the element-wise error used in the loss function shown in equation (3).

$$x_i = \begin{cases} \frac{1}{2}(y_j - Q(\phi_j, a_j, \theta_i))^2 & \text{if } |x_i| \leq \delta \\ \delta|y_j - Q(\phi_j, a_j, \theta_i)| - \frac{1}{2}\delta^2 & \text{if } |x_i| > \delta \end{cases} \tag{3}$$

$\delta = 1$ is used to line up with [3] using the range $[-1, 1]$. This loss function is scaled by $\frac{1}{2}$ to make the function differentiable.

The experience buffer had to be limited to a size of the 100 000 last frames, as opposed to the 1 million used in [2] and [3]. This is due to how TF-Agents handles saving the experience transitions in memory. With no way of saving each image frame only once and assembling the stacks when sampling, every frame is saved in four different stacks. When saving the experience replay buffer to disk, a copy of the buffer is made, which also requires a large amount of additional free memory space. For more information on how image frames are stacked, see section 7.2.

An epoch length of 50 000 training steps is used, which is consistent with [2], as opposed to [3] where 250 000 steps is used. However, the first paper counts one training step as the combination of a single experience collecting step and one gradient descent step. In the second paper, four experience collecting steps are performed for every gradient descent step, but every step counts towards the training step total regardless of whether a gradient descent update is performed. This thesis will define one training step as four experience collecting steps and one gradient descent step. This way, 50 000 gradient descent updates are performed in each epoch, as is done in [2]. Using four collect steps for each training step might also help with the smaller experience replay buffer size, by renewing the experience available for sampling more often. The target update frequency is set at every 10 000 gradient descent updates or training steps and is consistent with [2]. The same is true for the total training period, which is set at 100 epochs, equaling 5 million total training steps. Evaluation is carried out in the way described in [2] using the evaluation environment described in section 7.1.

It is not clearly stated which initialization is used for the Q-network weights in [3], so here TensorFlow's VarianceScaling initializer[39] is used with default parameters. This is similar to the initialization derived in [40], in that it uses a similar function for the standard deviation of the distribution, given in equation (4).

$$\text{SD} = \sqrt{\frac{\text{scale}}{n}} \tag{4}$$

The scale used here is 1.0 and n is the number of nodes in the input layer, as opposed to [40] which uses 2.0 and where the n varies between layers, depending on the number of nodes in the layer. TensorFlow's initializer also truncates the distribution by default, by discarding and redrawing values more than two standard deviations from the mean. No other parameters were tested for the initializer.

A full list of the hyperparameters used for training with DQN can be found in appendix A.1.

4.2 DQN algorithm description

In algorithm 1 the training process for the specific DQN implementation is described. Note that the image x_i here is meant to represent the 4-frame stack of preprocessed images that is returned from the training environment. The reward r_i is the accumulated reward for all the frames the stack represents. Refer to section 7.2 for a description on how frames are processed and stacked. Also note that every time an episode ends the training environment resets as is described in section 7.1. The ϵ update, which reduces the value of ϵ from 1.0 to 0.1 over the first 1 million frames, is not updated for every collect steps, but rather updated every training step, meaning four collect steps passes between each update. This deviation should be negligible.

The DQN model was implemented specifically for the thesis using TF-Agents to ensure the environment, preprocessing and frameworks are as similar as possible between the DQN model and the ACROMUSE genetic algorithm described in section 5. For a discussion around details concerning the technical implementation, refer to appendix C.1.

Algorithm 1: Deep Q-Network training with replay experience

Initialize experience replay buffer E
repeat
  Perform random action a_i in environment
  Collect accumulated clipped reward r_i and image stack x_{i+1}
  Save transition (x_i, a_i, r_i, x_{i+1}) in E
until *Replay Buffer Filled To Initial Capacity* B_0
Initialize Q-network Q with random weights θ
Initialize target network $\hat{Q}$ with weights $\hat{\theta} = \theta$
for *epoch = 1,M* **do**
  for *training step = 1,T* **do**
    for *collect experience steps = 1,C* **do**
      `// Collecting experience`
      repeat
        Perform random action a_c with probability $\epsilon_{\text{collect}}$
        Else perform action $a_c = \text{argmax}_a Q(x_c, a; \theta)$
        Save transition (x_c, a_c, r_c, x_{c+1}) in E
      until *Loss of life*
    end
    Reduce ϵ from 1.0 to 0.1 linearly over 1M collect steps
    Sample batch of transitions from E
    $$y_j = \begin{cases} r_j & \text{loss of life at step j+1} \\ r_j + \gamma \cdot \max_{a'} \hat{Q}(x_{j+1}, a'; \hat{\theta}) & \text{otherwise} \end{cases}$$
    Perform gradient descent step using eq.(3) with respect to θ
    Every N_T steps update target network so that $\hat{\theta} = \theta$
  end
  for *evaluation step = 1,V* **do**
    `// Evaluating agent`
    repeat
      Perform random action a_v with probability $\epsilon_{\text{eval}} = 0.05$
      Else perform action $a_v = \text{argmax}_a Q(x_v, a; \theta)$
      Add unclipped reward R_v to the accumulated episode score
    until *End of full episode*
    Evaluation score = average score of completed episodes
  end
  Save log and backup agent
end

5 ACROMUSE Genetic Algorithm

ACROMUSE was introduced in [4] by B. McGinley, J. Maher, C. O'Riordan, *et al.* It is a genetic algorithm which evolves generations of individual agents. New generations are created based on the previous generation's agents and their fitness, using evolutionary methods such as tournament selection, crossover, and mutation. The main focus of this method is avoiding convergence towards a local optimum, by maintaining a diverse population that is also high-performing. This is done using the two measures SPD and HPD. These are used in adaptive operators which determine the tournament size, mutation rate, and implicitly the size of the exploitation and exploration subpopulations. Although this model adaptively determines these hyperparameters, other fixed values are used to calculate these, which can be seen as new hyperparameters. For these new hyperparameters, the recommended values in [4] are used.

5.1 Standard Population Diversity (SPD)

SPD is a measure used to determine strictly the diversity of the population, using the Euclidean distance between the agent gene values. The fitness of the agents is not considered, so a high-performing population and a low-performing population can have the same SPD. To calculate SPD, first the average agent needs to be found.

Calculating the average agent The average agent is simply an agent where all the individual genes (network weights or parameters) are the average of the individual genes in the entire population. This is expressed in equation (5) where G_i, n is a specific gene n in individual i. P is the number of agents in the total population.

$$G_n^{\text{avg}} = \frac{1}{P} \sum_{i=1}^{P} G_{i,n} \tag{5}$$

Calculating SPD As explained in [4], to normalize the SPD measure, the gene-wise standard deviation is used in further calculations. This standard deviation is expressed in equation (6).

$$\sigma(G_n^{\text{avg}}) = \sqrt{\frac{1}{P} \sum_{i=1}^{P} (G_{i,n} - G_n^{\text{avg}})^2} \tag{6}$$

The SPD measure is defined as a coefficient of variation for the average individual, which is the reason for shifting and scaling the network weights, as is described in section 5.4. Equation (7) shows how the SPD is the average of the coefficients of variation for the average genes G_j^{avg}.

$$\text{SPD} = C_V(G^{avg}) = \frac{1}{N}\sum_{j=1}^{N}\left(\frac{\sigma(G_j^{\text{avg}})}{G_j^{\text{avg}}}\right) \tag{7}$$

This value is used in the operators to determine crossover rate and mutation rates, which are described in section 5.6.

5.2 Healthy Population Diversity (HPD)

The HPD measure considers both the diversity of the population and the fitness of the individuals by weighting their contribution to the measure by their relative fitness. These weights, not to be confused with network weights G_n, are calculated for agent i as described in equation (8), where f_i denotes the fitness of agent i.

$$w_i = \frac{f_i}{\sum_{k=1}^{P} f_k} \tag{8}$$

Again, a weighted calculated individual is found, here denoted $G_n^{W,avg}$, using equation (9).

$$G_n^{\text{W,avg}} = \sum_{i=1}^{P} w_i G_{i,n} \tag{9}$$

Calculating HPD HPD uses the gene-wise weighted standard deviation, expressed in equation (10).

$$\sigma(G_n^{\text{W,avg}}) = \sqrt{\sum_{i=1}^{P} w_i (G_{i,n} - G_n^{\text{avg}})^2} \tag{10}$$

The HPD is from here on calculated in a similar fashion to SPD. The coefficient of variation for the weighted average agent is shown in equation (11).

$$\text{HPD} = C_V(G^{W,avg}) = \frac{1}{N}\sum_{j=1}^{N}\left(\frac{\sigma(G_j^{\text{avg}})}{G_j^{\text{avg}}}\right) \tag{11}$$

Individual HPD contribution It is also necessary to calculate the individual agent's contribution to the HPD measure. This is done using equation (12).

$$\text{HPD}_i = w_i \sqrt{\sum_{n=1}^{N} (G_{i,n} - G_n^{W,avg})^2} \tag{12}$$

This contribution will be used to select parent agents in the tournaments, described in section 5.6.

Variable maximum SPD and HPD When testing the algorithm, it was apparent that SPD and HPD values above the practical maximum suggested in [4] occur, particularly early on in the first few generations. Using variable maximum values was suggested in [37] and involves using the largest observed value of SPD and HPD at any point in the training. These maximum values are initially set at $\text{SPD}_{\max} = 0.4$ and $\text{HPD}_{\max} = 0.3$ as suggested in [4], but each time SPD and HPD are calculated, these parameters are updated. The largest values will occur at the very beginning of training, typically in the initial generation, since at this point, weights are simply uniformly distributed in the search space and not concentrated around local optima. Using variable maximum SPD and HPD values has the effect of pushing the agents toward more exploration and less exploitation.

5.3 Population size

In [4], a generation size of only 40 agents is used. The paper explains that ACROMUSE was developed to function effectively using a small population. The same population size of 40 is used here. A larger, or significantly larger, population might be preferable, especially considering the large number of genes or parameters in the neural network. However, this is very resource intensive and given the modest hardware used for training, the trade-off between more coverage with a larger population and shorter time spent on each generation is difficult. A shorter time for each generation means the process of crossover and mutation can occur more often, hopefully further optimizing the agents.

5.4 Network weights and shifting

For the purposes of this thesis, the network weights for the ACROMUSE method are kept within a limited range. This is not directly prescribed in [4] but is done to limit the search space. It might be useful for the agent to be able to take both negative and positive values, which is why the range $\langle -1, 1 \rangle$ was chosen. There was no testing of alternative ranges. The network weights and biases are initialized with a simple uniform distribution in this range.

In the description of SPD and HPD it is mentioned that these are coefficients of variance. This means they need an absolute number range to work on. To achieve this the network weights are simply shifted and scaled from the range $\langle -1, 1 \rangle$ to $\langle 0, 1 \rangle$. Once shifted, the scale of the range does not matter, since the mean and variance would increase proportionally. This is only done to calculate the SPD and HPD measures, which measure the diversity and fitness of the population. Since these are measures of distance between agents, shifting and scaling the weights should not be an issue. Weights are shifted and scaled according to equation (13), where minval and maxval denote the lower and upper bounds of the network weight values.

$$G_n^* = \frac{G_n + \text{minval}}{\text{maxval} - \text{minval}} \tag{13}$$

Dealing with lower bound value weights While training with the ACROMUSE model, an occasional issue that came up was that a gene might be set to a value equal to the lower bound. Even though this in theory should be very improbable, it seems that in practice this is a very real possibility within the framework used. This might be due to some form of rounding that is happening somewhere in the process, but the exact cause was not determined. After several generations, a single such gene could propagate throughout the population of agents. In this case there would be a division of zero by zero, which gives a NaN value. To avoid this, every time network weights are assigned to an offspring, it is checked for values equaling the lower bound. Are any such values found, they are replaced by the lower bound plus a small buffer value set to $1 \cdot 10^{-6}$

5.5 Fitness and evaluation

There is no prescription for how to estimate the fitness of the agents in [4], due to how different the applications of the algorithm could be. It would take too long to run evaluation for all agents in the generation, so to approximate the fitness, 2 000 steps are run in the evaluation environment for each agent. The elite agent is the highest scoring agent in the generation and goes through full evaluation, as described in section 7.3. The elite agent is passed on unchanged to the next generation. This implementation uses the evaluation environment described in section 7.1 for all fitness approximation and evaluation purposes. No separate training environment is used.

It is an option to estimate the fitness using more steps in the environment, and to evaluate several of the highest scoring agents to determine which is the fittest to a greater degree of certainty. However, when trying this approach, it is apparent that this will take too much time, reducing the number of generations. Because of this, an increased uncertainty in estimating the fitness and finding the elite is chosen to facilitate more generations during training.

5.6 Generating offspring

The process of generating offspring in ACROMUSE, illustrated in figure 1, is determined by several factors controlled by the SPD and HPD values of the parent generation. Note that in this thesis, one network parameter or weight is considered as a single gene. All agents have the same number of genes since they all share network structure.

Once all agents in a generation have been given a fitness score and the elite agent is determined and evaluated, the process of creating the next generation starts. First, the SPD and HPD measures are calculated according to the process in sections 5.1 and 5.2. The elite agent is carried over unchanged.

Using the SPD measure the crossover rate is determined according to equation (14). This rate decides whether the next offspring belongs in the exploitation or exploration subpopulation. Agents in the exploitation population are created with two parents and a uniform crossover. The uniform crossover simply chooses a gene from either parent with uniform probability. Agents in the exploration

population have a single parent, meaning no crossover is performed. The new fixed hyperparameters are set to $K_1 = 0.4$ and $K_2 = 0.8$ as in [4] and define the upper and lower bound of the crossover rate.

$$P_c = \left[\left(\frac{\text{SPD}}{\text{SPD}_{\max}} \cdot (K_2 - K_1) \right) + K_1 \right] \tag{14}$$

To select either one or two parents for a next-generation agent, tournament selection is used. Agents are selected at random in the population and the highest rated agent is selected to be a parent. In the case of crossover, two parents are chosen this way. The parent agents are selected according to their HPD contribution, as opposed to their fitness score. This means the agents are selected according to both fitness and how much they contribute to the diversity of the population. The size of the tournament, meaning the number of agents selected at random, is determined by equation (15) where the maximum tournament size is given by (16).

$$\text{Tsize} = \left[\frac{\text{HPD}}{\text{HPD}_{\max}} \cdot \text{Tsize}_{\max} \right] \tag{15}$$

$$\text{Tsize}_{\max} = \left\lceil \frac{\text{Population size}}{6} \right\rceil \tag{16}$$

Finally, the agents are mutated. For agents going into the exploitation subpopulation, a small mutation rate of 0.01 is used, for local search. For agents going into the exploration subpopulation, which are not products of crossover, a higher mutation rate is used, which is adaptively determined according to equation (17). The new fixed hyperparameter is set to $K = 0.5$ as in [4] and defines the upper bound of the mutation rate.

$$P_m = \frac{P_m^{fit} + P_m^{div}}{2} \tag{17}$$

The two parts P_m^{fit} and P_m^{div} are determined according to equations (18) and (19), where f is the parent agent's fitness. $f_{\max}$ and $f_{\min}$ is the highest and lowest fitness scores in the generation.

$$P_m^{fit} = K \cdot \left(\frac{f_{\max} - f}{f_{\max} - f_{\min}} \right) \tag{18}$$

$$P_m^{div} = \frac{\text{SPD}_{\max} - \text{SPD}}{\text{SPD}_{\max}} \cdot K \tag{19}$$

If a gene is selected for mutation, it is re-initialized within the range specified in section 5.4 with uniform probability, as in [4].

Notice that the lower the SPD value, meaning there is low diversity, the larger the exploration subpopulation and the higher the mutation rate used for these agents. The operators also work to preserve more information in high fitness parents than in low fitness parents.

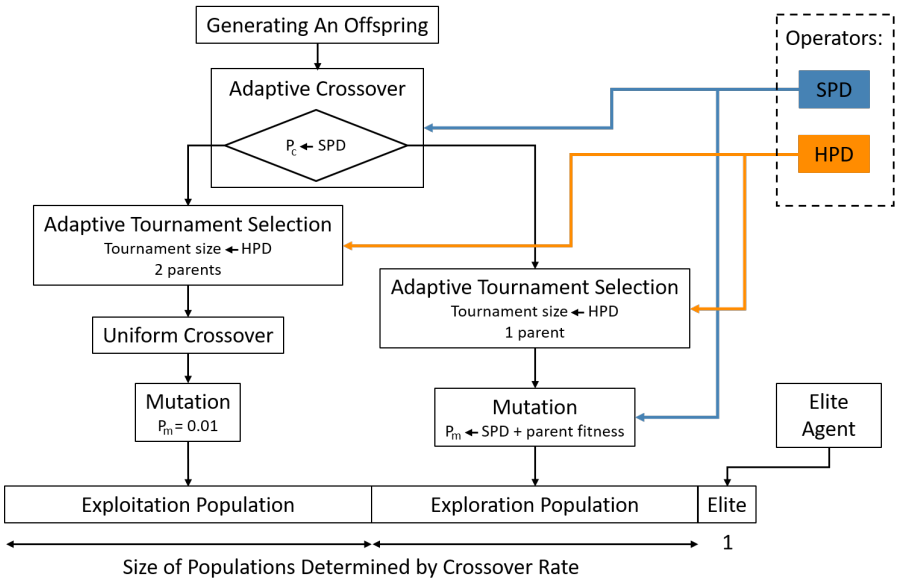


Fig. 1: The process of generating an offspring in a new generation. Note the variable population sizes at the bottom, although the total population is fixed. Based on fig. 5 in [4].

5.7 ACROMUSE algorithm description

The full process of training with ACROMUSE is described in algorithm 2. Note that tournament selection of agents is performed not with the agent score, but with the agent HPD contribution, described in section 5.2.

A full list of the hyperparameters used for training with ACROMUSE can be found in appendix A.2 and some aspects of the code implementation of ACROMUSE is discussed in appendix C.2.

Algorithm 2: ACROMUSE genetic algorithm

```
for Number of total generations do
    if the first generation then
        Initialize generation of random agents
    else
        // Creating new generation
        Preserve elite agent from previous generation
        for i=2,N_agents do
            Pick exploitation or exploration with probability P_crossover
            if exploitation then
                // adding agents to exploitation subpopulation
                Pick 2 parents using tournament selection with size T_size
                Perform uniform crossover between parents
                Mutate agents with probability P_mut, exploit = 0.01
            else
                // adding agents to exploration subpopulation
                Pick 1 parent using tournament selection with size T_size
                Mutate agents with probability P_mut, explore
            end
        end
    end
    for all agents i do
        // Estimating agent fitness
        repeat
            repeat
                Perform random action a_r with probability ε_eval = 0.05
                Else perform action selected by network
            until End of full episode
            Save unclipped scores for finished episodes
        until N_fit steps steps has passed
        Agent fitness = average score for finished episodes
    end
    Pick agent with highest average score as elite
    repeat
        // Evaluating elite agent
        repeat
            Perform random action a_v with probability ε_eval = 0.05
            Else perform action selected by network
        until End of full episode
        Save unclipped scores for finished episodes
    until N_eval steps steps have passed
    Evaluation score = average score for finished episodes
    Calculate SPD and HPD for generation
    Calculate HPD contribution for all agents, used in tournament selection
    Calculate P_crossover, P_mut, explore and tournament size T_size for next
     generation.
end
```

6 The Neural Network

Both agents share the same neural network structure. It follows the design laid out in the paper 'Playing Atari with Deep Reinforcement Learning'[2]. This network was chosen over the slightly larger network described in the following 2015 paper[3] by V. Mnih, K. Kavukcuoglu, D. Silver, *et al.* to save time and reduce the complexity.

The network is a convolutional neural network and is illustrated in figure 2. The input to the network is a stack of four frames, giving an input dimension of $84 \times 84 \times 4$. Following the input layer, the first hidden layer is a convolutional layer consisting of 16 filters. These have size 8×8 and move along the input frames with a stride of 4. The next convolutional layer has 32 filters of size 4×4 and uses a stride of 2. After this follows a fully connected layer with 256 nodes. Finally, the output layer has 6 nodes, corresponding to all possible actions in the Space Invaders environment. Different Atari environments have different number of available actions, and the output layer always has one node for each action, including a node for doing nothing. Following all hidden layers, a Rectifier Linear Unit (ReLU) activation function is applied, as described in equation (20). This network has 677 686 parameters in total. The larger network in [3] contains an additional convolutional layer and a larger fully connected layer.

$$\text{ReLU} \quad x_i = \begin{cases} x_i & \text{if } x_i \geq 0 \\ 0 & \text{if } x_i < 0 \end{cases} \tag{20}$$

There is no mention of bias or intercepts in the papers, but this network uses a bias value for each filter in the convolutional layers, and each node in the fully connected layers. These values are added to the incoming value of a node, and indicate how large the incoming values are required to be to activate the nodes in the layer. They are initialized with the same process as the weights in the network.

Initialization The initialization functions are described in the sections on the DQN and the ACROMUSE genetic algorithm. Shortly explained, the DQN uses a truncated normal distribution with mean 0, where the standard deviation depends on the number of nodes in the input layer. The ACROMUSE uses uniform initialization in the range $\langle -1, 1 \rangle$.

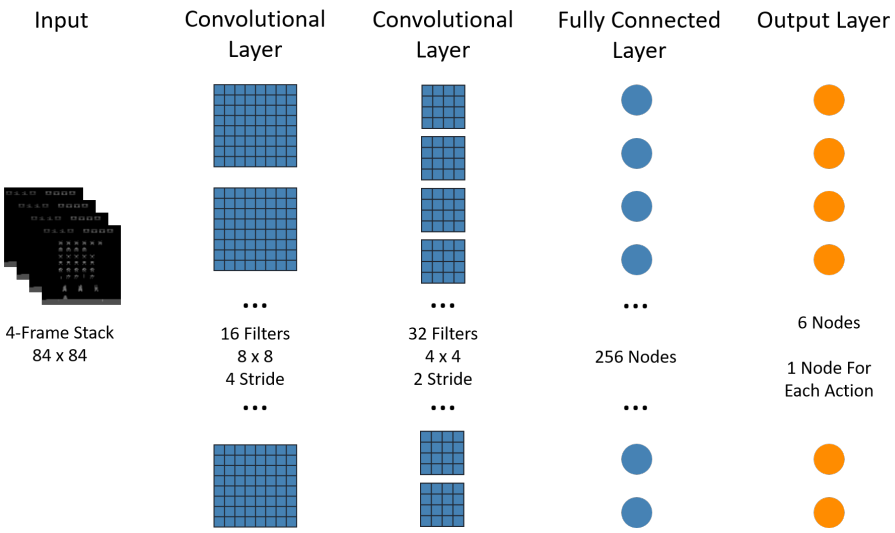


Fig. 2: Network structure. A ReLU activation function is used for the hidden layers.

7 Experiment Set-Up

7.1 The environment

To limit the scope of this thesis, a single Atari environment was chosen. The goal is that the algorithm implementations should generalize for other Atari games, but only one environment is used due to time limitations. The environment was chosen based on the results in papers developing models for playing Atari games[2][3][25][26], with the somewhat arbitrary criteria that it should perform fairly well for all methods described in the papers, and not outstandingly for any single method. The preprocessing of the environment follows [3].

There are two specifications of the environment, evaluation and DQN training. The differences between these are described further below. While the evaluation environment is used for both methods, the DQN training environment is only used for collecting experience trajectories that are stored in the experience replay buffer. The environment used in the code implementation is OpenAI Gym's Atari environment, but wrappers included in TF-Agents are used for preprocessing. Some modifications where done to these code wrappers for the purposes of this thesis, which are discussed in appendix C.

Description of the environment Almost all the Atari games based on the Stella emulator, which includes ALE and OpenAI Gym's implementations, are deterministic[1]. As explained in the follow-up paper on the ALE, [8], this means that given a fixed set of actions, the outcome in the environment will always be the same, as long as the initial state is the same. This can be exploited by the agents by outputting a fixed set of actions instead of reacting to the environment state. To avoid this, some randomness needs to be injected, which is described later in this section.

In the Space Invaders environment, the agent controls a player model at the bottom of the screen. Refer to the unprocessed frame in figure 3a to see how the environment is rendered visually. This player model only has the ability to move horizontally within the outer boundaries and can also fire projectiles directly upwards on the screen. These degrees of freedom gives a total of six actions, listed in table 1 together with the input value associated with them in the environment.

Above the player model are three destructible barricades evenly spaced on the screen. These will absorb projectiles from both the player and the enemies and will gradually erode as they are hit. The player can hide behind these barriers to avoid getting hit until they eventually break but has to fire projectiles while exposed to hit the enemy units.

In the middle of the screen is a grid formation of enemy units. These enemy units will intermittently fire shots at the player. An enemy unit is destroyed if hit by a single projectile from the player. There are 6 rows of 6 enemy units, making a total of 36 enemies on the screen at the start.

Table 1: Actions the agent can take in the environment

Action	Description
0	Do nothing
1	Fire projectile
2	Move Right
3	Move Left
4	Fire Projectile + Move Right
5	Fire Projectile + Move Left

Scoring The agent can score in the environment by firing projectiles at the enemy units. Occasionally, a highly valuable bonus enemy will move across the top of the screen, but due to a high movement speed and the other enemies being in the way, this is hard to hit. The points awarded for destroying the different enemy types are given in table 2. Enemies are ranked by rows, with higher row enemies giving more points.

Table 2: Enemy point values

Enemy Type	Point Value
Row 1	5
Row 2	10
Row 3	15
Row 4	20
Row 5	25
Row 6	30
Bonus	200

Game progression The entire enemy grid formation will move from one boundary to the other at even speed, and reverse direction once a boundary is reached. Each time a boundary is reached, the grid moves a step closer to the ground and player model. Once the grid reaches the barricades, the barricades disappear, regardless of their state of decay. When only a few enemies are left on the grid, they speed up, making them significantly more difficult to destroy.

If all enemy units on the board are destroyed, the entire environment resets, except for score and remaining lives which are carried over. A new enemy grid appears and the barricades are renewed. One iteration of this can be called a level, a term typically used in video games.

Interestingly, this causes the environment to get progressively more difficult, with a large spike at the end of one level, only to become significantly easier when a new level is started. It is difficult for an agent to overcome this final challenge, but if it does, it can easily rack up a much larger score.

End condition of game At the start of the game, the player is given three lives. The agent loses lives when it is hit by enemy projectiles. Once all three lives run out, the game episode ends. Another condition that can end the game is if any of the enemy units reach the ground where the player model sits. This instantly ends the game regardless of remaining lives.

Evaluation environment The evaluation environment specification is used to evaluate agents for both the DQN and ACROMUSE approach. The evaluation environment runs until all three allotted lives are lost. There is no score clipping, so the full score is rewarded based on the agent's actions within the environment.

DQN training environment The DQN training environment has everything in common with the evaluation environment, with a few specific features that sets it apart to aid in training efficiently. An important difference is that an episode run in the environment ends when a single life is lost, as opposed to after a certain number of available lives are lost. This is done to help the agent learn to avoid taking damage and losing a life in the environment. Second, the scores from the environment are clipped, so that they all lie within the range $[-1, 1]$. Any values outside of this range is clipped to the highest value. This is done to make the model work for different environments with widely different ways of scoring and size of scores. Since we only use a single environment to test the models in this thesis, this is not strictly necessary. However, the implemented code can now be more easily used for other environments, by only changing the name in the configuration files. Also, while [3] explains that the value of the learning rate was determined simply using an informal search on some of the games, it is still determined using environments where the score was clipped. Because of this, this learning rate might not be appropriate if the unclipped score is used in training. In the end, it is useful that the model is similar to the original papers for easy comparison, to check that the model works as expected.

While training, the DQN model varies the epsilon from $\epsilon = 1.0$ to $\epsilon = 0.1$ linearly the first 1 million steps of training. After this it is kept at $\epsilon = 0.1$. As a reminder, epsilon here refers to the probability of performing a random action instead of the action determined by the policy/network.

Introducing stochasticity into evaluation Both algorithms use a value for epsilon of $\epsilon = 0.05$ during evaluation, as in [3]. In [8], this is described as perhaps having too much of an impact on the agent policy, but that it also has the positive of being equivalent to the ϵ-greedy approach used for collecting experience.

As in [3], 30 no-operation actions are performed at the start of the environment, for both training and evaluation. This is actually inconsequential for Space Invaders, since a frozen screen appears at the start of every game, displaying the number of lives left, with the player model flashing. This screen lasts slightly longer than these 30 no-op actions are performed, and so the agents start with the same environment state every time. Since this thesis wishes to compare with [2] and [3], this is still implemented and might have an effect in other games, if the code is used to run these. This variation in impact for the initial no-ops actions is listed as a negative in [8].

7.2 Preprocessing

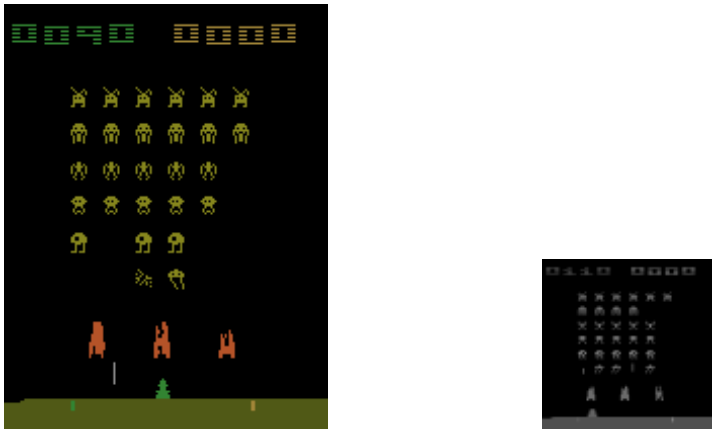
Figure 3 shows a single unprocessed frame and a processed frame side by side. First, the frame is simply fetched from the Atari environment in greyscale. This reduces the dimensions of the frame, using values between 0 and 255 to represent each pixel, instead of three separate color channels as in the RGB representation. The original frame has size 210×160 which is downsampled to 84×84. The frames in figure 3 are to scale.

Figure 4 shows how single preprocessed frames are stacked together. The model in [3] uses frame-skipping, listed as skipping four frames. In practice, however, in a sequence of four frames output by the environment, the first two are skipped entirely, but the last two are max pooled to give a single frame. Max pooling here simply means that for each pixel, pick the value from the two frames that is highest. Each of these four frame sequences produces a single frame, and four of these frames are stacked together. This means that each stack represents a sequence of 16 frames output by the environment. The stack has dimensions $84 \times 84 \times 4$ and is used as input to the neural network. A new stack is formed for every frame, which means that consecutive stacks have an overlap of three frames, and every frame belongs in four stacks. This is shown in figure 5 and because TF-Agents saves every stack separately, there is a lot of redundancy. This is discussed further in appendix C.1.

7.3 Evaluation

Evaluation is performed the same way for both methods. For DQN, evaluation is performed after a set number of steps and defines an epoch. Following the epoch length in [2], evaluation is performed every 50 000 training steps. As a reminder, a training step consists of four experience collection steps and one gradient descent step, as described in section 4.1. Evaluation in ACROMUSE happens at the end of every generation and uses the generation elite agent, meaning the agent that scored the highest.

The evaluation is run in the evaluation environment, which means a full episode runs until the agent has no lives left, and the full score from the environment is returned without clipping. To evaluate an agent, the environment is run over and over until the set evaluation length of 10 000 steps have passed. During this, the scores from every completed episode is recorded. The evaluation



(a) Unprocessed frame. (b) Preprocessed frame.

Fig. 3: Comparison of game frames, to scale, with and without preprocessing.

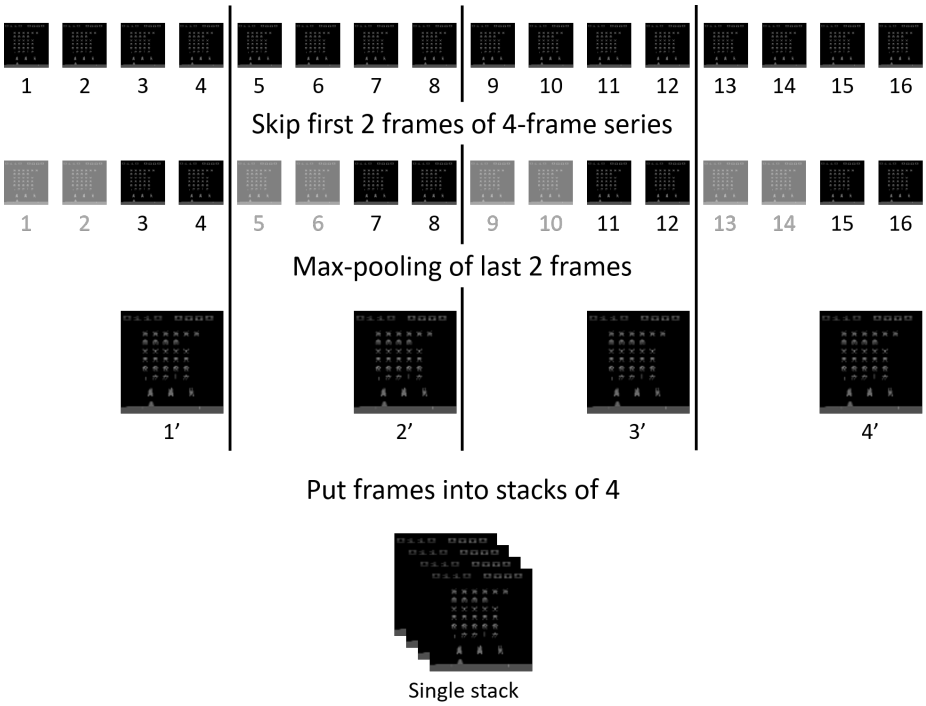


Fig. 4: Illustration of how frames are skipped and stacked. A single stack is used as input to the network.

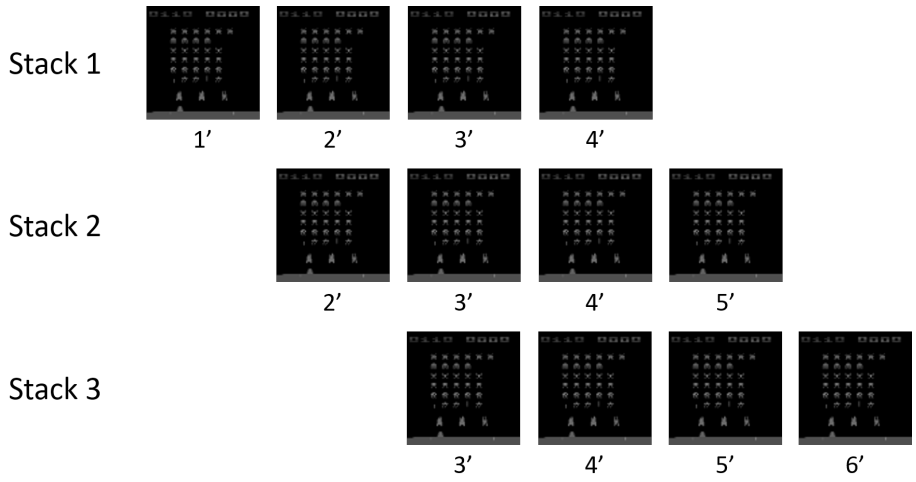


Fig. 5: Stacks of frames overlap, meaning every frame belongs in 4 stacks.

score is the average of the episode scores, but the maximum episode score is also logged since this is listed in [2].

7.4 Hardware

All training and evaluating was performed on a personal computer. The hardware specifications are described in Appendix B. The DQN model makes use of the GPU to accelerate the gradient descent. For the genetic algorithm approach, there is no way to make use of the GPU to speed things up with the framework used for this report.

The reason for using a personal computer is practical, as gaining access to high-performance equipment proved difficult and/or expensive. However, this provides the opportunity to evaluate these approaches on the sort of equipment that is widely available and relatively inexpensive. While training the models will take longer, it should still provide a useful comparison as far as the relative performance of the models.

8 Results

The results presented in this section are the best performing model iterations found during training. In the tables reporting the results, the epoch or generation at which the best model occurs is listed. This may not be the model achieved at the end of training, but for a method that converges, it will typically be a model iteration close to the end of training. M. C. Machado, M. G. Bellemare, E. Talvitie, *et al.*[8] explains the downside of this being that it does not show the stability of training. Hopefully, the figures showing evaluation scores over time will give an indication of this aspect.

8.1 DQN baseline results

The scores for the DQN baseline training is listed in table 3 and the progression of evaluation scores during training is shown in figure 6. We can see that the best agent appears late in training, at epoch 96 out of 100. Considering the scores, the agent trained here achieves essentially the same scores as reported in [2], which uses the same epoch length and number of total epochs. That paper does not show any graph of the progression during training for the Space Invaders environment. From the graph in figure 6, there is not a strong trend showing convergence, but there is a slight trend towards better evaluation scores. It is also clear that there is a high degree of variability in the DQN training process, something noted in [8], and also clear in figure 2 in [2]. The peaks of the graph, meaning the best performing agents do seem to get better over time. A longer period of training would reveal whether these improvements continue or if a plateau is reached.

Agent behavior When visually inspecting the agent play the game, it seems the agent has learned a strategy of moving to and staying around the middle barricade, while moving around to avoid enemy projectiles, and firing intermittently. Later on in the game level, once the barricades are gone and the enemy units are getting closer, the agent does not really alter its behavior. This may indicate that the agent has not had enough experience with these later scenarios. The agent only seems to react to enemies close above it, following these for a short period of time. Staying in the middle and opening up the enemy grid might be a more successful strategy for hitting bonus enemies, which it seems to do in some episodes, including during a deterministic run with $\epsilon = 0$. The behavior does not appear particularly 'intelligent', and the patterns of behavior seem very simple.

8.2 ACROMUSE results and comparison

The ACROMUSE genetic algorithm training was run until the time used by the DQN algorithm was surpassed. In this way, the comparison is based on what the algorithms can achieve within the same period of time. The scores for the

Table 3: DQN baseline scores showing epoch with best average and best episode score.

Training Epoch	Average Score	Best Episode Score
96	**553.33**	**1150.00**

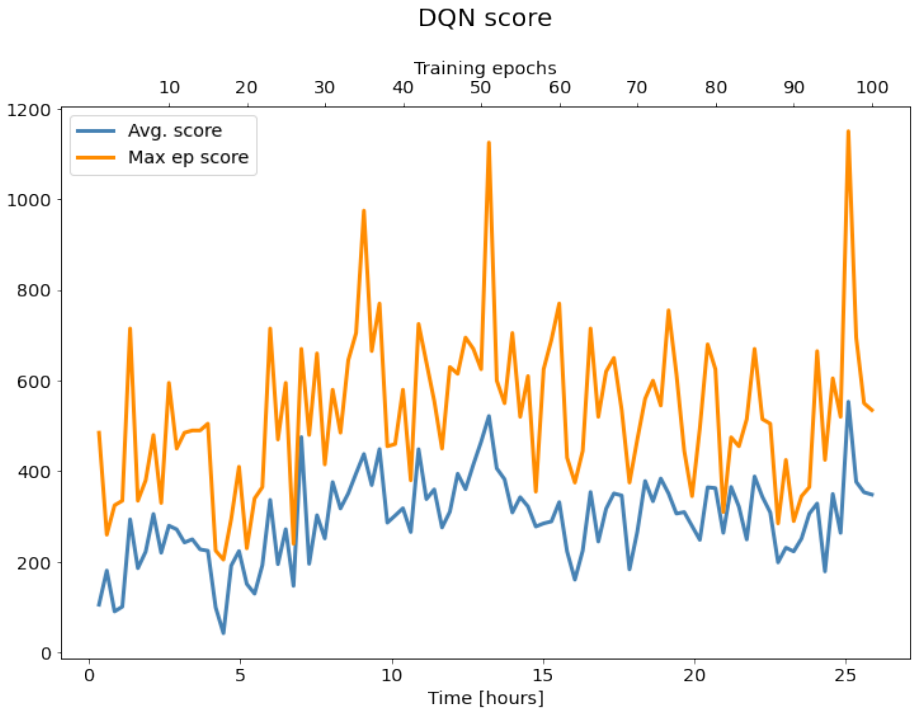


Fig. 6: Training progression for DQN. Average scores and best episode scores against time and epochs.

ACROMUSE approach is listed in table 4 and shown in figure 7. From the scores and figure, there does not seem to be any trend of improvement shown. The best agent occurs early, in generation 5 out of 40. Here, a sharp spike in maximum episode score occurs, which could indicate an increase in performance, or simply a coincidence, such as a lucky hit of a bonus enemy. The low average score indicates that this is an anomaly in a small number of episodes rather than high performance.

Table 4: ACROMUSE scores showing generation with best average and best episode score.

Training Generation	Average Score	Best Episode Score
5	**411.67**	**935.00**

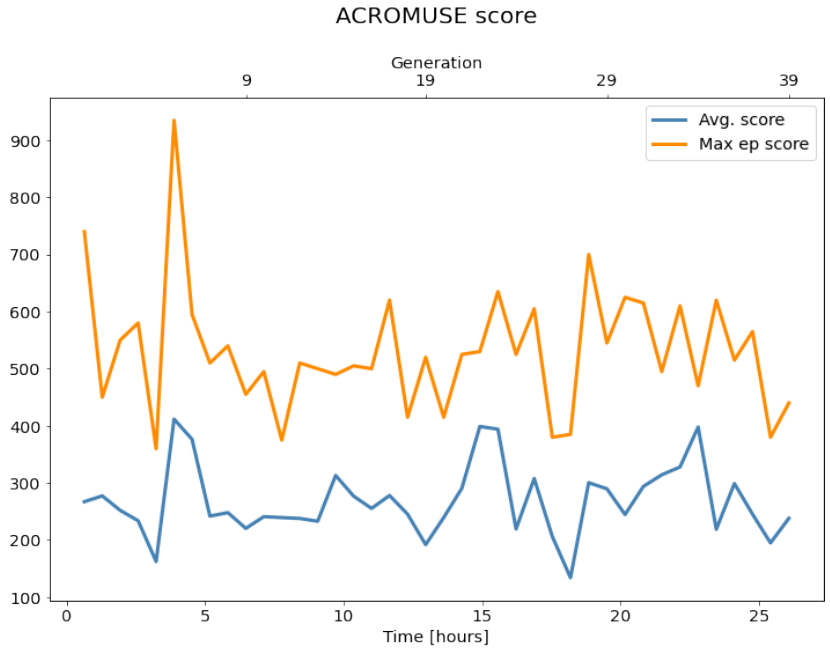


Fig. 7: Training progression for ACROMUSE. Displaying average score and maximum episode score for each generation's elite.

Looking at the SPD and HPD measures shown in figure 8, they decrease rapidly to start, and hover around 0.47 which is slightly above the maximum SPD value in [4]. The HPD value follows SPD closely, and is not lower like indicated in the paper. This could be because no proper local optima are found. From the average generational fitness, shown in the same figure 8, it is apparent that the average estimated fitness increases. This is because there are fewer terrible agents than in the initial generations.

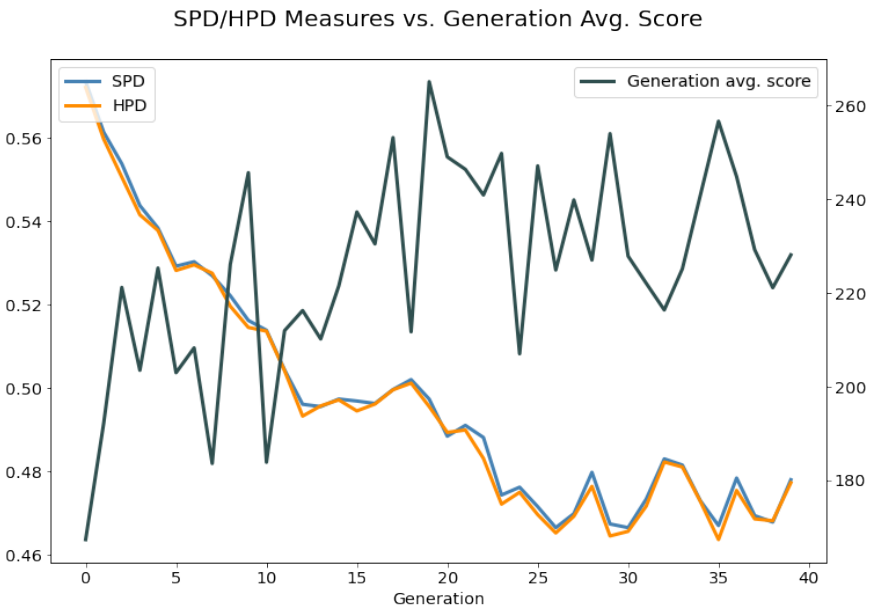


Fig. 8: Values for SPD and HPD during training. Also plotted is the average fitness score for all agents in each generation.

Based on these results, the ACROMUSE genetic algorithm implemented here and with these hyperparameters, does not seem to reach the performance of the DQN model. While the average generation score clearly increases, the elite agent score does not seem to improve from generation to generation.

Agent behavior The best performing elite agent stands still for the most part, occasionally moving across the screen. It fires intermittently, supplying some points. In the end it sticks to one side and seems to be able to hit the bonus enemy in some runs, which explains its relatively high score. The general behavior does not seem advanced and no clear 'strategy' can really be seen.

Because the best agent was found so early, the behavior of the last elite agent, in generation 39, was also inspected to see if there were any positive changes.

Here, the agent just stands still on the left side and fires. This will net some points as the grid passes over and an entire column can be cleared. All in all, there does not seem to be any interesting behavior on display from these agents. When investigating agents from other, relatively high scoring generations, the behavior was the same, standing still and firing on the left side.

8.3 Larger frame skip for DQN training.

An interesting mistake was made during an earlier run of training the DQN model, causing a slight discrepancy from the baseline DQN model. Since the result might be of interest, it is included here. TF-Agents[15] uses the OpenAI Gym[6] Atari environments, which themselves are based on the Arcade Learning Environment (ALE)[1]. OpenAI has several versions of the environments, and since TF-Agents implements frame skipping in its preprocessing, the right Atari environment to use is the 'NoFrameskip' variant. Instead, however, the 'Deterministic' variant was chosen since this is described as being the same as in [3]. This variant also implements its own frame skipping, usually four, but set to three for Space Invaders as it is in [2]. This means frame skipping is implemented twice. The environment only returns every third frame to the TF-Agents preprocessing, where four frames are skipped, in the way described in section 7.2. This means that in effect twelve frames are skipped, where the max pooling happens over frame number 9 and 12.

The evaluation results from the best average scoring and best episode scoring agents during the 100 epochs and within the baseline DQN time frame, are given in table 5. The training progression is shown in figure 9 against the number of epochs. Note that this training took a slightly longer time. In figure 10, the average scores for this model and the baseline DQN is compared. Here the time is limited to what the baseline DQN model used.

This model converged faster than when skipping four frames, and scores markedly better than the score listed for DQN in table 1 in [2].

Agent behavior These agents appear to be of a much higher caliber than the agents trained by the other models. Here the agent moves along with the enemy grid, stays around the barricades for cover, times projectiles well and even seems to prioritize targets closer to the ground, especially at the start of the episode. This is the only method where the agent seems to have a proper 'strategy' that seems reminiscent of what a human player might do. This is true both for the best agent within the baseline DQN time frame and the full 100 epochs. It is also clear that the behavior degrades slightly as the episode progresses.

If these agents are run in an environment with a frame skip of four, like the other results, the performance degrades drastically. This is to be expected as the timing and spread of the frames fed to the network is completely changed. A node which would trigger a certain behavior in the environment which it was trained in, might trigger at a wrong time, or not trigger at all with a different frame skip.

Table 5: DQN with 12 frame skip.

Within baseline DQN time frame

Training Epoch	Best Average Score	Best Episode Score
80	**800.42**	1600.00
67	782.39	**2025.00**

Full 100 epoch training

Training Epoch	Best Average Score	Best Episode Score
89	**878.96**	1435.00
98	817.75	**2460.00**

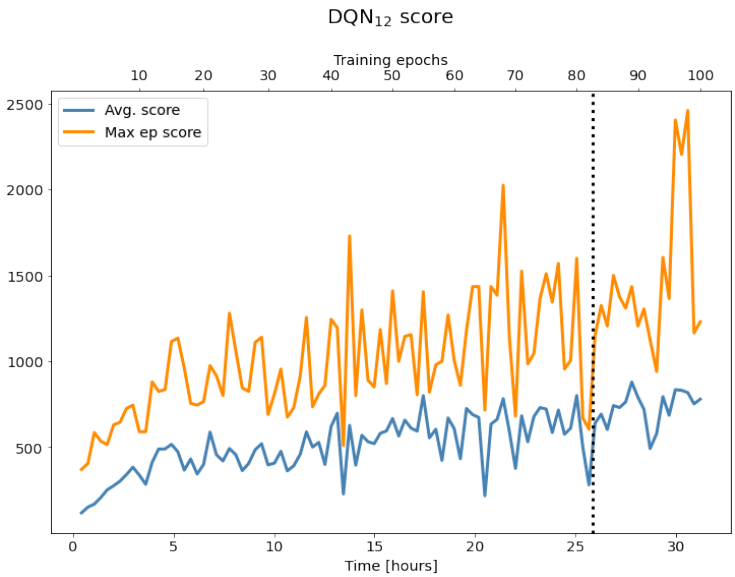


Fig. 9: Training progression for DQN with 12 frame skip. Note the dotted line indicating the time when the baseline DQN was finished training.

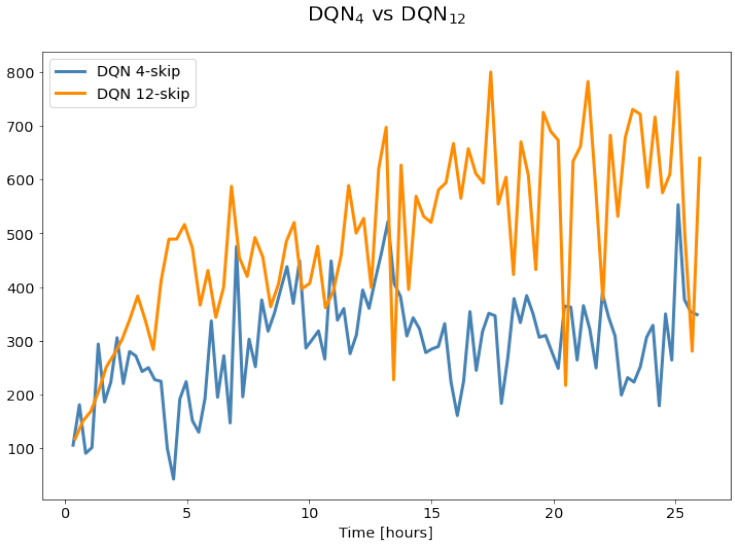


Fig. 10: Comparison of DQN with 12 and 4 frame skips. Average scores are plotted against time trained.

8.4 Discussion

Deep Q-Network approach The results from the DQN approach seems to live up to [2], but it is difficult to say for certain since they only report the single best agents score. The results from training is not particularly impressive, and seems very unstable, with only certain peaks reaching high scores. Even then, the agent's behavior does not seem to follow a particularly coherent strategy. The approach presented in the 2015 paper [3], seems to be more robust, and clearly show that the agent is converging, the longer training proceeds. It makes sense that [3] is the baseline in most papers, due to the higher training quality. However, since the model was to be implemented and tested on the same hardware as the ACROMUSE genetic algorithm in this thesis, the less complex and time-consuming approach in [2] was chosen.

It is interesting that changing the number of frames skipped, as described in section 8.3, makes such a big difference. This may suggest that such a DQN model, at least for certain environments, could benefit from skipping more frames than the four used in [3]. If little information is lost when doing this, it could allow the model to, as stated in [2], play the equivalent of more games in a shorter amount of time and converge faster. In [41], A. Braylan, M. Hollenbeck, E. Meyerson, *et al.* found that the number of frames skipped impacted performance greatly for their version of a genetic algorithm. In some cases, including Space Invaders, a large number of skipped frames increased performance. This would have to be tested further, and since this model is supposed to work generally for the Atari games, on several different environments.

The difference between the four and twelve frame skip training also highlights how sensitive these algorithms are to different hyperparameters and environment preprocessing. If changing the frame skipping from four to twelve makes such a difference, one can only imagine the multitude of possibilities there are for all algorithms that are available, and what an extensive task it is to optimize these to unveil their true potential. Even a paper like [3], achieving good results, having many people working on it and access to high performance hardware, simply used an informal search to set hyperparameters, due to the computational cost.

ACROMUSE genetic algorithm approach It is natural to compare the genetic algorithm approach here to the paper "Deep Neuroevolution: Genetic Algorithms Are a Competitive Alternative for Training Deep Neural Networks for Reinforcement Learning"[25], since they use a simple genetic algorithm, with only mutation and truncated selection. In that regard, the ACROMUSE is a more complex algorithm, using both crossover and tournament selection, not to mention adapting the crossover rate, mutation rate and tournament size depending on the population diversity and fitness. Despite [25] not training in the Space Invaders environment, it is interesting to see what reasons there might be why the ACROMUSE genetic algorithm implemented here fails, while the simple genetic algorithm in [25] succeeds, at least in several environments.

One problem that is immediately obvious is that [25] maintains 1 000 agents per population. For the implementation of ACROMUSE used here, only 1 600 agents are created throughout the entire training process. This difference in scale is a big concern, especially when so many network parameters are involved. F. P. Such, V. Madhavan, E. Conti, *et al.* states that it was assumed that genetic algorithms, such as the one implemented here, would fail when dealing with this number of network parameters[25]. The big innovation that was made in that paper is the ability to distribute the testing of the agents in the generations. Encoding the networks allows for compressing them, and so they can easily be passed back and forth between processors for testing. Also, the training was sped up by letting the environment take advantage of GPUs for running. Although there is no guarantee that the ACROMUSE algorithm would work within these environments if the training was distributed, the number of agents significantly increased and fitness evaluation time decreased drastically, this does appear to be a necessary condition.

Based on the improving average generation score and the visual inspection of agent behavior, it seems that while the average score of the agents in the generations increase as training proceeds, the agents might be trapped in a local optimum of sorts. The reason this seems to be the case is that the exhibited behavior of standing still and intermittently firing is the same, both in the early best performing agent, and the elites in the later generations. Unfortunately, the local optimum is a very poor one. A typical very low performing policy that might be removed early is one that would cause the agent to stand still and never fire projectiles. While simple, short-sighted strategies, such as simply firing intermittently would quickly win out. The algorithm seems unable to progress

beyond this and the diversity of behavior seems very low. This is slightly unexpected as ACROMUSE is made to adapt and maintain a diverse population. It might be the case that the large number of genes in the network, means that the diversity is high, despite the differences being inconsequential and leading to similar, low-performance behavior. Also, the HPD measure is based on relative fitness, which means that poor agents receive a high HPD score despite performing poorly compared to agents trained with other methods. In this way, agents that perform 'well', relatively speaking, are preserved, when in fact, they are very poorly performing agents.

It is not clear whether any improvements in the agents, or useful information in the networks, is preserved since a high-performing agent never appeared. An interesting experiment could be to introduce a high-performing agent trained with a different method to the population and check whether the information is preserved or quickly lost.

9 Conclusion

It seems the DQN model was implemented successfully although the performance during training was quite unstable. This is likely the cost that is paid to reduce training time and complexity. It was shown that using a frame skip of twelve significantly increased the quality of training and it was achieved within a reasonable time frame using modest equipment.

The ACROMUSE approach did not converge, and it seems that the number of agents is simply too small, the search space too large, the network too complex, and the computational cost too high. It is hard to determine if any simple change in hyperparameter, simpler network, heavier preprocessing, or other modifications could make the approach able to train agents for these Atari environments. This does not seem likely, since the genetic algorithms shown to work for this purpose has made use of distributed fitness evaluation to accelerate the training process. The conclusion for this implementation and equipment is that the ACROMUSE genetic algorithm performed worse than the established DQN model in the Space Invaders environment.

9.1 Future work

For the DQN algorithm, used as a baseline in this thesis, it is interesting to see how the frame skipping affected the training. It seems there might still be performance to be found using DQN and its many variations, by simply optimizing hyperparameters, network structure and environment preprocessing. This is not the most interesting field of research, but it is likely to produce results, if a systematic approach is taken. The question would be if any optimizations would generalize or just be applicable to the specific environment used in training and evaluation.

For the ACROMUSE algorithm, it seems like there is somewhat of a hard limit due to how resource intensive a larger population would be to run, unless

a distributed approach like [25] could be implemented. Possibly, testing with a large frame skip could be interesting. Also, there is a slim possibility that the algorithm might perform better in other environments, but this also seems less than likely as the problems when training in Space Invaders should carry over to other environments.

F. P. Such, V. Madhavan, E. Conti, *et al.* in [25] predict that improvements could be made to their simple genetic algorithm by making use of more advanced methods, such as crossover, indirect encoding, LSTMs, regularization, and dropout, to name a few. It seems that the success of [25] is due to the distributed training, and the issue would be to find a way to implement these methods, while still compressing the network enough to transfer them between processor units. This seems difficult and certainly outside the scope of a thesis like this one, but large potential could be unlocked if an efficient approach to this could be found.

General progress in the area of adaptive algorithms is also very interesting. If adaptive algorithms that perform well in many different tasks, whether genetic or otherwise, is developed, this could reduce the computational cost affiliated with determining good hyperparameters massively. However, it still seems likely that different algorithms with different hyperparameters will perform well at different tasks, at least for the foreseeable future.

References

[1] M. G. Bellemare, Y. Naddaf, J. Veness, and M. Bowling, "The arcade learning environment: An evaluation platform for general agents," *Journal of Artificial Intelligence Research*, vol. 47, pp. 253–279, Jun. 2013, ISSN: 1076-9757. DOI: 10.1613/jair.3912. [Online]. Available: http://dx.doi.org/10.1613/jair.3912.

[2] V. Mnih, K. Kavukcuoglu, D. Silver, A. Graves, I. Antonoglou, D. Wierstra, and M. Riedmiller, "Playing atari with deep reinforcement learning," 2013. arXiv: 1312.5602 [cs.LG].

[3] V. Mnih, K. Kavukcuoglu, D. Silver, A. A. Rusu, J. Veness, M. G. Bellemare, A. Graves, M. Riedmiller, A. K. Fidjeland, G. Ostrovski, S. Petersen, C. Beattie, A. Sadik, I. Antonoglou, H. King, D. Kumaran, D. Wierstra, S. Legg, and D. Hassabis, "Human-level control through deep reinforcement learning," *Nature*, vol. 518, pp. 529–533, 2015. DOI: https://doi.org/10.1038/nature14236.

[4] B. McGinley, J. Maher, C. O'Riordan, and F. Morgan, "Maintaining healthy population diversity using adaptive crossover, mutation, and selection," *IEEE Transactions on Evolutionary Computation*, vol. 15, no. 5, pp. 692–714, 2011. DOI: 10.1109/TEVC.2010.2046173.

[5] OpenAI. (2021). "Gym is a toolkit for developing and comparing reinforcement learning algorithms," [Online]. Available: https://gym.openai.com/envs/#atari.

[6] G. Brockman, V. Cheung, L. Pettersson, J. Schneider, J. Schulman, J. Tang, and W. Zaremba, "Openai gym," 2016. arXiv: 1606.01540 [cs.LG].

[7] T. Salimans, J. Ho, X. Chen, S. Sidor, and I. Sutskever, "Evolution strategies as a scalable alternative to reinforcement learning," 2017. arXiv: 1703.03864 [stat.ML].

[8] M. C. Machado, M. G. Bellemare, E. Talvitie, J. Veness, M. Hausknecht, and M. Bowling, "Revisiting the arcade learning environment: Evaluation protocols and open problems for general agents," 2017. arXiv: 1709.06009 [cs.LG].

[9] M. Hausknecht and P. Stone, "The impact of determinism on learning atari 2600 games," in *AAAI Workshop on Learning for General Competency in Video Games*, Austin, Texas, USA, Jan. 2015.

[10] C. J. C. H. Watkins, "Learning from delayed rewards," Ph.D. dissertation, King's College, Cambridge, 1989.

[11] C. J. C. H. Watkins and P. Dayan, "Q-learning," *Machine Learning*, vol. 8, pp. 279–292, 1992.

[12] M. Bellemare, J. Veness, and M. Bowling, "Investigating contingency awareness using atari 2600 games," 2012. [Online]. Available: https://www.aaai.org/ocs/index.php/AAAI/AAAI12/paper/view/5162/5493.

[13] H. van Hasselt, A. Guez, and D. Silver, "Deep reinforcement learning with double q-learning," 2015. arXiv: 1509.06461 [cs.LG].

[14] L.-J. Lin, "Reinforcement learning for robots using neural networks," UMI Order No. GAX93-22750, Ph.D. dissertation, USA, 1992.

[15] D. Hafner, J. Davidson, and V. Vanhoucke, "Tensorflow agents: Efficient batched reinforcement learning in tensorflow," 2018. arXiv: 1709.02878 [cs.LG].

[16] TensorFlow. (2021). "Tf-agents: Agents is a library for reinforcement learning in tensorflow," [Online]. Available: https://www.tensorflow.org/agents.

[17] T. Schaul, J. Quan, I. Antonoglou, and D. Silver, "Prioritized experience replay," 2016. arXiv: 1511.05952 [cs.LG].

[18] H. Y. Ong, K. Chavez, and A. Hong, "Distributed deep q-learning," 2015. arXiv: 1508.04186 [cs.LG].

[19] A. Nair, P. Srinivasan, S. Blackwell, C. Alcicek, R. Fearon, A. D. Maria, V. Panneershelvam, M. Suleyman, C. Beattie, S. Petersen, S. Legg, V. Mnih, K. Kavukcuoglu, and D. Silver, "Massively parallel methods for deep reinforcement learning," 2015. arXiv: 1507.04296 [cs.LG].

[20] M. Hausknecht and P. Stone, "Deep recurrent q-learning for partially observable mdps," 2017. arXiv: 1507.06527 [cs.LG].

[21] V. Mnih, A. P. Badia, M. Mirza, A. Graves, T. P. Lillicrap, T. Harley, D. Silver, and K. Kavukcuoglu, "Asynchronous methods for deep reinforcement learning," 2016. arXiv: 1602.01783 [cs.LG].

[22] M. Jaderberg, V. Mnih, W. M. Czarnecki, T. Schaul, J. Z. Leibo, D. Silver, and K. Kavukcuoglu, "Reinforcement learning with unsupervised auxiliary tasks," 2016. arXiv: 1611.05397 [cs.LG].

[23] J. Martin, S. N. Sasikumar, T. Everitt, and M. Hutter, "Count-based exploration in feature space for reinforcement learning," 2017. arXiv: 1706.08090 [cs.AI].

[24] K. De Jong, "Learning with genetic algorithms: An overview," *Machine Learning*, vol. 3, pp. 121–138, Oct. 1988. DOI: 10.1007/BF00113894.

[25] F. P. Such, V. Madhavan, E. Conti, J. Lehman, K. O. Stanley, and J. Clune, "Deep neuroevolution: Genetic algorithms are a competitive alternative for training deep neural networks for reinforcement learning," 2018. arXiv: 1712.06567 [cs.NE].

[26] M. Hausknecht, J. Lehman, R. Miikkulainen, and P. Stone, "A neuroevolution approach to general atari game playing," *Computational Intelligence and AI in Games, IEEE Transactions on*, vol. 6, pp. 355–366, Dec. 2014. DOI: 10.1109/TCIAIG.2013.2294713.

[27] M. Hausknecht, P. Khandelwal, R. Miikkulainen, and P. Stone, "Hyperneat-ggp: A hyperneat-based atari general game player," in *GECCO '12*, 2012.

[28] K. O. Stanley, D. B. D'Ambrosio, and J. Gauci, "A hypercube-based encoding for evolving large-scale neural networks," *Artificial Life*, vol. 15, no. 2, pp. 185–212, 2009. DOI: 10.1162/artl.2009.15.2.15202.

[29] K. O. Stanley and R. Miikkulainen, "Evolving neural networks through augmenting topologies," eng, *Evolutionary computation*, vol. 10, no. 2, pp. 99–127, 2002, ISSN: 1530-9304.

[30] A. Eiben and S. Smit, "Parameter tuning for configuring and analyzing evolutionary algorithms," *Swarm and Evolutionary Computation*, vol. 1, pp. 19–31, Mar. 2011. DOI: 10.1016/j.swevo.2011.02.001.

[31] A. E. Eiben, R. Hinterding, and Z. Michalewicz, "Parameter control in evolutionary algorithms," *IEEE Transactions on Evolutionary Computation*, vol. 3, no. 2, pp. 124–141, 1999. DOI: 10.1109/4235.771166.

[32] G. Karafotias, M. Hoogendoorn, and A. E. Eiben, "Parameter control in evolutionary algorithms: Trends and challenges," *IEEE Transactions on Evolutionary Computation*, vol. 19, no. 2, pp. 167–187, 2015. DOI: 10.1109/TEVC.2014.2308294.

[33] R. Hinterding, Z. Michalewicz, and T. C. Peachey, "Self-adaptive genetic algorithm for numeric functions," in *Parallel Problem Solving from Nature — PPSN IV*, H.-M. Voigt, W. Ebeling, I. Rechenberg, and H.-P. Schwefel, Eds., Berlin, Heidelberg: Springer Berlin Heidelberg, 1996, pp. 420–429, ISBN: 978-3-540-70668-7.

[34] F. Herrera and M. Lozano, "Adaptation of genetic algorithm parameters based on fuzzy logic controllers," Jun. 1996.

[35] R. Ursem, "Diversity-guided evolutionary algorithms," vol. 2439, Oct. 2002, ISBN: 978-3-540-44139-7. DOI: 10.1007/3-540-45712-7_45.

[36] H. Shimodaira, "Dcga: A diversity control oriented genetic algorithm," in *Second International Conference On Genetic Algorithms In Engineering Systems: Innovations And Applications*, 1997, pp. 444–449. DOI: 10.1049/cp:19971221.

[37] X. Marsault, "A multiobjective and interactive genetic algorithm to optimize the building form in early design stages," in *Proceedings of BS2013: 13th International Conference of the International Building Performance Simulation Association*, E. Wurtz, Ed., 2013, pp. 809–816, ISBN: 978-2-7466-6294-0.

[38] P. J. Huber, "Robust Estimation of a Location Parameter," *The Annals of Mathematical Statistics*, vol. 35, no. 1, pp. 73–101, 1964. DOI: 10.1214/aoms/1177703732. [Online]. Available: https://doi.org/10.1214/aoms/1177703732.

[39] TensorFlow. (2021). "Tf.keras.initializers.variancescaling," [Online]. Available: https://www.tensorflow.org/api_docs/python/tf/keras/initializers/VarianceScaling.

[40] K. He, X. Zhang, S. Ren, and J. Sun, "Delving deep into rectifiers: Surpassing human-level performance on imagenet classification," 2015. arXiv: 1502.01852 [cs.CV].

[41] A. Braylan, M. Hollenbeck, E. Meyerson, and R. Miikkulainen, "Frame skip is a powerful parameter for learning to play atari," Jan. 2015.

Appendices

A Hyperparameters

A.1 DQN hyperparameters

The hyperparameters used for training an agent with the DQN reinforcement learning algorithm is presented in table A.1.

Table A.1: Parameters for DQN training

Learning Rate	0.00025	Learning rate used in RMSProp
Total Steps	5 000 000	Total number of steps to train
Discount	0.99	Discount used by the DQN agent
Target Update	10 000	Number of steps between target network update
Collect Steps	4	Number of experience collecting steps taken per training step
Replay Buffer Size	100 000	Number of frame stacks in replay buffer
Initial Collect Steps	50 000	Number of steps taken to initially populate replay buffer
Initial Epsilon	1.0	Initial epsilon value for collecting experience
Final Epsilon	0.1	Final epsilon value for collecting experience
Epsilon Variation Steps	1 000 000	Number of steps to reduce epsilon from 1.0 to 0.1
Batch Size	32	Size of minibatch used for training
RMSProp Momentum	0.0	Momentum used by RMSProp optimizer
RMSProp Decay	0.95	Decay used by RMSProp optimizer
Evaluation Epsilon	0.05	Epsilon used during evaluation
Evaluation Interval	50 000	Steps between evaluations
$N_{\text{Eval Steps}}$	10 000	Number of evaluation steps

A.2 ACROMUSE hyperparameters

The hyperparameters used for training agents with the ACROMUSE genetic algorithm are presented in table A.2. The ACROMUSE training was stopped once the DQN training time was reached.

Table A.2: Parameters for ACROMUSE training

Agents	40	Number of agents per generation
Rank Steps	2 000	Number of steps used to estimate agent fitness
Evaluation Steps	10 000	Number of evaluation steps
Elite Evaluated Agents	1	Number of most fit agents to evaluate to determine elite
Evaluation Epsilon	0.05	Epsilon used during evaluation
Weight Minval	-1	Lower bound of network weight value
Weight Maxval	1	Upper bound of network weight value
Buffer Value	10^{-6}	Buffer value added in case of lower bound weight value
$P_{mut,exploit}$	0.01	Mutation rate used for exploitation agents
K_1	0.4	Lower bound of crossover rate
K_2	0.8	Upper bound of crossover rate
K_{mut}	0.5	Upper bound on exploration mutation rate
SPD_{max}	0.4	Initial maximum SPD value
HPD_{max}	0.3	Initial maximum HPD value

B Equipment Specification

All training and evaluation was performed on a personal computer. The computer components were primarily bought in 2015 with expanded memory and storage added later, using exactly the same components. It is to be expected that a newer computer with similar types of components would have slightly more performance, but the comparison of the models relative to each other should still hold up. The relevant specifications are listed in table B.1.

Table B.1: Equipment specification used for training and evaluating all models.

OS	Ubuntu 18.04.5 LTS
Language	Python 3.7.4
Framework	TensorFlow 2.4.1
CUDA	Version 10.0
cuDNN	Version 7.6.5
Processor	Intel Core i5-4690K 4-Core 3.5GHz Base Clock Speed
Memory	32GB DDR3 SDRAM 1866MHz
GPU	Gigabyte Windforce G1 Gaming NVIDIA GeForce GTX970 4096MB GDDR5 Memory 1644 Cores 5.2 NVIDIA Compute Capability
Storage	Samsung 850 EVO 500GB SSD Model nr. MZ-75E500 540/520 MB/s sequential r/w speed SATA 6Gb/s

C Code Implementation

The code is available on GitHub:

`https://github.com/a-nesse/acromuse_atari`

The specific GitHub commit used for training can be found with the commit ID (SHA-1 hash):

`245bfb6dfb2dd47798328c43db4f07497583d199`

The results in section 8.3 was achieved with commit:

`b3d9a03a14c9d37b72100ec67a28a25336172182`

Config files Both methods use simple configuration files, formatted as dictionaries, and handled in the JSON format. An abridged snippet of the config file for the DQN model is shown here:

```
{
"env_name":"SpaceInvaders",
"learning_rate":0.00025,
"num_iterations":5000000,
...
"save_name":"dqn",
"keep_n_models":100
}
```

Environment wrappers TF-Agents has wrappers for the OpenAI Gym Atari environments. However, some of these files had to be changed for the purposes of this thesis, such as the `load` function in `suite_atari`. This function loads the correct Atari environment, with the proper wrappers. If `eval_env=True` is passed to the function (line 9), an evaluation environment is loaded, by changing the default training environment wrapper to an evaluation environment wrapper in line 15-16. In line 13, the dtype was changed from `np.uint8` to `np.float32`. This was necessary for the Keras based network to use the output from the environment. This is slightly strange, since Keras is part of TensorFlow. In line 18-21, the environment version name is set to the correct "SpaceInvadersNoFrameskip-v4" in the case of using the Space Invaders environment. This means no frame skipping or repeated actions happening at the environment level, but rather in TF-Agents' preprocessing.

```
def load(
        environment_name: Text,
        discount: types.Int = 1.0,
        max_episode_steps: Optional[types.Int] = None,
        gym_env_wrappers: Sequence[
            types.GymEnvWrapper] =
    ↪ TRAIN_ATARI_GYM_WRAPPERS_WITH_STACKING,
```

```
        env_wrappers: Sequence[types.PyEnvWrapper] = (),
        spec_dtype_map: Optional[Dict[gym.Space, np.dtype]] =
    ↪  None,
        eval_env: bool = False
) -> py_environment.PyEnvironment:
    """Loads the selected environment and wraps it with the
    ↪ specified wrappers."""
    if spec_dtype_map is None:
        spec_dtype_map = {gym.spaces.Box: np.float32}

    if eval_env:
        gym_env_wrappers =
    ↪ EVAL_ATARI_GYM_WRAPPERS_WITH_STACKING

    environment_name = game(
        name=environment_name,
        mode='NoFrameskip',
        version='v4')

...
```

There are two versions of the `atari_preprocessing` script, one for DQN training and one for evaluation. Below is the first part of the `reset` function, which wraps the environment's own `reset`. This function is the same for both training and evaluation environment. It has been edited to perform a `fire` action after a reset for certain environments which require this before starting. Also, up to 30 no-operation actions are taken at the start. The exact number is chosen at random, and in line 14, this number is multiplied by the number of frames skipped. This is to simulate the action being taken by an agent, which only sees one frame for every four frames in the environment, and performs one action that will be repeated for the next four frames. For Space Invaders, this has no influence, however, it is implemented in case the code is used for other games. The frame returned from the environment is processed and returned from the `reset` function.

```
def reset(self) -> np.ndarray:
    """Resets the environment.
    Returns:
            observation: numpy array, the initial observation
    ↪  emitted by the environment.
    """
    self.env.reset()
    # executing 'fire' step
    if self.env.game in ['breakout', 'beam_rider']:
        self.env.step(1)
    # implemented a no-op start, equivalent to 30 no-op
    ↪ actions
    # this is inconsequential for SpaceInvaders, but is left
    ↪ in for other games
    noops = np.random.randint(0, 31)
```

```
13      # multiply by frame_skip to emulate actions
14      noops = noops * self.frame_skip
15      for _ in range(noops):
16          _, _, done, _ = self.env.step(0)
17          if done:
18              self.env.reset()
19      self.lives = self.env.ale.lives()
20      self.game_over = False
21      self._fetch_grayscale_observation(self.screen_buffer[0])
22      self.screen_buffer[1].fill(0)
23      return self._pool_and_resize()
```

The training environment also clips the rewards returned from the environment in the manner shown below. The code used for [3] seems to clip it, rather than simply set it to -1, 0 or 1. Since all returned rewards for Space Invaders are zero or positive integers, this does not matter for our case, as all positive rewards are set to 1. The evaluation environment has no reward clipping.

```
1 # clipping rewards between -1, 1 for training env
2 reward = 1.0 if reward > 1.0 else reward
3 reward = -1.0 if reward < -1.0 else reward
```

C.1 The AtariDQN class

The DQN model is implemented with the TF-Agents library for reinforcement learning, by TensorFlow. During training, the algorithm makes use of a training and an evaluation environment, which are both initialized in the constructor. As can be seen in line 2 and 3 below, the only difference is the input parameter `eval_env` which is set to `False` for training environments and `True` for evaluation environments. Also notice how the python environment that is loaded has to be wrapped by the `TFPyEnvironment` class to function in the TensorFlow framework.

```
1 self.train_py_env = suite_atari.load(environment_name=self.
    ↪ env_name, eval_env=False)
2 self.eval_py_env = suite_atari.load(environment_name=self.
    ↪ env_name, eval_env=True)
3 self.train_env = tf_py_environment.TFPyEnvironment(
    ↪ self.train_py_env)
4 self.eval_env = tf_py_environment.TFPyEnvironment(
    ↪ self.eval_py_env)
```

The optimizer specified in [2] and [3] is the RMSProp and shown below is how the TensorFlow version of this is initialized. This is then passed to the DQN agent. Notice how many of the parameters are fetched directly from the imported config. Any parameters that will be used in other methods in the class, are declared as object attributes.

```
1 self.optimizer = tf.compat.v1.train.RMSPropOptimizer(
2     learning_rate=self.dqn_conf['learning_rate'],
```

```
    momentum=self.dqn_conf['momentum'],
    decay=self.dqn_conf['decay'],
    epsilon=self.dqn_conf['mom_epsilon'])

...

self.agent = dqn_agent.DqnAgent(
    self.step_spec,
    self.action_spec,
    q_network=self.q_net,
    optimizer=self.optimizer,
    emit_log_probability=True,
    td_errors_loss_fn=common.element_wise_huber_loss,
    epsilon_greedy=1.0,
    target_update_period=self.target_update,
    gamma=self.dqn_conf['discount'])
self.agent.initialize()
```

Since the epsilon ϵ will be varied for the DQN model during training, a separate evaluation policy with a fixed epsilon $\epsilon = 0.01$ is defined:

```
self.eval_policy = epsilon_greedy_policy.EpsilonGreedyPolicy(
    policy=self.agent.policy,
    epsilon=self.dqn_conf['eval_epsilon'])
```

TF-Agents also has functions for maintaining a replay buffer, saving checkpoints for this, which can be used when restarting training after a stop, and a dynamic driver, that can take a set number of steps in the environment and save the experienced trajectories to the experience replay buffer. Something to point out is that the experience replay buffer actually is quite inefficient, since it will save all frames in stacks, and there is no functionality to save individual frames and collect them into stacks as they are being sampled. How the stacks overlap is shown in figure 5. This means one frame will appear in four stacks. This, along with limited memory, is why a replay buffer size of only 100 000 was used. TF-Agents does have the `PyHashedReplayBuffer` class which saves experience in a much more efficient way, without so much redundancy, by saving frames individually. However, it is not straightforward, if possible, to use this with the DQN class, and there was no time to create such a solution.

```
self.replay_buffer = tf_uniform_replay_buffer.
    ↪ TFUniformReplayBuffer(
    data_spec=self.agent.collect_data_spec,
    batch_size=self.train_env.batch_size,
    max_length=self.replay_buffer_max_length)

self.replay_ckp = common.Checkpointer(
    ckpt_dir=os.path.join(
        os.getcwd(), 'saved_models_dqn', self.save_name + '
    ↪ replay'),
    max_to_keep=1,
```

```
    replay_buffer=self.replay_buffer)

# initializing dynamic step driver
self.driver = dynamic_step_driver.DynamicStepDriver(
    self.train_env,
    self.agent.collect_policy,
    observers=[self.replay_buffer.add_batch],
    num_steps=self.collect_steps_per_iteration)
```

Below is how the epsilon is changed from $\epsilon = 1.0$ to $\epsilon = 0.1$ for collecting experience over the first 1 million experience collecting steps. Notice the frames being calculated in line 5 is the number of training steps multiplied by the number of experience collecting steps, this is because for every training step, four experience collect steps are performed. Once 1 million collect steps have been passed, the epsilon is set permanently at $\epsilon = 0.1$ as `exploration_finished` is set to `True`.

```
exploration_finished = False

...

frames = int(step*self.collect_steps_per_iteration)
# changing epsilon linearly from frames 0 to 1 mill, down to
    ↪ 0.1
if frames <= self.final_exploration:
    scaled_epsilon = self.initial_epsilon - (0.9*frames/
    ↪   self.final_exploration)
    self.agent.collect_policy._epsilon = max(
        self.final_epsilon, scaled_epsilon)
elif not exploration_finished:
    self.agent.collect_policy._epsilon = self.final_epsilon
    exploration_finished = True
```

C.2 The AtariAcromuse class

The AtariAcromuse class runs training with the ACROMUSE algorithm, and tests the agents in the environment. Along with this, the SPD and HPD measures are calculated here. First, the average agent and the average fitness-weighted agents must be found. The function `zero_net` returns a zero-value network with the correct dimensions. The Euclidean distance is used for this, where the fitness-weighted agent uses the proportional fitness of each agent instead of dividing by the number of agents.

```
def zero_net(self):
    zero_net = []
    for layer in self.net_shape:
        zero_net.append(np.zeros(layer))
    return np.array(zero_net,dtype=object)

```

```
def _find_avg_agent(self):
    total_fit = np.sum(self.scores)
    spd_sum = self.zero_net()
    hpd_sum = self.zero_net()
    weights = []
    for i, agt in enumerate(self.agents):
        spd_sum += agt.get_scaled_weights()
        w_i = self.scores[i]/total_fit
        weights.append(w_i)
        hpd_sum += w_i*agt.get_scaled_weights()
    self.weights = weights
    self.spd_avg = spd_sum/len(self.agents)
    self.hpd_avg = hpd_sum
```

The function for calculating SPD follow the equations given in section 5.1. Note that the `spd_max` value is reset at the bottom in case the SPD value exceeds the previous max value.

```
def _calc_spd(self):
    gene_sum = self.zero_net()
    for agt in self.agents:
        gene_sum += (agt.get_scaled_weights()-self.spd_avg)
    ↪ **2
    std_gene = self._arr_sqrt(gene_sum/self.n_agents)
    spd = self._arr_sum(std_gene/self.spd_avg)/self.n_weights
    self.spd = spd
    if spd > self.spd_max and self.adaptive_measures:
        self.spd_max = spd
```

It is similar for HPD, but here the individual HPD contribution is calculated at the same time. The weights for the agents were calculated in the `_find_avg_agent`-function listed above.

```
def _calc_hpd(self):
    self.hpd_contrib = np.zeros(self.n_agents)
    weighted_gene_sum = self.zero_net()
    for i, agt in enumerate(self.agents):
        sq_diff = (agt.get_scaled_weights()-self.hpd_avg)**2
        self.hpd_contrib[i] = self.weights[i]*np.sqrt(self.
    ↪ _arr_sum(sq_diff))
        weighted_gene_sum += self.weights[i]*sq_diff
    w_std_gene = self._arr_sqrt(weighted_gene_sum)
    hpd = self._arr_sum(w_std_gene/self.hpd_avg)/self.
    ↪ n_weights
    self.hpd = hpd
    if hpd > self.hpd_max and self.adaptive_measures:
        self.hpd_max = hpd
```

Below are the functions that finally calculate the adaptive parameters according to the equations in 5.6. Note that if the SPD/HPD value is above the maximum value, it is clipped to this value. If the SPD/HPD value is set to a

variable maximum value, updating in case of higher values, this has no effect. This can be disabled in the config to use the default SPD/HPD maximum values from [4]. The crossover rate is calculated using SPD and the fixed values K_1 and K_2. The mutation probabilities are actually calculated in the AtariGen class described under, but the `_calc_p_mut_fit`-function returns a list of the P_m^{fit} values for each agent. This is the P_m^{fit} in equation (17) and (18) in section 5.6. The P_m^{div} is strictly based on SPD. The tournament size is calculated based on the HPD value for the generation.

```
def _calc_pc(self):
    spd_lim = self.spd_max if self.spd>self.spd_max else
    ↪    self.spd
    return ((spd_lim/self.spd_max)*(self.k2_pc-self.k1_pc))+
    ↪    self.k1_pc

def _calc_p_mut_fit(self):
    p_muts = []
    f_max = np.max(self.scores)
    f_min = np.min(self.scores)
    for score in self.scores:
        p_muts.append(self.k_p_mut*((f_max-score)/(f_max-
    ↪    f_min)))
    return p_muts

def calc_measures(self):
    self._find_avg_agent()
    self._calc_spd()
    self._calc_hpd()
    p_c = self._calc_pc()
    spd_lim = self.spd_max if self.spd>self.spd_max else
    ↪    self.spd
    hpd_lim = self.hpd_max if self.hpd>self.hpd_max else
    ↪    self.hpd
    p_mut_div = ((self.spd_max-spd_lim)/self.spd_max)*
    ↪    self.k_p_mut
    p_mut_fit = self._calc_p_mut_fit()
    tour_size = math.ceil((hpd_lim/self.hpd_max)*
    ↪    self.t_size_max)
    return p_c, p_mut_div, p_mut_fit, tour_size
```

The weights for Keras networks are given as arrays arranged in a list, which is impractical. In this script the list is converted to an array of objects, where the objects are NumPy arrays of differing dimensions. It is possible to add, subtract, multiply and so on, with these object arrays, regardless of the differing dimensions. However, it is not possible to sum or take the square root of the values in one operation. Because of this, functions were written that deal with this.

```
def _arr_sum(self,arr):
    tot_sum = 0
```

```
    for a in arr:
        tot_sum += np.sum(a)
    return tot_sum

def _arr_sqrt(self,arr):
    nw = []
    for a in arr:
        nw.append(np.sqrt(a))
    return np.array(nw,dtype=object)
```

The AtariGen class The AtariGen class contains the methods used for evolving a new generation. AtariAcromuse creates an AtariGen object and passes the parent generation to it, along with the crossover rate, mutation rates, tournament size and so on. The AtariGen object returns the offspring generation.

The tournament function performs tournament selection by selecting random agents from the population, and selecting the highest performing agent. However, as is prescribed in ACROMUSE, the HPD contribution is used instead of the fitness score to select the parent(s).

```
def _tournament(self,probs,n,size):
    participants = np.random.choice(
        self.n_agents,
        size=size,
        replace=False)
    winners = np.argpartition(probs[participants], -n)[-n:]
    return participants[winners]
```

Crossover is done by a simple uniform probability of choosing genes or weights from either parent. In the code, a simple boolean array is initialized along with an inverse array. These are used to add values from the parents together.

```
def _uniform(self, arr1, arr2):
    sel1 = np.random.randint(0,2,arr1.shape,dtype=bool)
    sel2 = ~sel1
    return (arr1*sel1) + (arr2*sel2)
```

Mutation is done similarly, by selecting genes to be mutated with probability `p_mut` and re-initializing values for these. Using boolean arrays again, these re-initialized weights are added with the untouched old weights.

```
def _mutate(self,arr,p_mut):
    mut = np.random.random_sample(arr.shape)<p_mut
    no_mut = ~mut
    mut_val = np.random.uniform(low=self.minval,
    ↪ high=self.maxval,size=arr.shape)
    return (no_mut*arr) + (mut*mut_val)
```

The AtariNet class The AtariNet class is used by the AtariAcromuse class, along with the demonstration script, to emulate the Deep Q-Network used in

AtariDQN. Both of these networks are built up from the same config file, and have the same structure. The class acts as a wrapper for a Keras Sequential class, and builds up the convolutional and fully connected layers in the constructor. There is a method for returning the shifted and scaled weights, as described in section 5.4.

```
def get_scaled_weights(self):
    span = self.maxval-self.minval
    return (self.get_weights()-self.minval)/span
```

The `set_weights`-method is also made to check for any lower bound values before setting the weights in the network. This should not be a problem in theory, but this did occur during earlier testing, so this was implemented to make sure this does not happen. The time spent between generations is minuscule as compared to the testing of the agents either way. The cause for lower bound values appearing is not known.

```
def set_weights(self,weights):
    for i,layer in enumerate(weights):
        #checking for any values equal to minval
        if np.any(layer==self.minval):
            weights[i]=np.where(weights[i]==self.minval,
    ↪     self.replace_min,weights[i])
    super().set_weights(list(weights))
```

The `action`-function takes an observation as input, and predicts an action based on a forward pass through the network weights. It can also perform a random action with probability epsilon. The function returns the index for the action node with the highest value.

```
def action(self, observation, epsilon=0):
    if epsilon and epsilon>np.random.rand():
        return np.random.randint(self.action_shape)
    activations = super().predict(observation.observation)
    return np.argmax(activations)
```