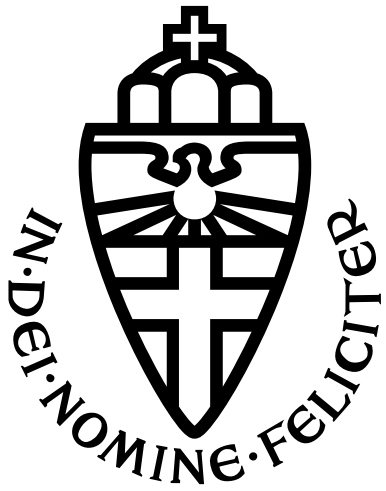

Natural Computing
Learning to play MsPacman with Evolutionary Strategies

Marius M. Kästingschäfer
s3058301

Lieuwe Meijdam
s1047496

Djesse Dirckx
s1046968

Abstract

In this study an altered version of the World Model was implemented on top of the OpenAI gym version of MsPacman. This altered World Model consisted of a variation of the original RNN, because the VAE was replaced by a different low dimension latent space representation. The goal of the study was to test the ability of several natural computing algorithms to learn optimal parameters for the controller component which navigates the agent. The chosen algorithms consisted of a sequence mutation algorithm, genetic algorithm, and covariance matrix adaption evolutionary strategy. These different natural computing algorithms were evaluated in terms of score, complexity and learned behavior, under different hyperparameter configurations. After this analysis it was found out that the best configuration in terms of average score was achieved by the CMA-ES which only used state information as input. However, the configuration with a CMA-ES which also utilised temporal information input, did result in lower score but was superior in terms of behavior since it showed more human-like behavior and displayed awareness of its surroundings and the behavior of adversary agents. The RNN (temporal information) does add significant extra computational complexity to this configuration, but it is believed that the shown behavior and potential of the method (the method was still improving and did not reach convergence yet) outweigh this.

Word count: 6991

Contents

1 Introduction

Designing agents that are able to maximize reward within a certain environment is a non-trivial task. This is especially true if the agent must discover the most suitable actions on its own. In this research, several natural computing algorithms are investigated for optimising an agent to maximise reward within the game MsPacman. We use the OpenAI gym [1] version of MsPacman, since this provides an easy-to-use platform that can get integrated inside a development setup without additional setup. It was chosen to use MsPacman because on one hand it provides a straightforward objective and game-play where on the other hand it is known for its high state complexity due to the complicated behavioral patterns of game dynamics.

As a starting point, the World Model architecture proposed in [2] is implemented. They utilize a variational auto-encoder(VAE) and recurrent neural network(RNN) to obtain an informative game representation. Using this representation, the model that controls the agent is optimized. Similar to the work in[2], it is investigated if the CMA-ES algorithm [3] is capable of fulfilling this task. In comparison a genetic algorithm is also used to optimise the controller. These algorithms are provided with either solely the VAE state representation or a combination with the RNN temporal information to investigate how this influences behavior and performance. Experiments using different hyper parameters combinations are conducted in search for an optimal configuration. These algorithms are interesting for several reasons. Genetic algorithms are a widely used gradient-free optimization technique that have proved effective within different domains [4]. A wide variety of possible parameters exists to tweak and increase the algorithms performance[5], which makes them interesting for our research. The use of CMA-ES is interesting since it is more robust than comparable evolutionary strategies [6] and has been used to solve reinforcement learning problems before [7][8]. As baselines, an agent that chooses each action at random and a sequence mutation algorithm that tries to find an optimal sequence of actions are used. Both baselines are unaware of the game state.

For this study a select amount of goals has been setup:

1. Implementing a model influenced by the World Model architecture to operate on MsPacman.
2. Implementing CMA-ES for optimization of the controller parameters and comparing this with other methods such as Sequence mutation and GA.
3. Measuring the impact of variables such as population size, noise levels and the amount of model parameters (weights) on model performance and practicalities such as time complexity.
4. Analyzing the displayed behavior of the best model configurations to discover if it acts in an intelligent way.

These goals will be chronologically explained throughout the report. All source code and the graphs of results that did not fit in the report can be found in our Github repository[1].

2 Background Theory

2.1 Environment: MsPacman

Before diving into the technical details, first the game setting and its components will be briefly described. This is essential in order to properly analyse and understand the learned behavior later on in this study.

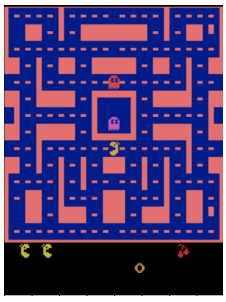

Figure 1: MsPacman Environment

The game setting is that of a human controlled agent (MsPacman) that has to manoeuvre through a maze without getting caught by adversary agents (coloured ghosts) which act according to their own rule-set. However, simply avoiding the adversaries is not the objective of the game. Throughout the maze there are little dots which grant MsPacman "points" when collected. The objective of MsPacman is to obtain as much "points" as possible, which can be seen as maximizing reward. The adversary agents try to prevent MsPacman from collecting dots. When MsPacman and a adversary agent collide MsPacman is moved back to the starting position and one "life" is subtracted (a life represents a ticket to continue without the environment and score being reset). Next to collecting dots MsPacman can earn points by collecting a big dot which triggers a set time-period in which collision between MsPacman and ghosts results in MsPacman eating the ghost and earning extra points (the ghost will be returned to its starting location). When all dots are collected from the environment, the environment will reset (the score is not reset), making it possible to almost infinitely continue the game. The general objective can be summarized as: *"Maximize points and eat all dots whilst avoiding losing lifes"*. All the components described can be seen in Figure 1.

[1] https://github.com/djessedirckx/nc-project

Now that the game setting and objectives are clear we will describe the action space and behavior of the adversarial agents. Starting with the action space of the human player, it can be summarized as 2-dimensional motion. The movement actions can be represented by thinking of a joy-stick, which results in the following discrete action space: Up, Down, Left, Right, Left-up, Left-Down, Right-up, Right-down, No-operation.

When looking into the adversary agents, they act to specific patterns. The adversary agents act according to the possible "modes" they are in. The first mode being the "chase" mode, in which they spend most of their time. During the chase mode they will simply follow MsPacman according to their rule-sets (more on these sets later). The second mode is "scatter" mode, in this mode each of the ghosts will move towards a specific corner of the board and stay there for a set time. The last mode is "Frightened" which is triggered when MsPacman collects a big-dot. The reason for these three movement modes is that the creator of the original Pacman wanted the player to be threatened in "waves" which is simulated by sometimes making the ghosts retreat when in scatter mode.

As mentioned before the ghosts are mostly in chase mode, in this mode their movement is dependent on that of MsPacman. This movement can be roughly summarized in the following behavioral rules (Figure 2 visualises these rule-sets):

- Red ghost (Blinky): Chasing by following the player
- Pink ghost (Pinky): Intercepting by cutting off the player
- Blue ghost (Inky): Intercepting by using both the players orientation and that of Blinky
- Orange ghost (Clyde): Feigning ignorance, alternates by chasing and avoiding

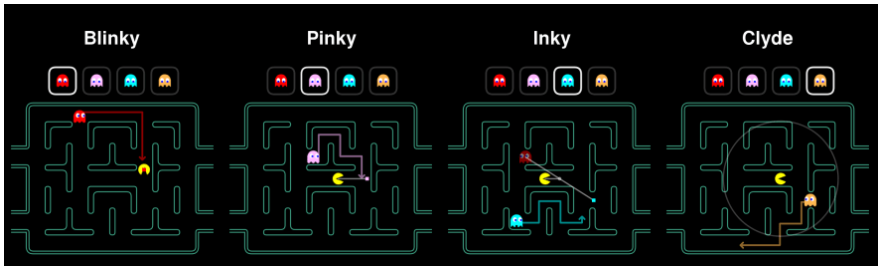

Figure 2: Visual intuition of adversary agent behavior

The environment can be represented in multiple ways. One way is by using the state of the maze as an image, this is the initial plan. An alternative way would be using the so-called "RAM-state", this represents the original 128 bytes of RAM of the original Atari 2600 machine [9]. This RAM-state can be seen as a latent representation of the current state of the game. Since the initial goal is to use a VAE to learn a latent space from the image state this will only be used as a backup-plan.

2.2 Sequence Mutation (SQM)

The Sequence mutation does neither take into account environment states nor recurrent state information. The action sequence is initialised in the following way: $S = [a_1,\ a_2,\ ...,\ a_{n-1},\ a_n]$. Here S denotes the sequence of actions the agent will take and a_i denotes the individual actions. Initially the actions are determined by randomly drawing an action from the set of possible actions. The algorithm to optimise the sequence is the actual sequence mutation (SQM) algorithm. The algorithm applied iteratively changes a certain percentage of actions and reevaluates the sequence afterwards. If the performance of the sequence has improved the mutated sequence becomes the new standard, otherwise the mutations are discarded.

2.3 World Model (WM)

World Model learning is closely related and enabled through the advancements made within the fields of semi-supervised learning and representation learning [10]. The model we are using was originally introduced in 2018 by Ha and Schmidthuber [2]. They also explained this model extensively in this blog post [2]. The architecture of the World Model consists of several important components. The architecture can be seen in Figure 3. The first component is the variational autoencoder [11] [12]. The variational autoencoder (VAE) contains an information bottleneck that

[2] https://worldmodels.github.io

compresses the incoming input data into a lower-dimensional latent space representation. The VAE is also described as the vision module since it contains the agents visual representation of the environment. This latent space representation is then fed into the MDN-RNN. The RNN can be understood as the agents memory module. It compresses what happens over time. The model predicts future states given the current state.

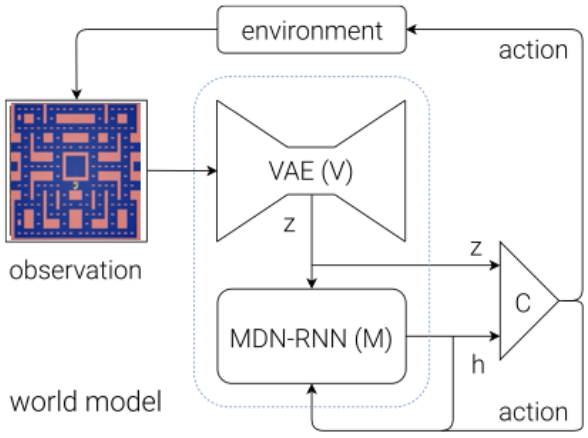

Figure 3: Visualization of the World Model

The component that actually sends inputs (actions) to the environment is the controller. The component contains inputs from both the VAE and the MDN-RNN and generates meaningful actions from them. The agents good representations of past and future states allow the agent to make accurate predictions about the influence of its actions on the environment.

2.4 Controller training

The part of the World Model that chooses the appropriate actions while receiving inputs from both the VAE and the RNN is called the controller. For the controller no gradient information exists, instead natural computing methods are applied to obtain a suitable parameter configuration.

2.4.1 Evolutionary strategies

Evolutionary strategies are a set of optimisation techniques which are inspired by notions of evolution [13]. These techniques often operate iteratively by providing a set of candidate solutions (a generation) on a problem. These solutions are afterwards scored by a so-called fitness function in the actual environment. The best candidate solutions will be recombined and form the next generation, this way of evolving generations will continue until a stopping criterion is met (e.g., the candidates do not improve anymore) [13].

2.4.2 Genetic Algorithm (GA)

The team from OpenAI [8] in their 2017 paper showed that evolutionary strategies can be a viable alternative when compared to standard reinforcement learning algorithms. Evolutionary strategies are sometimes also referred to as black-box optimization methods. Their advantage is that they often work reasonable well even in the absence of gradient information. Candidate solutions are mutated, recombined and a fitness score determines which individuals are taken into the next round. This form of optimisation algorithm is roughly inspired by natural evolution [14]. It therefore utilises certain operations which are also found in biology like mutation, crossover and selection [13]. The process of natural selection revolves around the notion that the “fittest” individuals in a population survive and in turn produces offspring which inherits these strong characteristics from the parents. This notion can be repeated for an arbitrarily long time period where each generation grows fitter. The population has a set size, and the least fit individuals will make place for the new fitter offspring.

There are multiple phases in a genetic algorithm, but the algorithm always starts with an initial population. This population exists out of a set of individuals which all have slightly different characteristics from each other (also known as genes). All characteristics of an individual together form a chromosome, which in the context of GA can be seen as a solution and the characteristics as parameters. However, before continuing with the algorithm we also need to select a fitness function to measure how fit an individual is (this function is task dependent).

Given that we have an initial population and a fitness function the actual algorithm can be started. In the selection phase, the fittest individuals (based on fitness function) are selected and are allowed to pass their genes to the next generation(this is done in pairs forming parents). Next all of the parent pairs are made to create offspring by a process called crossover (which is the process of exchanging genes between parents). These new offspring will then be added back to the population, however each offspring has a slight chance of being altered/ mutated. Mutation is done to preserve diversity within the population and countering premature convergence. This process will be iteratively repeated until a stopping condition is met, which is often when convergence occurs (offspring hardly changes). Extending this idea to this study, the goal would be to find optimal parameters for the controller unit of the pipeline. A fitness function could be the score that is achieved by the agent.

Since learning from pixels directly (210x160x3) would be to expensive and because our VAE was not working as intended we decided to use the RAM state. Since our main interest is to compare models with different complexities we choose the evolutionary strategy to learn a minimal model consisting of a direct state-action mapping. To learn such a model we initialised the weights in a way such that the resulting r value (explained in the formula below) is close to the area we need. For updating the weights we used different noise levels to update the weights. We also experimented with sparse updates onto the weight vector. The resulting action was determined in the following way: r $= \sum_i x_i * w_i$. here both x and w are vectors of length 128 and the resulting scalar is denoted with r. We can now use r to determine the resulting action in the following way:

$$\text{Actions} \begin{cases} \text{Action 1: } a_1 & r > a \text{ and } r <= b \\ \text{Action 2: } a_2 & r > b \text{ and } r <= c \\ \text{...} & \\ \text{Action n: } a_n & r > z \text{ and } r <= z \end{cases} \tag{1}$$

Here $a, b, ..., z$ define predefined cutoff-values. The full model architecture is also depicted in Figure 6 in the Appendix.

2.4.3 Covariance Matrix Adaptation Evolution Strategy (CMA-ES)

In the World Model paper[2], the authors make use of an evolutionary strategy called Covariance Matrix Adaptation Evolutionary Strategy (CMA-ES) to optimize the parameters of the controller model. This algorithm was introduced in 2003 by Hansen et. al [15]. In CMA-ES the offspring is generated by sampling from a multivariate normal distribution $\mathcal{N}(0, \mathbf{C}^{(g)})$. This sampling procedure uses the following equation: $\mathbf{x}_k^{g+1} \sim \mathbf{m}^{(g)} + \sigma^{(g)}\mathcal{N}(0, \mathbf{C}^{(g)})$.

Here $\mathcal{N}(0, \mathbf{C}^{(g)})$ defines the multivariate normal distribution with covariance matrix $\mathbf{C}^{(g)}$ for generation g, $\mathbf{x}_k^{(g+1)}$ represents the offspring (λ) at generation $g+1$, $\mathbf{m}^{(g)}$ is the mean value of the search distribution at generation g and $\sigma^{(g)}$ defines the step-size at generation g. In order to iteratively use this sampling procedure, $\mathbf{m}^{(g+1)}$, $\mathbf{C}^{(g+1)}$ and $\sigma^{(g+1)}$ need to be computed. $\mathbf{m}^{(g+1)}$ is computed by taking a weighted average of selected samples from the current offspring. By assigning different weights to the samples, a selection mechanism in the current offspring is created. The weights are determined using the ranking of function values based on the objective function. This selection mechanism is extended by selecting $< \lambda$ samples which can be seen as truncation selection. For updating the covariance matrix, a so-called rank-1 and rank-μ update are used. The rank-1 update is an average of the changes of $\mathbf{m}$ where the rank-μ estimates the covariances of the weighted offspring. By combining these rules, the covariance matrix can be updated. Finally, $\sigma^{(g+1)}$ is updated using so-called evolution paths which is a series of consecutive steps. If the path is short, these steps cancel each other out (all going in a different direction). This means the step size should be decreased. If the path is long, the steps are pointing in the same direction. This means they can be replaced by a single larger step which means the step size should be increased [3][16]. Using these update rules, it is possible to iteratively proceed to a (local) optimal solution is achieved as one would also do in a gradient descent approach. Generating new offspring is continued until a stopping criterion is met.

The MsPacman environment that is used for this project only provides a positive reward function, which is not ideal given that the goal is to minimise an objective function. Therefore it was chosen to use a custom objective function instead. This is defined as $h - \sum_i r_i$ where h represents the maximum achieved highscore on the OpenAI platform (≈ 6000)[3] and $\sum_i r_i$ the sum of obtained rewards for a single episode.

[3]https://gym.openai.com/evaluations/eval$_k pL9bSsS4GXsYb9HuEfew/$

3 Implementation Details

3.1 Data collection

For both the RNN and the VAE we collect data that originated from the environment. For the data that was needed for the VAE, we simply sampled states from the MsPacman environment. These states were sampled at random, to fetch as many possible state representations. This was needed to ensure that the VAE had examples of almost all possible states, so that it could learn a good latent representation. For the RNN something similar was done, but since the RNN needed to learn the actual time relevant dynamics of the environment here the images were ordered per sample run. The actions at each state were also collected to be able to learn state transitions better. These sample runs were created with a random agent, since the RNN only needs to be able to predict the next state. This was done so that the RNN could learn that MsPacman dies when it runs into a ghost for example.

3.2 World Model components (VAE and RNN)

3.2.1 VAE

The VAE was implemented using Tensorflow [17]. We implemented a preprocessing step to convert all VAE inputs into the required format. The decoder and the encoder where defined separately using predefined convolutional and deconvolutional Keras layers [18]. As loss function we used reconstruction and KL-loss. The resulting loss curves together with the obtained reconstructions were plotted for visual inspection and model fine-tuning. A more detailed description of the VAE and its implementation can be found in the Appendix 7.3. Since the VAE was unable to obtain the required performance we use the RAM state of the game instead of the latent space as compressed state representation in consecutive experiments.

3.2.2 RNN

The World Model paper proposed to integrate a recurrent module into their network that is able to predict the next state and reward. Since complex systems are often stochastic in nature, the authors propose to predict the probability distribution the next state vector is drawn from instead of trying to do a deterministic prediction of the next state [2].

In the original work, this is implemented as follows. The recurrent module is implemented as an LSTM layer. The internal state of this network is used as input for the controller. The LSTM layer is followed by a fully connected layer that serves as the mixture density network that outputs the probability distribution the next state is drawn from. This prediction consists of the parameters of this distribution. As input, the network expects a concatenated vector consisting of the current state representation, the current action and the reward the agent obtained in the previous state. To optimise this network, a loss function is used that takes into account the reconstruction loss of the predicted state and the predicted reward [19].

As a consequence of using the RAM state, the values that parametrize the probability distribution states are drawn from, are not available anymore since this RAM state is simply a 1-D state vector representation. Therefore it was chosen to use a more simplistic version of the recurrent module instead. This module solely consists of a LSTM layer that is optimised to make a prediction of the next state. Even though the authors of the World Model paper stated that this might not be optimal given the stochastic nature of a complex environment, it is interesting to see whether it can still improve the performance of the model. As a loss function, the mean-squared-error between the predicted state and ground truth is used. After training, the model is exported such that it can be integrated into the World Model architecture without retraining. The network was implemented using the Tensorflow machine learning library [17].

3.3 Random agent (Baseline)

The implementation of the random agent was trivial. For MsPacman the agent draws a random action from the action space and performs this action.

3.4 Sequence Mutation (SQM)

The method was implemented using Python's standard library. We followed the formalism provided in the background theory. To improve evaluation stability the evaluation depth was set to 5 and we returned the average across those runs as evaluation score.

3.5 Genetic Algorithm (GA)

The method was implemented using Python's standard library. The pseudocode can be found in Appendix Figure 17. Since we do not use any cross-breading in the algorithm itself the final implementation is probably best referred to as genetic algorithm. The implementation was divided into three separate parts, consisting of an agent evaluation part, a part that determined the current action given the weight configuration and the environment state and a third main module responsible for running the main loop. Within this third part either old weights could be (re-)used as starting point or new weight configurations could be generated. To improve the quality of individual agent mutations we averaged those across 5 games. This decreased fluctuations and improved the chance that a generation used for weight updating is indeed stronger in terms of performance.

3.6 Covariance Matrix Adaptation Evolution Strategy (CMA-ES)

CMA-ES was implemented using the Python *pycma* library[4] that is built and maintained by the original authors of the algorithm. This implementation allows for testing with a variety of population sizes and initial standard deviations. The algorithm optimises the parameters of a controller that consists of a single fully connected layer followed by a softmax non-linearity. Since the evolutionary strategy expects a single vector for optimisation, the weight matrix is flattened and concatenated with its respective bias values. Similar to the GA setup, rewards are averaged over 3 consecutive runs to improve reliability of performance. It was chosen to use 3 instead of 5 runs to reduce running times, given that each training step consists of the evaluation of multiple population individuals.

4 Experimental results

In this section, the setup and results of all experiments is described. For each algorithm, various parameter configurations are tried to investigate how this influences the behavior and the performance of the agent. For all algorithms, it is investigated whether an action space of 4 results in a different performance compared to allowing the complete action space. The algorithm-specific parameter configurations are described in their respective subsections. After optimising for the specified number of training episodes, the average obtained reward over 10 separate test runs using the optimised controller is reported together with the standard deviation. This provides additional insights in the algorithm performance. For the best performing parameter configuration of the genetic algorithm and CMA-ES, it is evaluated whether including the RNN that predicts the next state achieves better performance. In this experiment, it is tested if a more informative input representation also leads to better performance. It is chosen to perform this experiment for the best performing configuration only, given the substantially longer running times when including the RNN. In the final subsection the results of the different algorithms are compared to find out which performed best along with a behavioral analysis.

4.1 World Model components (VAE and RNN)

The testing of the VAE is discussed in detail in Appendix Section 7.3. We were unable to obtain a useful latent space representations from the VAE. We thus used the equally low dimensional RAM state as replacement. The loss curves for the RNN for an action space of both 4 and 9 can be seen in Figure 15 and Figure 16 respectively. Based on these loss curves, it can be stated that the RNN is able to successfully predict the next state. To find out whether the agent can improve its performance by comprehending this information, the hidden state of the RNN is concatenated to the state input. Results of this experiment are described in Section 4.5.

4.2 Random agent (Baseline)

To obtain a valid 'lower-bound' for our experimental results we evaluated a random agent in the environment. For the MsPacman game a mean reward of 170 with a standard deviation of 21 over 1000 games. The highest once obtained reward is 980. The full reward distribution is displayed in Figure 18 of the the Appendix 7.6.

4.3 Sequence Mutation (SQM)

All runs were preformed over 1000 episodes. The best results during the training with an actions space of 4 were obtained for the algorithm with a mutation probability of 0.05 which obtained a score of 840. For 9 actions the algorithm with a mutation probability of 0.2 performed best with a score of 640. During the evaluation run, the algorithm with a mutation probability of 0.1 performed best for 4 and 9 actions. The stability of the obtained scored varied widely,

[4]https://github.com/CMA-ES/pycma

this is also the reason why training and evaluation score differ significantly. The overall training time for each of the algorithms took on average 1 hour. The results obtained are far from optimal which shows the limits of the applied method. Detailed training plots can be found in Section 7.7 of the Appendix.

Results Sequence Mutation Algorithm (SQM)

Model	**Generations**		**Training Score**		**Evaluation Score**	
	Action 4	Action 9	Action 4	Action 9	Action 4	Action 9
Mutation probability 0.05	7	3	**840**	560	177 (24.8)	212 (34.1)
Mutation probability 0.1	5	6	550	600	**210** (46.0)	**277** (63.3)
Mutation probability 0.2	6	5	530	**640**	187 (35.6)	197 (21.7)
Mutation probability 0.3	5	6	530	530	194 (40.0)	251 (37.2)

Table 1: We varied the mutation probability and the action space. Each run was trained for 1000 episodes. All sequence configurations where tested in a final evaluation run.

4.4 Genetic Algorithm (GA)

For the evolutionary strategies we ran several different parameter configurations. All runs were performed over 1000 episodes. We varied the size of the noise applied to each weight generation, tried different starting weights and checked whether applying sparse weight updates to each generation had a positive influence on the model performance. The individual training runs for the best agents are provided within Section 7.8 of the Appendix. An initial challenge was posed by finding weight configurations that aligned well with environment states and lead to both meaningful actions and more importantly consecutive reward signals. Overall it seems like medium weight-size configurations lead to stronger final agents. For almost all noise configurations we find that the weight updates happen during an early stage of the training process. We also observed that the best runs took both actions (up or down) with an equal probability.

Evaluation runs where taken over 10 runs with an evaluation depth of 1, thus a total of 10 evaluation runs per weight configuration. The best evaluation score was obtained using a noise level of 0.8 and sparseness for 9 actions and a noise level of 0.4 without sparseness for an action space of 4. Again the optimal algorithm performance differed between training and evaluation run. The average training time of the algorithms took 0.5 hours per training run. Since the algorithms are very similar, the runtimes almost identical for across the different runs.

Results Genetic Algorithm (GA)

Model	**Generations**		**Training Score**		**Evaluation Score**	
	Action 4	Action 9	Action 4	Action 9	Action 4	Action 9
Noise 0.4	5	7	1400	1100	**848** (202)	953 (03.6)
Noise 0.4 + sparse	7	6	1230	1030	356 (125)	629 (26.0)
Noise 0.8	5	9	1200	930	437 (149)	726 (38.1)
Noise 0.8 + sparse	5	9	**1650**	**1200**	418 (137)	**1057** (32.1)
Noise 1.2	10	3	1210	1100	793 (215)	482 (21.4)
Noise 1.2 + sparse	8	5	1060	1100	435 (154)	396 (23.5)
Noise 0.8 + sparse + RNN	n/a	5	n/a	1318	n/a	744 (143)

Table 2: We varied the amount of noise and whether the weight update was sparse. A single run was trained for 1000 episodes. All weight configurations where tested in a final evaluation run.

4.5 Covariance Matrix Adaptation Evolution Strategy (CMA-ES)

For optimising the controller using the CMA-ES algorithm, different parameters configurations were tried as well. Over several experiments, we tried variations in the population size of each generation, the initial standard deviation of the populations and the allowed action space the agent can predict.

When inspecting the results in Table 3, it can be seen that an action space of 9 leads on average to best performance. This difference in performance was also observed during training, as can be seen in Figure 24 and Figure 25 where the

average moving reward for both algorithms is shown respectively. This can be explained by the fact that this larger action space gives the agent more possibilities in the situations it encounters. However, when inspecting the action distribution of the best performing algorithm for both actions spaces in Figure 26 and Figure 27 respectively, it can be seen that the action space of 9 almost always chooses the same action. For the best performing algorithm of action space of 4, we observe a bit more variation. Based on these distribution graphs, it seems as if the agent is taking the same action most of the times. This would not be possible given the layout of the maze. It therefore seems as if the agent tries to get to a position in the maze where it can get optimal rewards.

When inspecting results for both action spaces in more detail, some interesting observations can be made. Looking at the best result obtained during training for the action space of 4, it looks as if a larger population size also leads to better performance. However, when looking at the evaluation performance it can be seen that population size 10 resulted in a better performance than population size 20. A population size of 5 for this action space seems to small to obtain good performance, as can be seen for both training and evaluation results. When comparing the initial standard deviations, a standard deviation of 0.01 appears to give too little initial variation in solutions to achieve good performance during training. However, during evaluation differences in performance are a bit smaller. For an action space consisting of 9 actions a different pattern is observed. For both training and evaluation results, a population of both 5 and 20 results in better performance compared to population size 10 (except for a few outliers). This is somewhat unexpected, given the results for the action space of 4. Also, smaller initial standard deviations do not lead to worse performance compared to larger initial values. It sometimes even results in better performance.

In the final experiment, the best performing state-only configuration was optimised again. This time, the RNN hidden state prediction is used as well. Since it was observed that this drastically increased running times, it was chosen to run for only 100 epochs instead. This still took $\approx$ 27 hours in total. When inspecting the result in Table 3, it can be seen that this configuration leads to the worst performance. But since it was trained for only 100 epochs, this is no fair comparison. When inspecting the moving average reward in Figure 36, it can be seen that it did not converge yet. It is therefore reasonably to assume that training for a longer time can lead to better performance.

Based on the observations that were made, we can make some conclusions. The most important thing we learned is that allowing the agent to do all possible actions overall leads to best performance. For both action spaces, a different patterns of results is observed. The specific parameter configuration (in terms of population size and initial standard deviation) therefore appears to be action space dependent. When looking at the action distributions, we can see that the best performing agents mostly choose the same action. This is somewhat unexpected, comparing to how a action distribution of a human player would probably look like. The agent seems to try to exploit some pattern where staying in the same position (choosing the same action all the time leads to getting stuck in some position) gives optimal results. This is discussed in more detail in Section 4.7. A final note about the RNN performance. Due to extremely long running times, it unfortunately was not possible to fully analyse this experiment. However, based on the moving average one could speculate that training for a longer time could lead to better performance. More plots for the best performing 4 and 9-action configurations and the configuration that uses both the game state and the RNN hidden state can be found in Section 7.9 of the Appendix.

Results CMA-ES

Model	**Generations**		**Training Scores**		**Evaluation Scores (*std.*)**	
	Action 4	Action 9	Action 4	Action 9	Action 4	Action 9
Pop 5 - std 0.01	8	14	1387	3490	353 (*03.9*)	1807 (*37.5*)
Pop 5 - std 0.1	15	11	1477	2073	536 (*07.2*)	1299 (*16.2*)
Pop 5 - std 0.5	12	9	1913	2753	661 (*05.0*)	1095 (*18.3*)
Pop 10 - std 0.01	12	10	2467	3916	1066 (*19.0*)	1194 (*28.0*)
Pop 10 - std 0.1	11	13	1847	**4870**	1307 (*23.8*)	**2792 (*47.7*)**
Pop 10 - std 0.5	8	11	2187	1857	**1449 (*27.7*)**	1373 (*15.3*)
Pop 20 - std 0.01	17	18	2057	3678	757 (14.0)	1843 (*41.4*)
Pop 20 - std 0.1	15	10	2580	2013	881 (*20.5*)	867 (*07.6*)
Pop 20 - std 0.5	13	16	**3200**	3570	887 (*21.2*)	1854 (*21.0*)
Pop 10 - std 0.1 + RNN	n/a	11	n/a	1700	n/a	937 (*14.0*)

Table 3: CMA-es results using only the game state / game state + RNN hidden state

4.6 Summary of the results

Here we compare the three different optimization algorithms, namely the sequence mutation (SQM), the genetic algorithm (GA) and the Covariance Matrix Adaptation Evolution Strategy (CMA-ES) with a baseline in terms of amount of parameters, their model complexity, training time and the obtained training and evaluation scores.

Results MsPacman

Model Input Size	Parameters	Complexity	Training	Evaluation	Training Time
Baseline	n/a	n/a	n/a	170	n/a
SQM	n/a	very low	600	277	1h
GA (state only)	128	low	1200	1057	0.5h
GA (state + RNN)	384	medium	1318	744	$\approx$ 6h
CMA-ES (state only)	1161	medium	4870	2792	$\approx$ 2.5h
CMA-ES (state + RNN)	3465	high	1700	937	$\approx$ 27h

Table 4: All results where included as described in the previous sections.

4.7 Behavioral analysis

In the previous sections we have taken a statistical approach to analyse the performance of our approaches. However, the ultimate goal of this study was to validate if our natural computing approaches were able to learn behavior that resembles intelligence/ shows some form of strategy whilst playing MsPacman. Therefore, in this section we will describe behavior and patterns in how our agent is interacting with the environment.

The SQM method shows no form of smart behavior and is almost seemingly random. However, after observing the behavior of the CMA-ES and GA trained controller (state only)[5], a clear strategy was observed; the agent always seemed to navigate towards the corners of the environment. When arriving there it will seemingly lost stay there for some time until the ghosts are getting closer. After they are somewhat close the agent eats the big dot and tries to eat the ghosts. On first glance it seemed strange that the agent was not prioritizing eating more small dots, however after some further research it was noted that in the game the player is rewarded 10 points per small dot that MsPacman eats. However, if the players eats a big dot and afterwards eats a ghost it will be able to earn a lot more points. This is due to the fact that the first ghost eaten is worth 200 points, the second 400 points, the third 800 and the fourth 1600 points. This seems to suggest that the agent prioritises to eat ghosts instead of eating small dots. It was even observed that the agent seemed to prefer the upper left corner as well, this is most likely due to the fact that the ghosts are more likely to get close to the agent in this area. The reason for this is that the layout of the maze (and the starting positions of the ghosts) are on the upper part of environment. For reference an example run can be seen in Figure 4.

However, after eating the big dot the player can only eat ghosts for a very short amount of time, and the agent often dies shortly after this time has exceeded. It was observed that the agent often navigated towards a different corner and attempt the same strategy again (until it runs out of lifes). So, the agent learned to use its lives to get moved back to the middle of the environment and use that trick to quickly move to other corners after getting killed by a ghost. Even though this strategy is somewhat smart, it has some major flaws. One flaw being that it is unreliable and has a very high degree of risk, which can be seen in the validation runs as well (showing very varying total scores). The agent often is so fixated on reaching the corner quickly that it pays less attention to the position of the ghosts and often gets intercepted. Another flaw is that the agent often accepts that it will die after the big dot time period exceeds, which results in costing it a life. Also, the reason that the point reward for eating multiple ghosts is so high is that it is a very hard task to eat many of them in a short period of time. The agent also seems not fully aware of the ghosts at times, which might also be caused by the fact that it can sometimes eat them (after collecting a big dot). As noted earlier it often happens that the agent gets intercepted by a ghost when it is on its way to one of the corners, in such moments it was observed that it was not always aware that it should avoid the ghosts. So, summarising the above, the agent did learn some very interesting behavioral patterns and priorities. However, it was not able to fully grasp all of the hidden patterns that MsPacman contains, for example the movement behavior of the ghosts Figure 2.

When observing the CMA-ES which also used the RNN input[6] in its training we noted that it behaved very differently. Instead of directly going for the bigger dots, this agent was navigating with accuracy through the maze targeting the small dots whilst avoiding the ghosts quite well. It was even observed that it caught upon some of the more hidden patterns of the ghosts, for example it seemed to have an understanding that the red ghost always tries to follow

[5] https://youtu.be/pDvYLiPEWgg

[6] https://youtu.be/WevmIJDQK5s

MsPacman and therefore the agent tried to steer clear of that. Similar interactions with the other ghosts were also observed, for example we observed that the agent often gets killed by the orange ghost. From its behavior it can be seen that the agent seems to expect it will always stay clear of MsPacman but since the orange ghost actually feigns ignorance the agent often falls into its trap. Similar observations were made for the GA with the RNN however this agent seemed to be less aware of the ghost movement patterns (e.g., chasing after the red ghost).

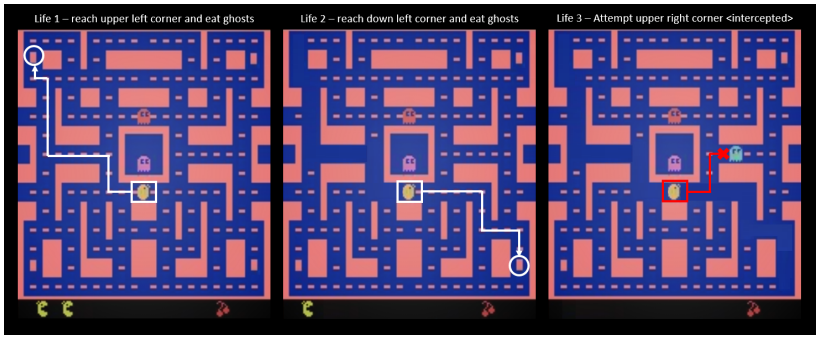

Figure 4: An example run of the GA/CMA-ES (state only) trained MsPacman agent

5 Conclusion

In this study we successfully replicated the original World Model and made it run on the MsPacman OpenAI environment. It was observed however that the VAE component was not performing well and suffered from mode collapse (not learning the small details like agent positions). For this reason the original World Model was altered, instead of using images of the game state the RAM-state was used which also meant that the RNN architecture was altered.

For optimisation CMA-ES, SQM and GA were successfully implemented to solve the environment. The three optimisation schemes differed widely in complexity. We additionally implemented a random agent as a baseline. Through experiments on the MsPacman environment the impact of different hyperparameters on performance were measured. For the SQM algorithm the agent performed worse for very large mutation probabilities (> 0.3). The optimal mutation probability differed between the two possible sizes of the action space and was either 0.05 or 0.2. The GA performed best for medium size weight updates which where applied in a sparse manner. For all results obtained variance differed widely across runs. For the CMA-ES algorithm, it was observed that the optimal parameter configuration (in terms of population size and initial standard deviation) was action space dependent. For an action space of 4, increasing the population size led to a better performance where this was not necessarily the case when using the complete action space (9). When not including the RNN hidden state, the agent managed to get a better evaluation performance compared to when the RNN was included. However, due to long running times when including the RNN it was not possible to make a fair comparison. We do think that including the RNN can lead to better performance in the long run given that the obtained reward kept increasing. Unfortunately we were note able to verify this, but we do think it is interesting for future research. Overall, the CMA-ES can be considered the most stable optimisation algorithm (which can be seen in the smaller variance scores and the more stable training process). In terms of time performance and implementation complexity the GA might outperform the CMA-ES slightly but performance outweighs this advantage.

Lastly when looking at the different models from a behavioral perspective it was observed that a higher average score did not necessarily mean that the agent was acting in a more intelligent way. Both the CMA-ES and GA optimizations learned a strategy which revolved around exploiting a point difference in eating ghosts/ big dots. These agents learned a strategy where it always immediately moved to the corners of the maze to eat a big dot and eat ghosts to inflate its score, but never really attempted to clear all the small dots in the maze (which is part of the actual objective). The CMA-ES and GA that were trained with RNN input both showed more intelligent/accurate movement, where the CMA-ES seemed to be the better method given its superior environmental awareness. It actively tried to eat all small dots whilst avoiding the ghosts (keeping some of their unique movement in mind). However, since the small dots are only worth 10 points each this agent scored a lower average score compared to the agents that used the ghost eating exploit. However, since these agents show more intelligent/ appropriate behavior we believe that this combination is still preferred, even at the cost of increased computational complexity due to the RNN. This also leads to the conclusion that the RNN (temporal information) stimulates more environment aware movement which proves valuable in games like MsPacman.

We believe our results show that evolutionary strategies have great potential in learning how to play MsPacman. In future research our best configuration could be improved upon with more training and would benefit from an increase in resources. We belief that simpler games such as Pong or Pinball could have been fully solved by the algorithms we provided within this report. Verifying such claims would also be interesting for future research.

References

[1] Greg Brockman, Vicki Cheung, Ludwig Pettersson, Jonas Schneider, John Schulman, Jie Tang, and Wojciech Zaremba. Openai gym, 2016. cite arxiv:1606.01540.

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

[3] Nikolaus Hansen. The cma evolution strategy: A tutorial. *arXiv preprint arXiv:1604.00772*, 2016.

[4] M. Dianati, In-Soo Song, and M. Treiber. An introduction to genetic algorithms and evolution. 2002.

[5] Darrell Whitley. An overview of evolutionary algorithms: practical issues and common pitfalls. *Information and Software Technology*, 43(14):817–831, 2001.

[6] N. Hansen. The CMA evolution strategy: a comparing review. In J.A. Lozano, P. Larranaga, I. Inza, and E. Bengoetxea, editors, *Towards a new evolutionary computation. Advances on estimation of distribution algorithms*, pages 75–102. Springer, 2006.

[7] Verena Heidrich-Meisner and Christian Igel. Evolution strategies for direct policy search. In Günter Rudolph, Thomas Jansen, Simon M. Lucas, Carlo Poloni, and Nicola Beume, editors, *PPSN*, volume 5199 of *Lecture Notes in Computer Science*, pages 428–437. Springer, 2008.

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

[9] Greg Brockman, Vicki Cheung, Ludwig Pettersson, Jonas Schneider, John Schulman, Jie Tang, and Wojciech Zaremba. Openai gym, 2016. cite arxiv:1606.01540.

[10] Yoshua Bengio, Aaron Courville, and Pascal Vincent. Representation learning: A review and new perspectives. *IEEE transactions on pattern analysis and machine intelligence*, 35(8):1798–1828, 2013. cite arxiv:1206.5538.

[11] Diederik P. Kingma and Max Welling. Auto-Encoding Variational Bayes. In *2nd International Conference on Learning Representations, ICLR 2014, Banff, AB, Canada, April 14-16, 2014, Conference Track Proceedings*, 2014.

[12] Diederik P. Kingma and Max Welling. An introduction to variational autoencoders. *Foundations and Trends® in Machine Learning*, 12(4):307–392, 2019.

[13] Hans-Georg Beyer and Hans-Paul Schwefel. Evolution strategies –a comprehensive introduction. *Natural Computing: An International Journal*, 1(1):3–52, May 2002.

[14] Charles Darwin. *On the Origin of Species by Means of Natural Selection*. Murray, London, 1859. or the Preservation of Favored Races in the Struggle for Life.

[15] Nikolaus Hansen, Sibylle D Müller, and Petros Koumoutsakos. Reducing the time complexity of the derandomized evolution strategy with covariance matrix adaptation (cma-es). *Evolutionary computation*, 11(1):1–18, 2003.

[16] Oswin Krause, Dídac Rodríguez Arbonès, and Christian Igel. Cma-es with optimal covariance update and storage complexity. In *NIPS*, pages 370–378, 2016.

[17] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dandelion Mané, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. TensorFlow: Large-scale machine learning on heterogeneous systems, 2015. Software available from tensorflow.org.

[18] Francois Chollet et al. Keras, 2015.

[19] D. Foster. *Generative Deep Learning: Teaching Machines to Paint, Write, Compose, and Play*, pages 237–274. O'Reilly Media, 1 edition, 2019.

[20] Richard S. Sutton and Andrew G. Barto. *Reinforcement Learning: An Introduction*. A Bradford Book, Cambridge, MA, USA, 2018.

6 Author contributions

6.1 Implementation

During the implementation phase most of the important/ core components from both environment and World Model were coded in collaboration with all members (Through Deepnote). Some smaller tasks were divided between the group members, Djesse looked into the implementation of the RNN; Marius looked into the alternative optimization strategies; Lieuwe and Djesse looked into the CMA-ES implementation.

6.2 Report

During the literature research phase all members looked into all topics in order for us all to get a complete overview of the important literature. However, the written-out parts in the document were equally divided (e.g., Lieuwe wrote about the Environment, Djesse wrote about CMA-ES, Marius wrote about the World Model). For the implementation details section, the parts that were implemented in collaboration were equally divided between the members, and the parts that were looked into individually were written by the members that implemented it. For the experimental results Marius wrote the analysis of the VAE and GA; Djesse wrote the analysis of the RNN and CMA-ES; Lieuwe wrote the analysis of the observed behavior of the agent. Lastly the conclusion and summary of the results were written in collaboration.

6.3 Collaboration

During the project we often met with all group members through voice calls, where all members actively participated and shared their vision. We all agree that everyone contributed equally and overall enjoyed the project together and are satisfied with the end result.

6.4 Acknowledgement

We would like to thank Elena and the other teachers for introducing us to the topic of natural computing. We would also like to thank the TA's for helpful discussions and guidance. We learned a lot during this course and had fun along the way!

7 Appendix

7.1 Agent-environment loop

Reinforcement learning (RL) is often defined as the science of learning to make decisions [20]. The standard RL problem is as follows: an agent is placed in an environment and receives observations O_t at each timestep. Based on the information the agent it produces an action A_t, or makes a decision, that changes the environment state and the agent state. Over time, the agent learns how to interact with the environment. This setting can be formalised as a Markov Decision Process.

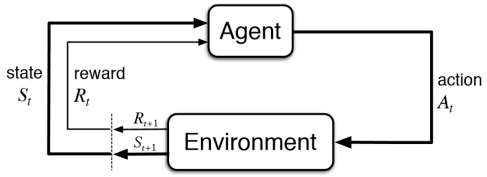

Figure 5: Showing the canonical agent-environment-loop. The agent acts upon the environment and obtains state and reward information from the environment [20].

7.2 Architecture GA

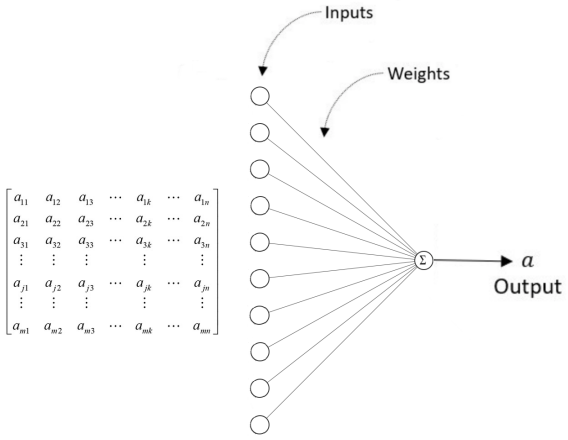

Figure 6: Architecture of the agent trained using evolutionary strategies. The ram state consists of a 1-D vector of length 128 which are weighted using the trained weights. The resulting score is summed and depending on the sum an action is chosen.

7.3 VAE

VAE's are a form of Autoencoders that learn a latent variable model for the input data. This involves learning the parameters of the probability distribution modeling the data. Therefore when points are sampled from this distribution (latent space) the generated outputs are reconstructed inputs.

Given an input vector $\vec{x}$ in $\mathbb{R}^m$ and a latent space vector $\vec{z}$ in $\mathbb{R}^n$, the probabilistic encoder learns an approximate Gaussian posterior distribution $q(\vec{z}|\vec{x})$. Two Vectors of dimension n are created, for mean's ($\vec{\mu}$) and variances ($\vec{\sigma}$) of the distribution. To create $\vec{z}$ a point is randomly sampled from this distribution:

$$\vec{z} = \vec{\mu} + \vec{\varepsilon} \odot \vec{\sigma} \tag{2}$$

Where $\vec{\varepsilon}$ are randomly generated from a normal distribution to maintain stochasticity and for numerical stability log-variances are output instead of the variances directly. It is important to see that these are vector equations in $\mathbb{R}^n$ that

are modelled by a diagonal Gaussian distribution and the operation of $\vec{\varepsilon}$ on $\vec{\sigma}$ is not a dot product. For example when the latent space dimension is $n = 3$:

$$\vec{z} = \begin{pmatrix} z_1 \\ z_2 \\ z_3 \end{pmatrix} = \begin{pmatrix} \mu_1 \\ \mu_2 \\ \mu_3 \end{pmatrix} + \begin{pmatrix} \varepsilon_1 \sigma_1 \\ \varepsilon_2 \sigma_2 \\ \varepsilon_3 \sigma_3 \end{pmatrix} \tag{3}$$

The VAE then reconstructs the image from the stochastic point $\vec{z}$ via a probabilistic decoder $p(\vec{x}|\vec{z})$. An overview of a VAE is shown below in Figure 7.

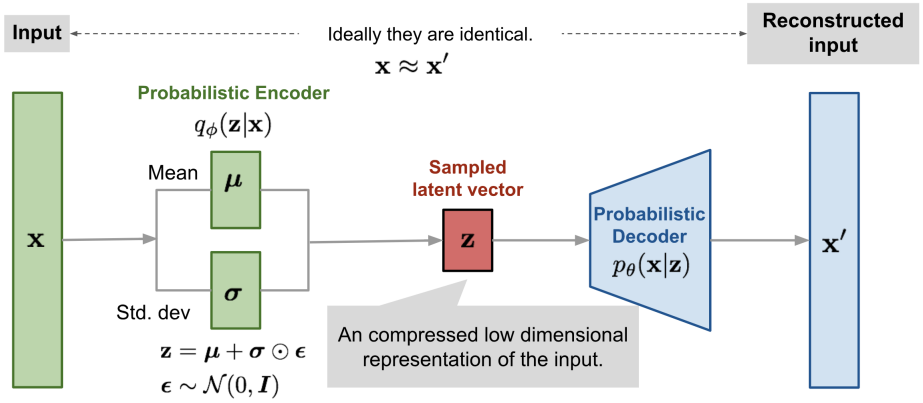

Figure 7: Illustrative example of VAE model.

The model is typically trained using a combination of two equally weighted loss functions. The first is the reconstruction loss function, which measures the difference between the original input and the output. The second is the Kullback-Leibler divergence (KL) or relative entropy between the learned latent distribution and the prior distribution.

Parameter details of our network:

During training both the KL and reconstruction loss reduced. The loss stopped decreasing after around 50 episodes.

Using the network above we obtained the following results:

```
Model: "decoder"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
input_6 (InputLayer)         [(None, 32)]              0
_________________________________________________________________
dense_5 (Dense)              (None, 1024)              33792
_________________________________________________________________
reshape_2 (Reshape)          (None, 2, 2, 256)         0
_________________________________________________________________
conv2d_transpose_9 (Conv2DTr (None, 4, 4, 128)         819328
_________________________________________________________________
conv2d_transpose_10 (Conv2DT (None, 8, 8, 128)         409728
_________________________________________________________________
conv2d_transpose_11 (Conv2DT (None, 16, 16, 64)        204864
_________________________________________________________________
conv2d_transpose_12 (Conv2DT (None, 32, 32, 32)        73760
_________________________________________________________________
conv2d_transpose_13 (Conv2DT (None, 64, 64, 3)         3459
=================================================================
Total params: 1,544,931
Trainable params: 1,544,931
Non-trainable params: 0
```

Figure 8: Decoder architecture

```
Model: "encoder"
__________________________________________________________________________________________________
Layer (type)                    Output Shape         Param #     Connected to
==================================================================================================
input_5 (InputLayer)            [(None, 64, 64, 3)]  0
__________________________________________________________________________________________________
conv2d_8 (Conv2D)               (None, 32, 32, 32)   1568        input_5[0][0]
__________________________________________________________________________________________________
conv2d_9 (Conv2D)               (None, 16, 16, 64)   32832       conv2d_8[0][0]
__________________________________________________________________________________________________
conv2d_10 (Conv2D)              (None, 8, 8, 128)    131200      conv2d_9[0][0]
__________________________________________________________________________________________________
conv2d_11 (Conv2D)              (None, 4, 4, 256)    524544      conv2d_10[0][0]
__________________________________________________________________________________________________
flatten_2 (Flatten)             (None, 4096)         0           conv2d_11[0][0]
__________________________________________________________________________________________________
dense_4 (Dense)                 (None, 32)           131104      flatten_2[0][0]
__________________________________________________________________________________________________
z_mean (Dense)                  (None, 32)           1056        dense_4[0][0]
__________________________________________________________________________________________________
z_log_var (Dense)               (None, 32)           1056        dense_4[0][0]
__________________________________________________________________________________________________
sampling_2 (Sampling)           (None, 32)           0           z_mean[0][0]
                                                                 z_log_var[0][0]
==================================================================================================
Total params: 823,360
Trainable params: 823,360
Non-trainable params: 0
```

Figure 9: Encoder architecture

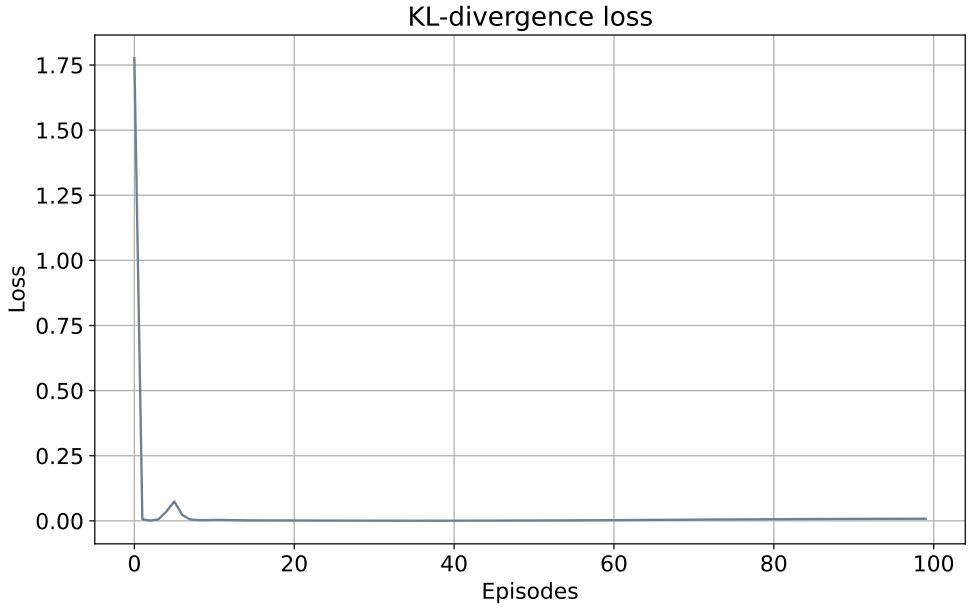

Figure 10: The KL-divergence loss of the VAE over 100 episodes.

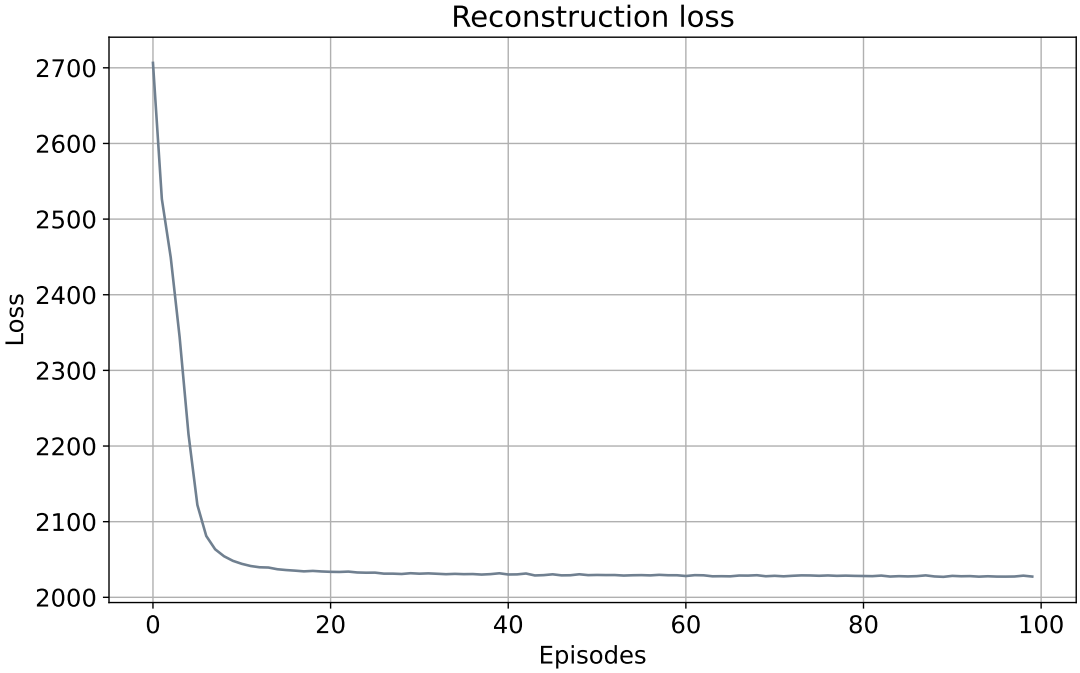

Figure 11: The reconstruction loss of the VAE over 100 episodes.

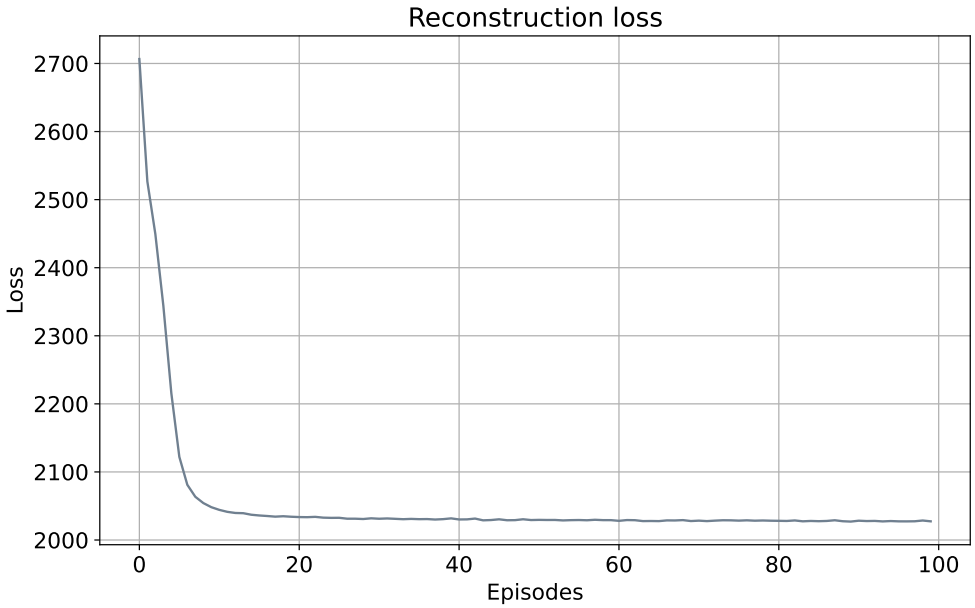

Figure 12: The combined loss (KL-loss and reconstruction loss of the VAE over 100 episodes. The total loss is by a large margin dominated by the reconstruction loss.

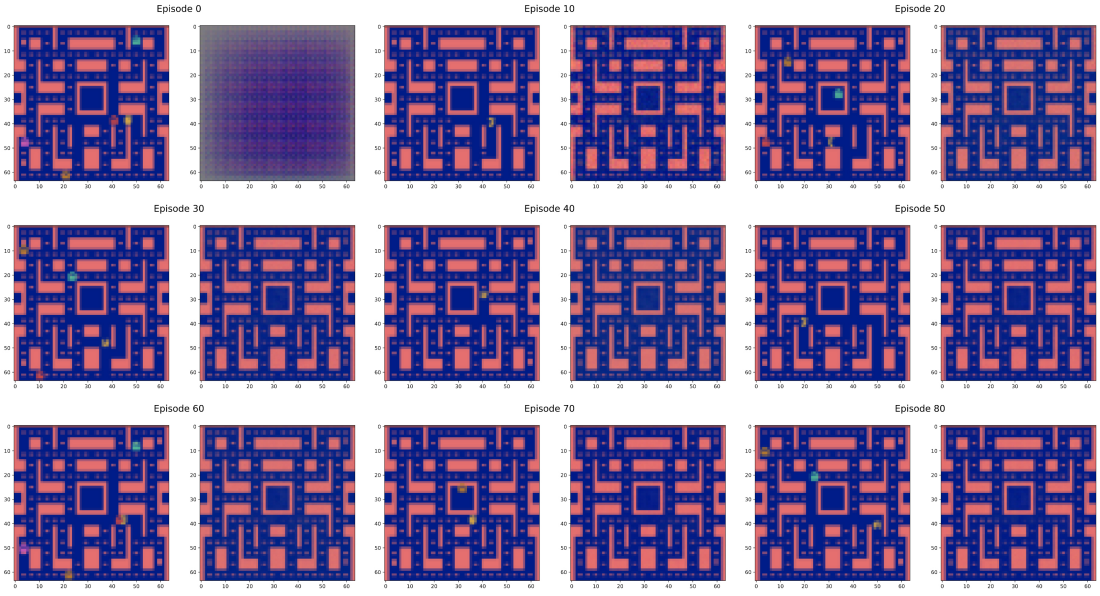

Figure 13: Reconstruction for the first 90 epochs of the training. The left image always shows the actual environment state while the right image shows the reconstruction. The environment is perfectly recreated but the ghosts are missing.

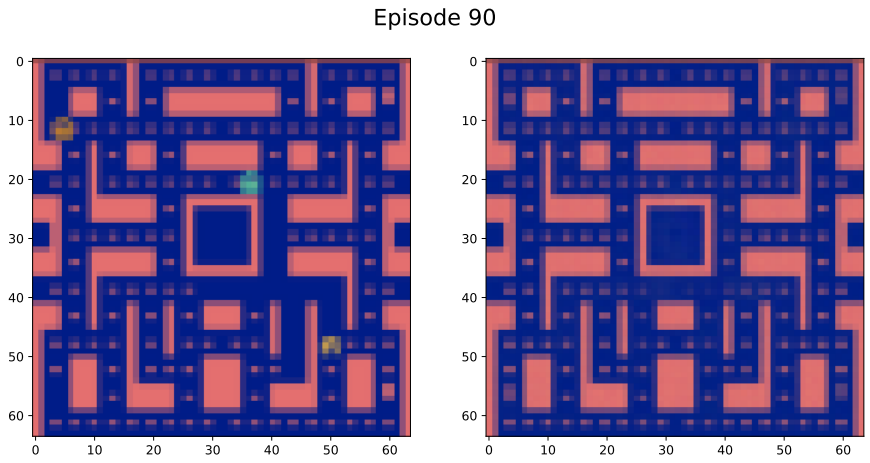

Figure 14: Showing the obtained reconstruction after 100 episodes. The main element of the game (the individual agents) for making the latent space compression of the VAE useful is still missing.

7.4 RNN results

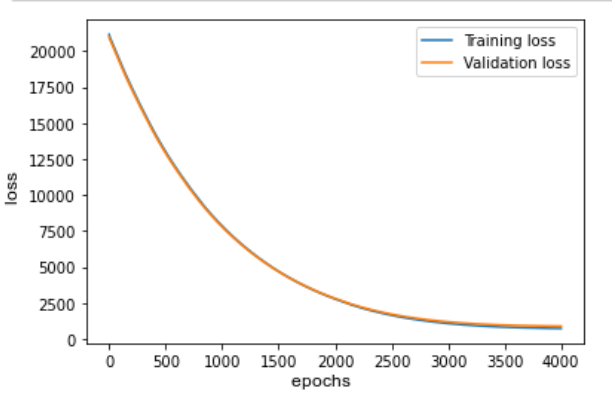

Figure 15: Training loss RNN trained on an action space of 4

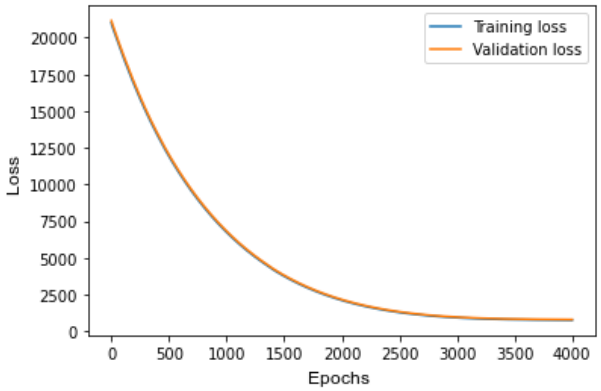

Figure 16: Training loss RNN trained on an action space of 9

7.5 Genetic Algorithm (GA) Pseudocode

Algorithm 1 Evolution Strategies

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

Figure 17: Pseudo code for Evolutionary Strategies [8]

7.6 Random agent (Baseline) Results

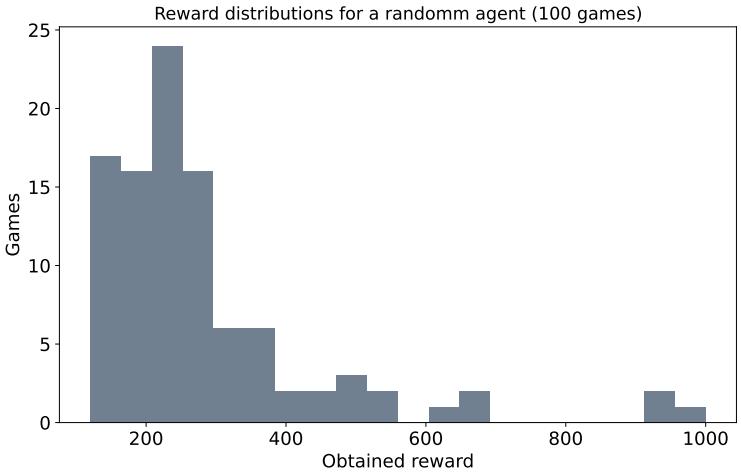

Figure 18: Reward distribution obtained for the random agent over 1000 games.

7.7 Sequence Mutation (SQM) Results

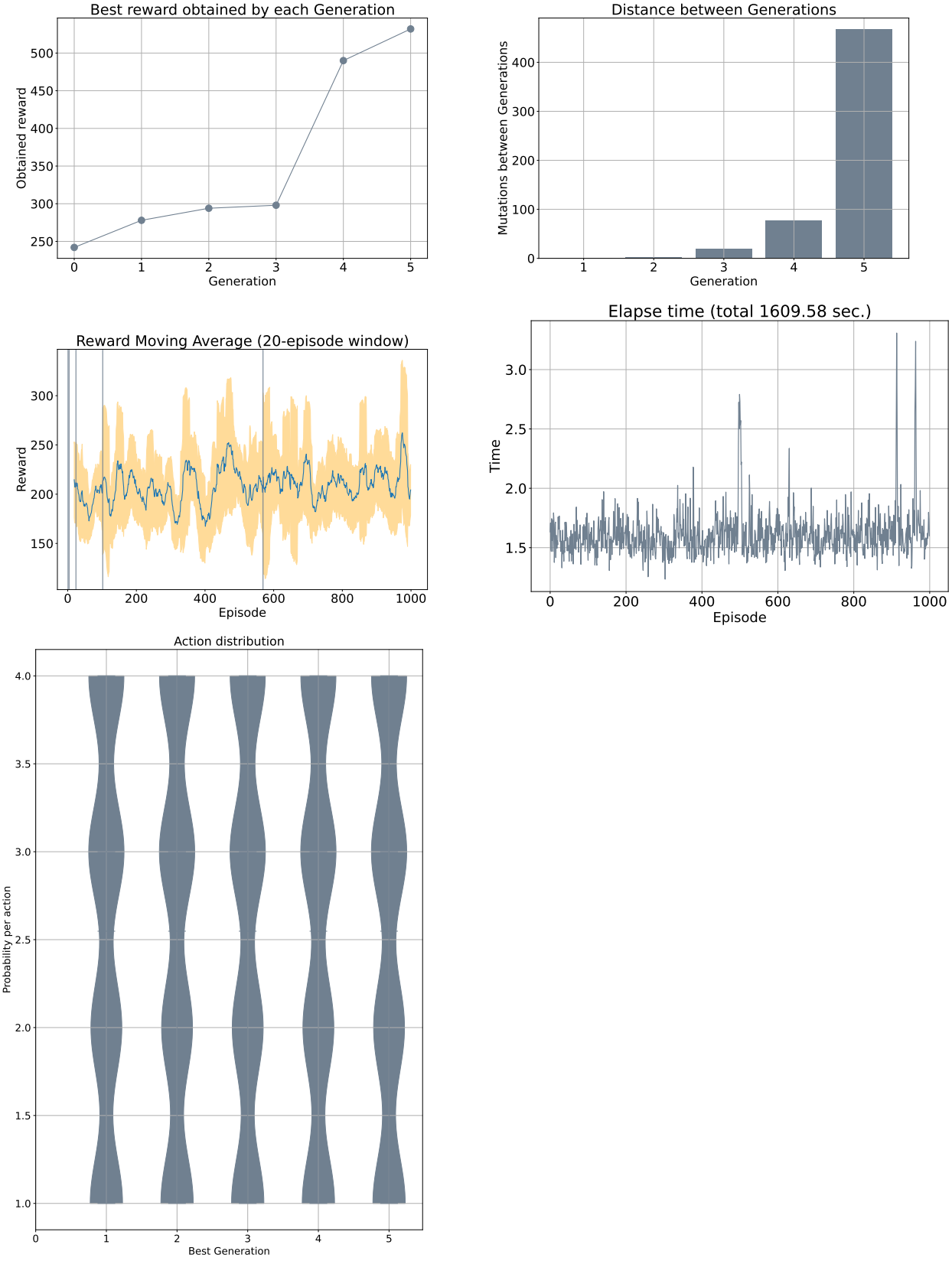

Figure 19: MsPacMan: Mutation probability of 0.1 with an action space of 4.

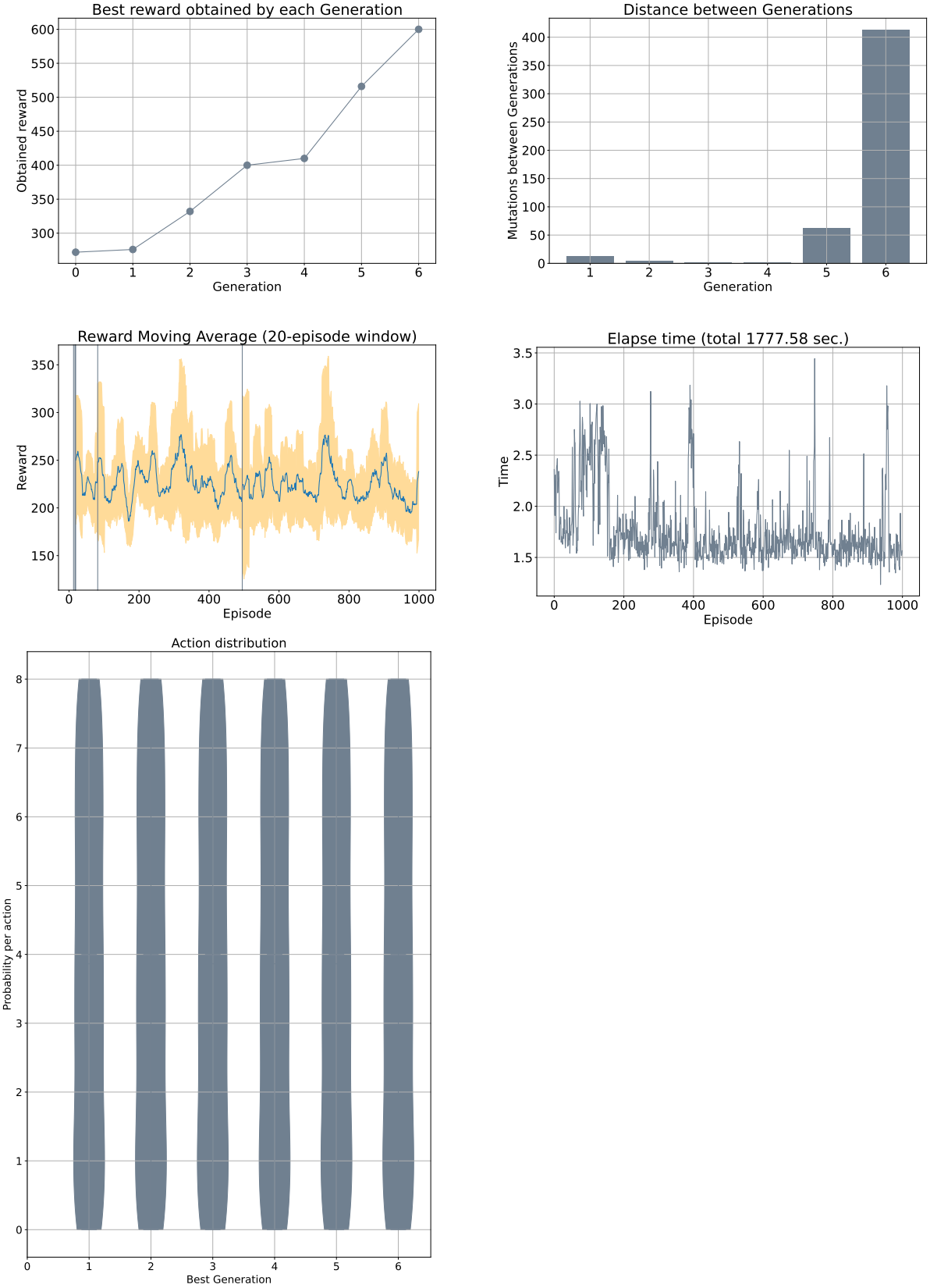

Figure 20: MsPacMan: Mutation probability of 0.1 with an action space of 9.

7.8 Genetic Algorithm (GA) Results

7.8.1 Results GA - state only

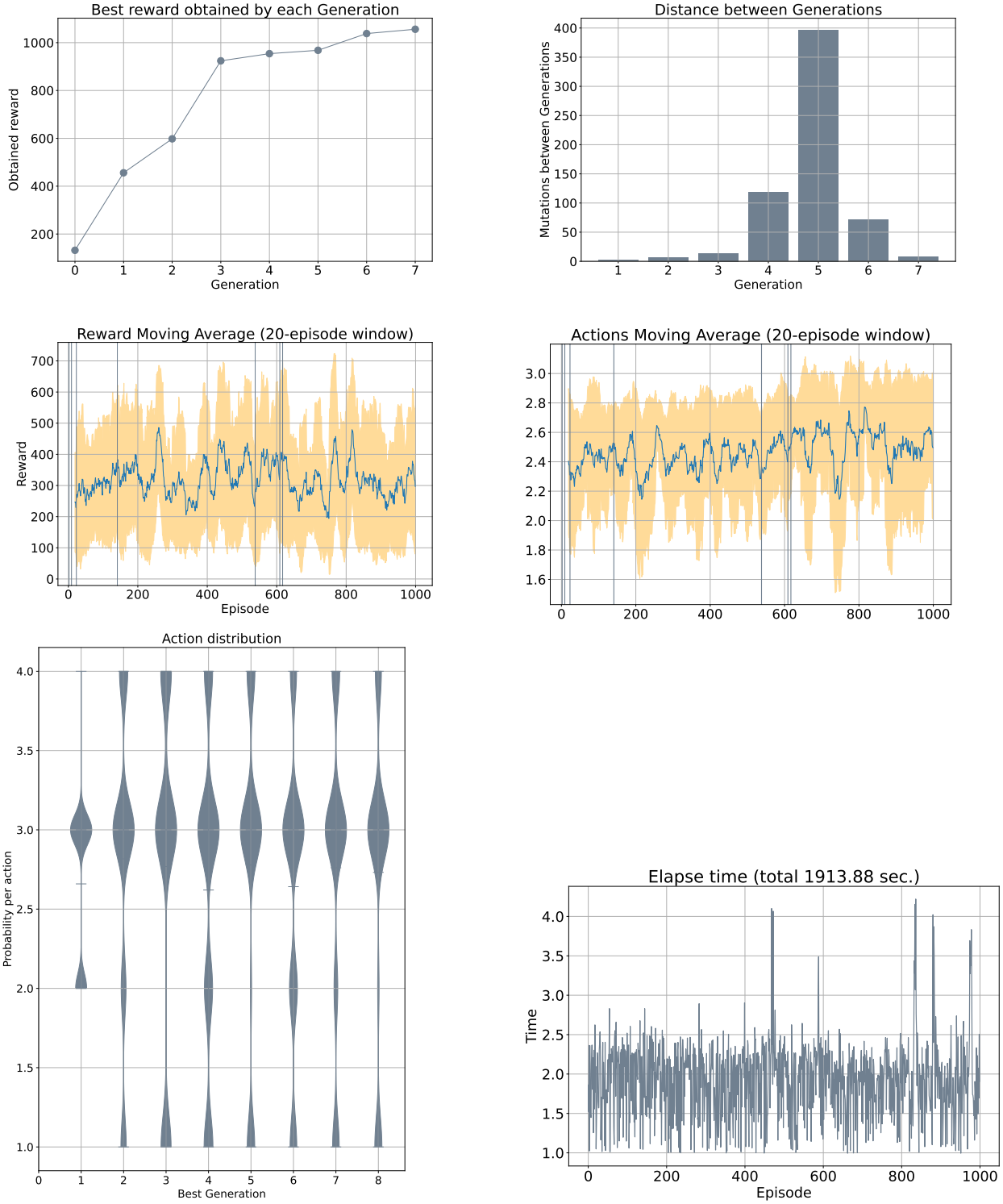

Figure 21: MsPacMan: Noise term of 0.4 with an action space of 4.

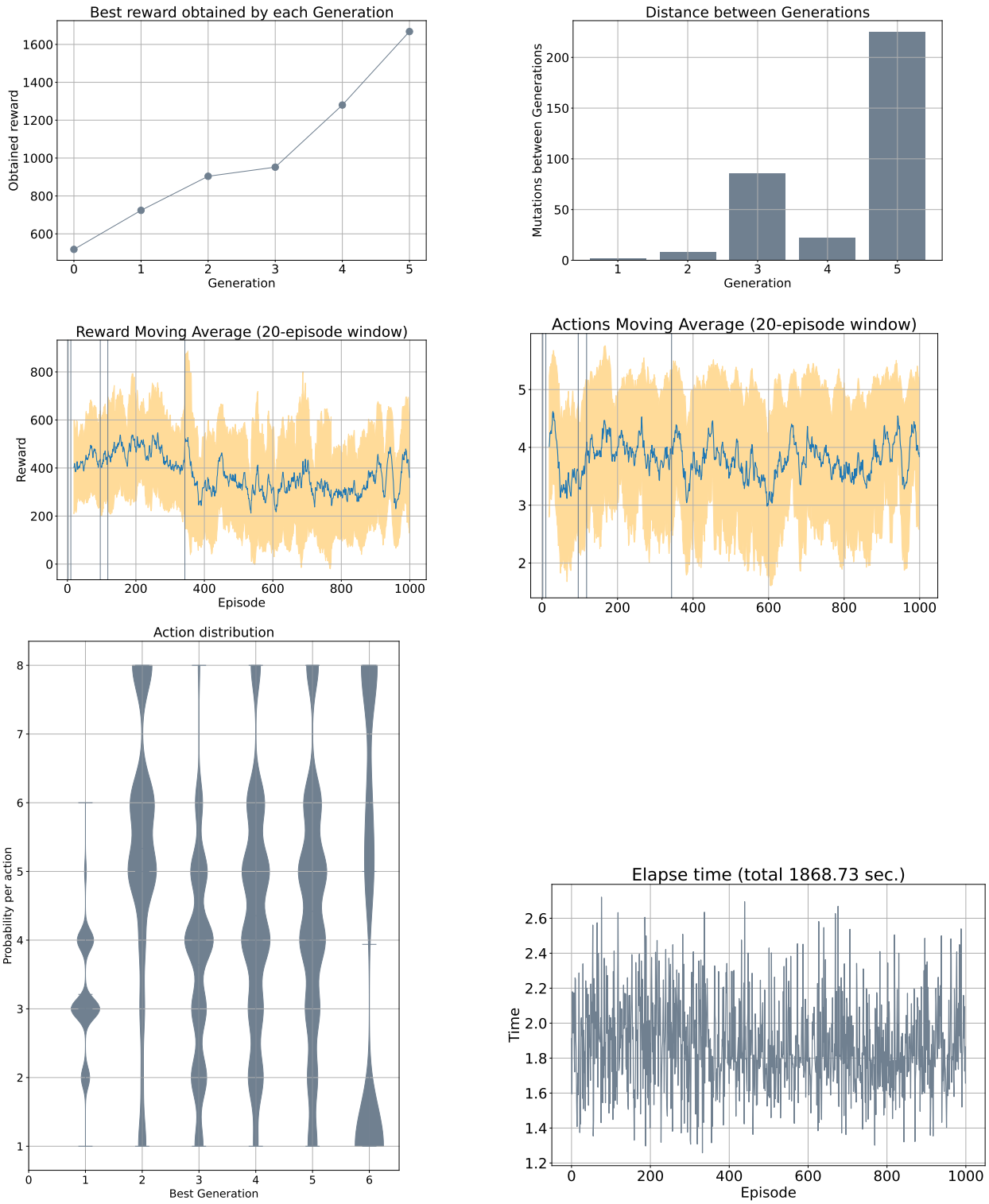

Figure 22: MsPacMan: Noise term of 0.8 + sparse with an action space of 9.

7.8.2 Results GA - state + RNN

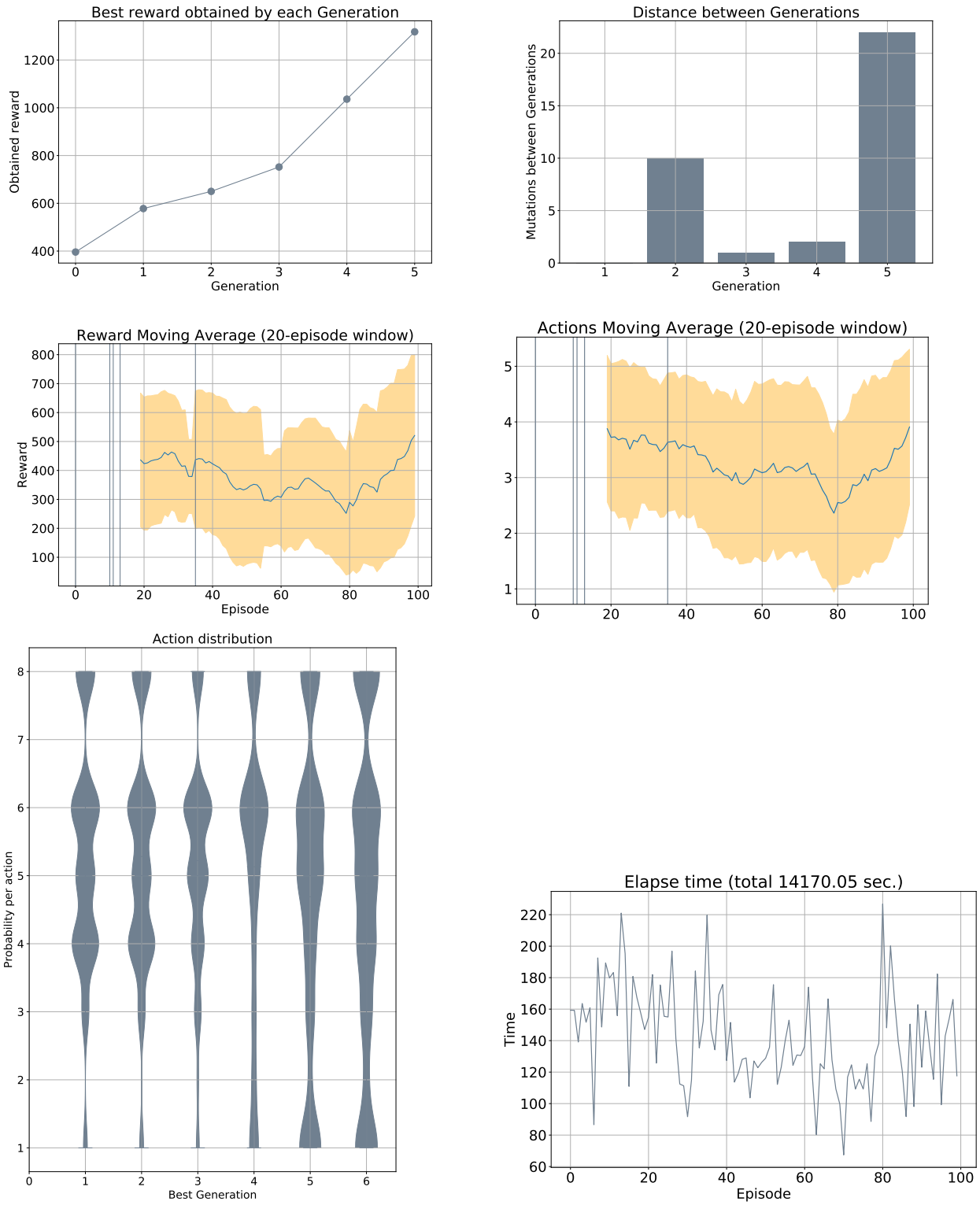

Figure 23: MsPacMan: Noise term of 0.8 + sparse and RNN with an action space of 9.

7.9 Covariance Matrix Adaptation Evolution Strategy (CMA-ES) Results

7.9.1 Results CMA-ES - state only

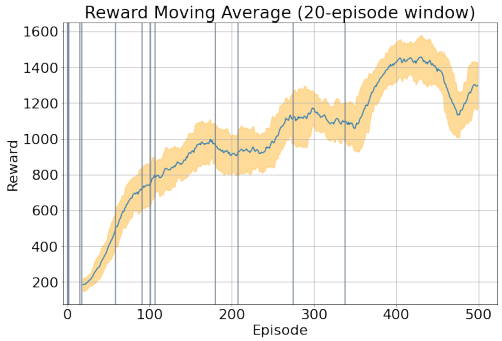

Figure 24: Average moving reward during training using an action space of 4

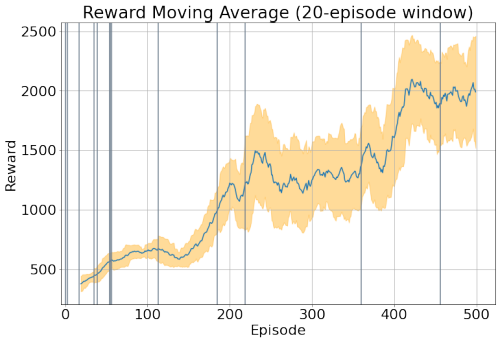

Figure 25: Average moving reward during training using an action space of 9

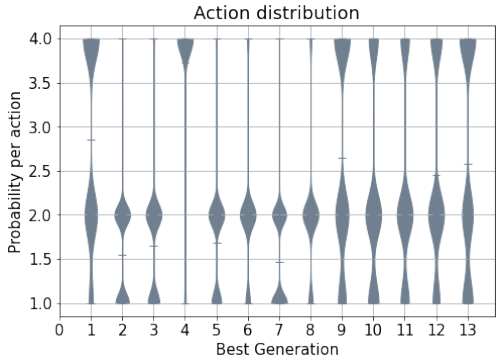

Figure 26: Distribution of actions for best performing configurations during training using an action space of 4

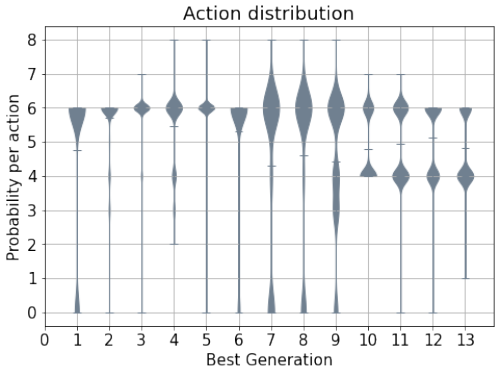

Figure 27: Distribution of actions for best performing configurations during training using an action space of 9

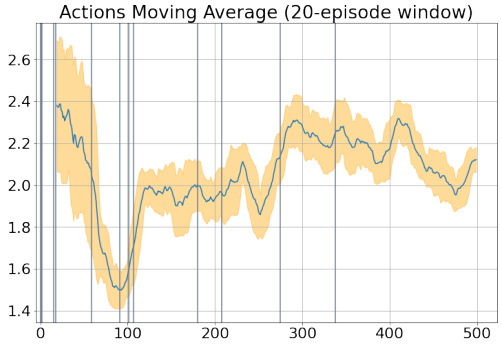

Figure 28: Average action per generation during training using an action space of 4

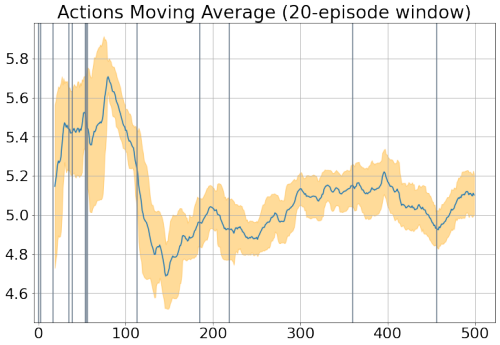

Figure 29: Average action per generation during training using an action space of 9

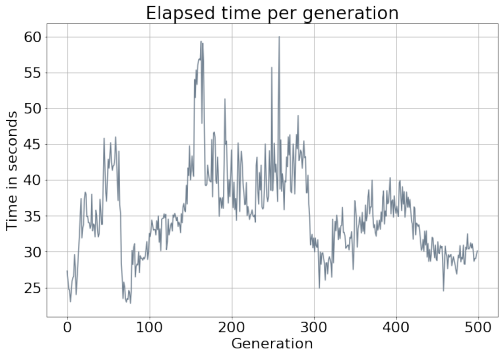

Figure 30: Elapsed time per generation during training using an action space of 4

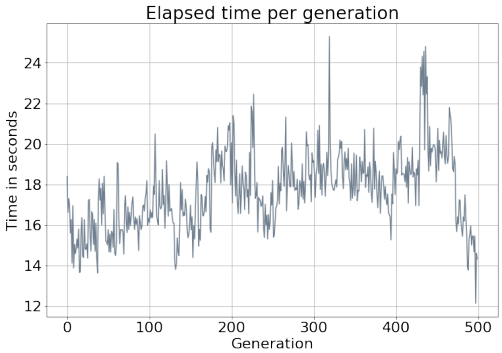

Figure 31: Elapsed time per generation during training using an action space of 9

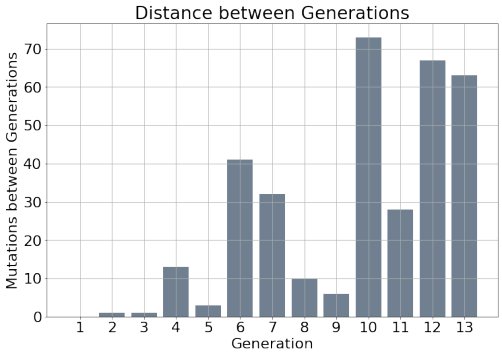

Figure 32: Number of generations between new high scores using an action space of 4

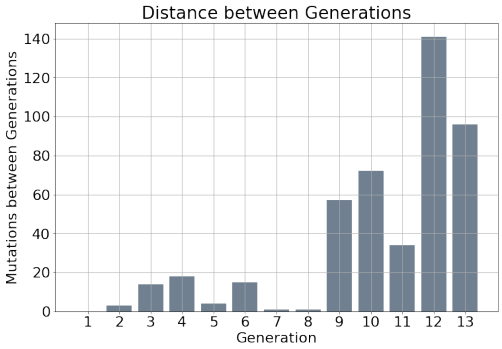

Figure 33: Number of generations between new high scores using an action space of 9

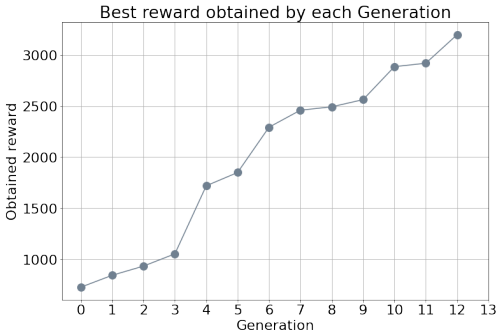

Figure 34: Obtained reward at high score generations using an action space of 4

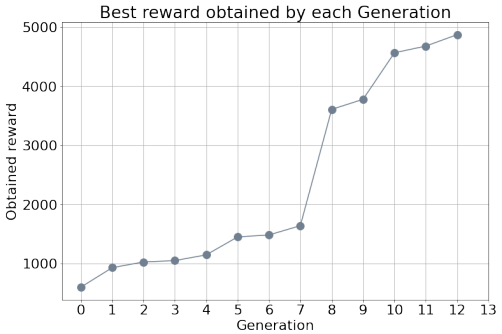

Figure 35: Obtained reward at high score generations using an action space of 9

7.9.2 Results CMA-ES - state + RNN

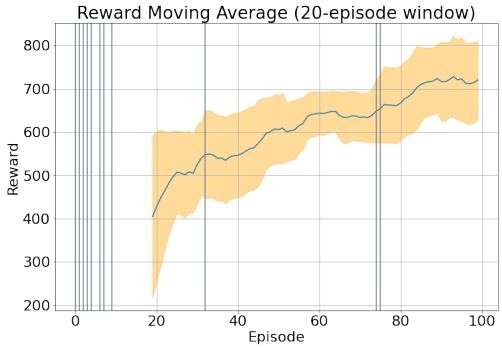

Figure 36: Average moving reward during training both game state and RNN hidden state included

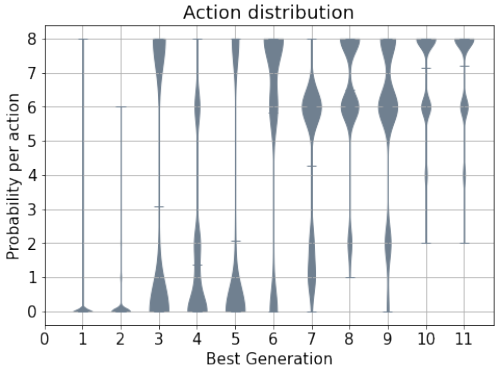

Figure 37: Distribution of actions for during training both game state and RNN hidden state included

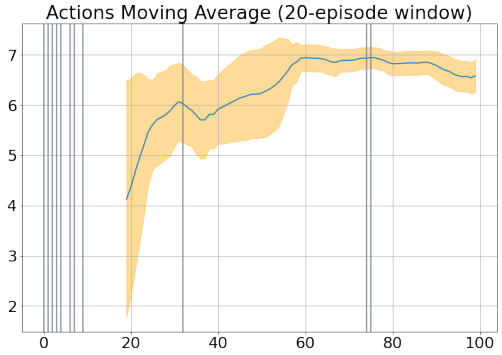

Figure 38: Average moving actions during training both game state and RNN hidden state included

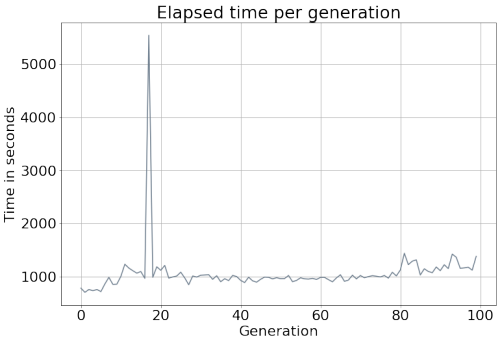

Figure 39: Elapsed time per generation during training both game state and RNN hidden state included (the spike can be explained by a system malfunction which led to an unintended pause of the run)

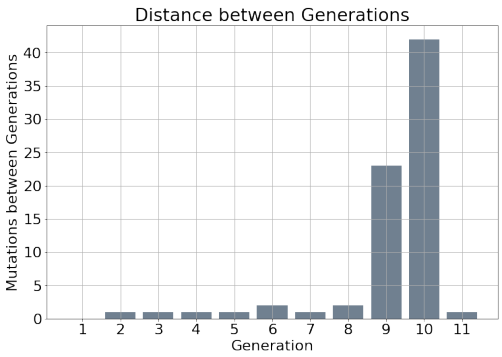

Figure 40: Number of generations between new high scores with both game state and RNN hidden state included

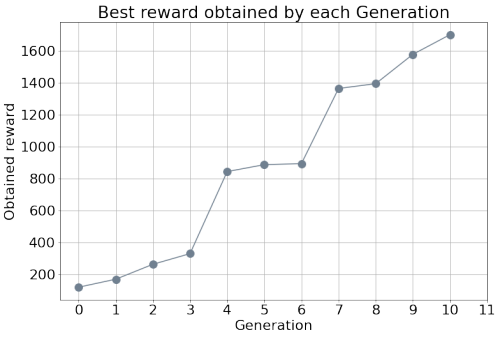

Figure 41: Obtained reward at high score generations with both game state and RNN hidden state included