Natural Computing Project: World Models

Mick van Hulst (s1013954), Jorrit van der Laan (s4254120)
& Jelle Piepenbrock(s4244729)

June 2018

1 Goals

In this report, we explore the possibilities opened by the 'World Models' paper, published as a preprint by David Ha and Juergen Schmidhuber in 2018 [1]. We attempt to expand on the experiments done in the paper, by playing a more complex game than attempted within the original work. Concretely, we use the World Model approach to play the game Space Invaders, which was the only game that was not conclusively beaten using Deep Q-Learning in the 2013 DeepMind paper[2]. We also let trained controllers generate more training data for later controllers, so that later controllers see more complex stages of the game, because earlier controllers access more later game-states. We hope to explore the possibilities of the combination of unsupervised environment compression and controller agents optimized by genetic algorithms, specifically Covariance Matrix Adaptation Evolution Strategies (CMA-ES) (which is possible because the world models contain most of the parameters, opening the door to optimization techniques that only work on relatively simple controllers).

2 Review of Related Work

2.1 General Reinforcement Learning

Reinforcement learning is a machine learning technique lightly based on the concepts observed in human learning. An agent is to reach a complicated goal requiring many steps (e.g. reaching a high score in a game). The main technique utilized is rewarding good decisions and punishing bad decision. This is known as credit assignment. This is not supervised learning in the traditional sense, because it is not known whether an individual step is correct or not. Rather, each sequence of steps is given a reward, and the algorithm must decide what steps to take in order to achieve a higher reward.[3] The existence of ill-conditioned, non-convex reward functions where credit assignment is difficult creates a need for highly flexible optimization procedures, such as those from the genetic algorithms field.

2.2 Deep Q learning / OpenAI

In 2013, the DeepMind laboratory, then still a separate entity from Google, released a paper named 'Playing Atari with Deep Reinforcement Learning'[2]. In this work, the group used a deep learning model to learn control policies for Atari 2600 games in an end-to-end manner. The model was a convolutional neural network that had, as its input, the visual representation of the game state. The model was tasked with predicting the future reward of the available actions. The major innovation here was that the model only used the raw visual state of the game and no preprocessed game state information. In essence, this approximates the way a human player would learn how to play a game. DeepMind achieved state-of-the-art results on 6 out of the 7 Atari games they experimented on, except for Space Invaders, where higher results were achieved using hand-crafted features. In 2015, the authors released an updated version of the paper with improved performance. However, we will only consider the first version as a benchmark for our experiments later, as this project is trying to prove a concept that may allow alternative optimization methods in the future. Immediately being competitive with state-of-the-art solutions is not a priority.[4]

In 2016, the Atari game platform's use as a training ground for reinforcement learning algorithms was formalized with the release of OpenAI's Gym research environments. Within these standardized environments, researchers can develop algorithms that can solve various kinds of control problems, as well as a large array of games originally developed for the Atari 2600 game console [5]. In 2018, the Sega Genesis console, which has more complex games available, was also added to the possibilities. Note that the results obtained by DeepMind are not state-of-the-art anymore, but the computing time required for the succeeding papers is well out of the scope of this project. Therefore the computationally less heavy World Model approach was chosen.

2.3 World Models

In a recent paper by Ha and Schmidhuber[1], a world model system was used to effectively solve a track-racing game, for which both control and planning were needed. The model has an internal model of the dynamics of the world, both in spatial (via a Variational Autoencoder [6]) and time (via a Recursive Neural Network, specifically a Long Short Term Memory network, LSTM [7]) dimensions. See 1 for an overview of the architecture. These compressed versions of the agent's reality are used to guide a simple controller agent, that can be trained using a Genetic Algorithm. Because most of the parameters are in the world model part of the agent, which does not have access to the reward function, the controller part has relatively few parameters. This means that large populations and generation numbers are possible to train agents, even for environments in which credit assignment is difficult (because there is no effective gradient that guides decisions toward a better outcome). One of the main contributions of the World Models paper is thus the opening of new possibilities in

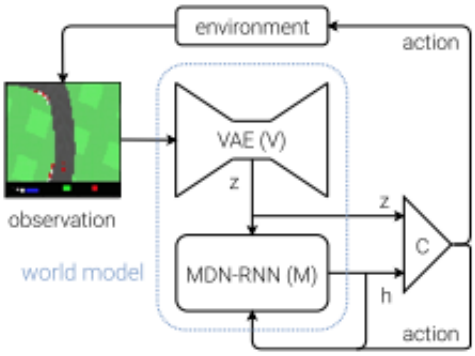


Figure 1: The world model architecture, source [1]

using evolutionary strategies to obtain effective agents to solve problems. The different components will be discussed in more detail.

3 Description of the Algorithm

3.0.1 VAE

An autoencoder is a neural network consisting of two parts. The first part compresses the input into a small vector, the latent vector, and is called the encoder. The second part reconstructs the original input from this vector and is called the decoder. The network is trained by feeding in training data and comparing the input with the output. Ideally, these are very similar. However, because the network squeezes the input through the latent vector, the output cannot be reconstructed perfectly. So, in order to approximate the input as accurately as possible, the network learns how to store the input data in the latent vector. When training is finished, the encoder is used to efficiently compress the data into a number of learned variables.
A traditional autoencoder is good at encoding familiar input. However, the decoder is not good at generating new data, because the latent space will contain discontinuities. When generating data using a latent vector from a discontinuity, this will result in an unrealistic output. In other words, there is no guarantee for the validity of a generated output, in the sense that it corresponds to a possible game state.

To alleviate this problem, a variational autoencoder learns a mean and a standard deviation for every latent dimension, instead of directly learning the latent vector. A sample is then taken from the ensuing Gaussian distribution, which gets decoded like in a normal autoencoder. The advantage gained is that the latent space is constrained: two similar inputs should have a similar encoding

and the Gaussian nature of the distribution ensures that decoded outputs stay within a certain range. Now, when using the decoder to generate data, more realistic pictures will be generated. However, the latent space might still not be used smoothly. That is, during training, the standard deviation will tend to get very small, to make the samples similar to the mean. This will result in small clusters in the latent space, which do not overlap. To address this, the Kullback-Leibler-loss was introduced, which punishes this behavior. This ensures that the latent variables stay Gaussian distributions with reasonable standard deviations.[8][9]

3.0.2 MDN-RNN

In a standard neural network, it is assumed that the input and output of the network are independent. However, for a lot of problems, like predicting the next word in a sentence or the next frame in a game, this assumption makes little sense. A recurrent neural network addresses this problem by also taking into account previous inputs when making a new prediction. More precisely, when processing an input, vectors called hidden states are modified. The prediction of the next input is then determined not only by the weights of the neural network, but also by these hidden states. Moreover, the RNN then modifies the hidden states again, 'remembering' this event.
Unfortunately, this still doesn't completely solves the problem. Often, it is deterministically unknown what a right prediction is. Rather, there is a range of possible solutions, each with a certain chance of being the right one. For example, when finishing the sentence 'The cat lies on ...', multiple endings are possible, like 'the couch' or 'the roof'. Although neither are wrong, 'the couch' may seem like a better answer. This probabilistic behavior can be simulated using a mixture density layer as the last layer of the network. Instead of one prediction, its output consists of several Gaussian distributions, where the amount of distributions depend on the number of nodes of the penultimate layer. Each of these distributions corresponds to a group of similar solutions. Because some groups are more plausible than others, these distributions have weights themselves as well. These weights add up to 1, again creating a probability distribution, the so called mixture density distribution.
The network combines these two components and is called a RNN-MDN, and is used in the world models paper.[10][11]

3.0.3 Controller

The controller is a simple linear model that maps the two latent vectors from the VAE and MDN-RNN to an action with the highest reward. The reason a linear model is chosen, as opposed to a more complicated neural network, is that this allows for evolutionary strategies. This is important, because estimating the gradient of a reward is difficult when the rewards are sparse. This can result in getting stuck in a local optimum, or even completely meaningless results. The evolutionary strategy used in this project is CMA-ES, or covariance matrix

adaptation evolutionary strategies [12].

3.1 CMA-ES

3.1.1 Evolutionary Strategies

CMA-ES is an evolutionary strategy, which mean that it approaches the process of optimization from an evolutionary perspective. The basic notion of an evolutionary strategy is to provide a set of candidate solutions for a certain problem. These candidate solutions are then scored against the actual environment or function. Afterwards, a new generation of solutions is generated through recombining the best candidate solutions. This process repeats until a satisfactory performance has been reached.
The quality of a candidate solution is determined with a fitness function. How this function looks can differ. For example, it could be the high score in a game. In particular, it does not have to be differentiable or continuous. This as opposed to a loss function in a neural network, which at least should be approachable by a differentiable function like a Taylor series, for the backpropagation algorithm to work.
Candicate solutions are generated by picking a solution which serves as our initial mean μ and standard deviation σ. Using this normal distributions (with mean μ and standard deviation σ), a set of candidate solutions can be generated. New generations use the average of the best solution from the previous generation as its mean.
Although this works well for a wide range of problems, a major drawback is that σ is fixed. This has two disadvantages. First, when the algorithm is not very confident about a solution, σ should be increased, to broaden the search space. This decreases the chance of getting stuck in a local optimum and speeds up converging. Second, when the algorithm is confident about a solution, σ should decrease, so we can really fine-tune our solution.To address these problems, CMA-ES was introduced.[13].

3.1.2 Covariance Matrix

CMA-ES stands for Covariance Matrix Adaptation Evolutionary Strategy. As the name suggests, this algorithm makes use of the covariance matrix. If a normal distribution has only one variable, the distribution is completely described by its mean and standard deviation. However, most problems have a multivariable solution, and for those a multivariable distribution should be used when sampling solutions. When a normal distribution has multiple variables, all these variables can have different standard deviations, which changes the distribution. When, for example, the standard variation is larger in one direction, the distribution will be stretched in that direction as well. Moreover, two variables can be correlated, which will result in a rotation. All this information can be found in the covariance matrix. Therefore, a covariance matrix shows how much a standard Gaussian distribution has been distorted in order to get the

current distribution. This also works the other way around. Given some data, computing the covariance matrix for a mean, results in a Gaussian distribution.

3.1.3 CMA-ES

To initialize CMA-ES, a set of candidate solutions is taken from a Gaussian distribution, and the best solutions are selected. The μ and σ of this distribution can be chosen. This initial σ is a parameter called the step-size. A distribution is genereated, such that the new generation of candidate solutions may be sampled. For CMA-ES, the value of μ is calculated by using the mean of all solutions, instead of the mean of the best solutions. This means that, when all the best solutions are far away in a certain direction, the covariance matrix of the best solutions will tell us that there are huge distortions (as in this case the standard deviation in that direction is large). If, on the other hand, all the best solutions lie very close to the mean, the distortions are small, and so the new standard deviations will also be small. After calculating the new covariance matrix, the new mean is calculated and a new generation of candidate solutions is sampled (see figure 2 one for an example of this process).[13]

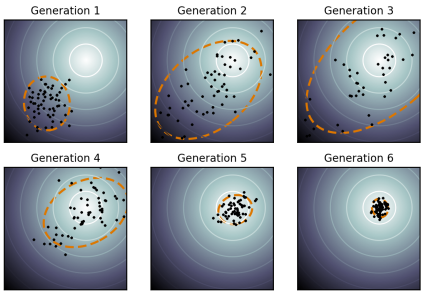


Figure 2: Several iterations of CMA-ES. The white spot is the optimum and the dashed line the distribution, source [14]

In figure 1 at generation 1 the best solutions all lie in the upper right corner. This causes a big distortion, hence the bigger ecliptic dashed line at the second generation. In the third generation the best solutions lie close to the mean, so the distortions gets less. In generation 6 the algorithm has approximated the solution already very well.

4 Description of testing procedure

In order to train the different components of the model, random data is generated by letting the agent move randomly in the Space Invaders environment. This is done for 160 episodes, each time saving the first 300 frames of the corresponding episode. Using this data, a VAE is trained with a latent vector of size 32.

After training the VAE, the RNN is trained. Using the VAE, the observations are encoded such that the RNN can train on them. The RNN is trained by predicting the next set of encoded observations, given the previous set of encoded observations and corresponding actions.

Lastly, the controller is trained by letting the population of agents play the Space Invaders game within the Atari Gym environment. Every generation, 12 individuals are tested, were each plays ten games of Space Invaders. Their performance is averaged and used to optimize in the CMA-ES procedure. After every generation, the distribution's mean parameters (i.e. the distribution that new individuals were sampled from) is tested for 100 episodes as if it is a single individual. As a reference point, a random agent gets a score of approximately 160 over 100 episodes.

For the VAE and RNN to work properly, the data which it is trained on should be a good representation of all the possible frames of the game. To determine the impact of high/low variance in data on the VAE, new data is generated after the controller has been trained. It is expected that the agent becomes better overtime, meaning that the agent survives for more timesteps[1] and can thus create a higher variance in the observations that are generated. By using the controller, new data is generated by first running the games 8 times. These runs last for a certain number of frames (i.e. until the agent dies), and the 60th percentile of these is calculated, lets call it x. By using x, the agent plays the game for 160 episodes and the first $x+n$ frames are stored, where n is a random number between 0 and 75. This n is added to further improve the variance. When an agent dies before reaching this number of frames, it is simply ignored.

Using this new data, the VAE, RNN and controller can be retrained, which in turn can result in higher scores. We will call this process of generating data and training the VAE, RNN and controller an iteration. This iterative process can be repeated and was also mentioned in the 'Future Work' section of the original World Models paper.

The following experiments were conducted. Note that all models within the same iteration are trained on the same data.

1. First iteration: Agent trained on random data
2. Second iteration: Agent trained with 0.5 initial sigma (step-size) [3 replicates]
3. Second iteration: Agent trained with 0.1 initial sigma (step-size) [3 replicates]

[1]The number of observations generated by a game up until a certain point. This also corresponds with the number of actions taken by the agent as the agent selects an action at every timestep.

4. Second iteration: Agent trained with 0.1 initial sigma (step-size) and information about timestep [3 replicates]

5. Third iteration: Agent trained with 0.1 initial sigma (step-size) and information about timestep [3 replicates]

In the table below, the original table from the 2013 paper on Deep Q Network learning is reproduced, in order to give reference performances. Note that in the years between 2013 and now, there have been several great advances in deep reinforcement learning, such as Asynchronous Reinforcement Learning (A3C), which can get a super-human score of over 22000. However, we chose to compare with methods that need somewhat comparable computational power. For example, the most modern A3C trains in 4 days on 16 CPUs, while we trained our agents for 3-4 hours on 16 CPUs. In addition to this, the original intent of the World Models paper is to train a relatively simple linear agent, so that this agent can be trained through genetic algorithms, and having most of the information encoded in the world models. Therefore we chose to compare mostly with the algorithms below.

Table 1: Performance of other algorithms, as of the original DeepQ learning paper. Note that better algorithms than DQN, such as A3C, have been published. These approaches however use more computational power.

	Random	Sarsa	Contingency	DQN	Human
Mean score	179	271	268	581	3690

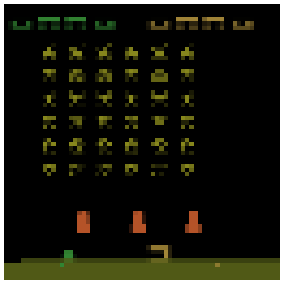
5 Results & Discussion

First, the quality of the results of the world model encoding will be shown and discussed, as these are essential for the optimal training of the controller agent. Afterwards, the performance of the controller unit while playing Space Invaders will be shown.

5.1 Variational Autoencoder

The input of the VAE consists of observations, which were resized to 64x64 to match the original paper's dimensions and to limit the amount of parameters. Figure 3 shows one of the same images, before and after resizing, which were used to conclude that resizing did not lead to a loss of information (i.e. testing images consisted of testing to see if all important features like enemies and projectiles were still visible).

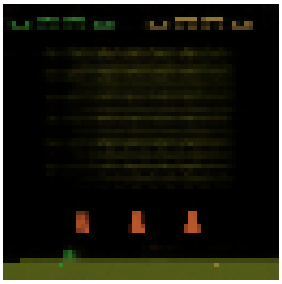
(a) Environment before resizing



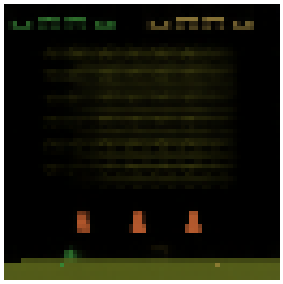
(b) Environment after resizing

Figure 3: Observations before and after resizing to 64x64

The integrity of the output data of the VAE was verified by decoding observations which the VAE previously encoded. Figure 4 shows two frames consisting of decoded images which were originally randomly generated images. One can observe that the encoded data, which is generated by the VAE is low in variety as the timesteps are significantly different and the VAE signals no change in terms of the environment.



(a) Timestep 0

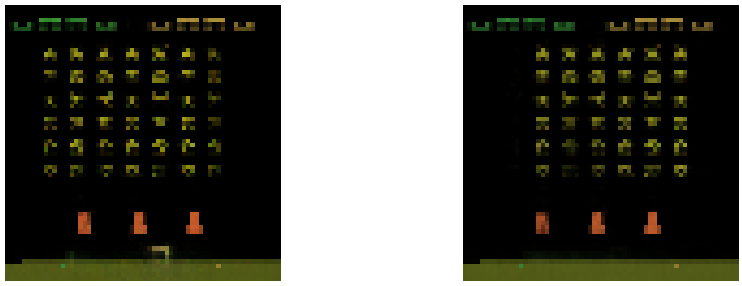


(b) Timestep 250

Figure 4: VAE with lower variety data

The observation lead to ensuring variety and punishing the loss function by randomly choosing a starting position for saving the observations and by increasing the punishment for the loss function by a factor of 255. This means that many of the observations don't have data for the first few frames of the agent. This change resulted in a higher loss as the variety of the data increased. Figure 5 shows two frames for the same timesteps as the previously mentioned figures, but then with the changes which are described above. The figures show a clearer environment (i.e. the enemies are now clearly visible), as well as a change in the position of the enemies. After the change, the VAE still doesn't encode the lasers and/or encode the change in position of the agent. As these are important

features, several settings for the VAE were tested (amount of latent variables of 16, 32, 64 and 128). These tests didn't show any desirable results when it came to encoding more information (i.e. lasers, updated position of the player or less blurry environment). In the case of 16 latent variables, the VAE experienced the exploding gradient problem. This was countered by using ReLU and batch normalizing [**15**]. This, however, didn't result in the VAE being able to encode the images effectively. Figure 6 shows an example of a decoded image which the VAE generated with the amount of latent variables being 16.

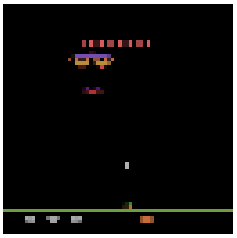


(a) Timestep 0 (b) Timestep 250

Figure 5: VAE with higher variety data

Figure 6: VAE decoded image with 16 latent variables

As the VAE is not able to capture all the significant features, the model might not be suited for Space Invaders. To verify whether or not this solely holds for Space Invaders, extra experiments were performed for a game called Raiden. Raiden is a game which is similar to Space Invaders, but has fewer enemies and bigger lasers. Figure 7 shows a before and after decoded image for the game Raiden. The VAE was not able to encode all the important features of the game. This leads to the hypothesis that this could be due to some of the features (lasers) being temporary data, meaning that not every data frame has a laser present, however, every data frame has an enemy present. This could mean that the VAE prioritizes data which is always present over data which isn't always present to minimize the loss. This leads to the conclusion that even after choosing a game (Raiden) where the enemies' projectiles are bigger, this still wasn't sufficient for the VAE to encode them.



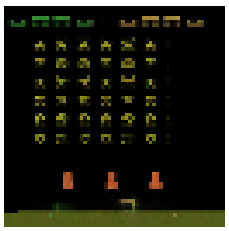
(a) Timestep 10 - original

(b) Timestep 10 - encoded and then decoded.

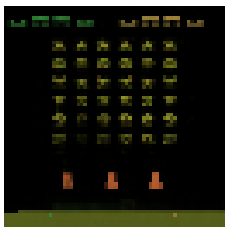
Figure 7: VAE data frames for Raiden

5.2 MDN-RNN

During training the MDN-RNN (Mixture Density Network - RNN) used the latent variables of the VAE as inputs. During runtime, the MDN-RNN uses the previous hidden state of the MDN-RNN and the action taken by the agent to output the next hidden state. This hidden state was then concatenated with the current latent variables which the VAE outputted. Using these values, the controller was able to output the course of actions to take. By using the distributions parametrized by the hidden layer of the RNN we were able to visualize the predictions of the MDN-RNN. Figure 8 shows an input image which was generated by the VAE and the resulting prediction from the MDN-RNN. In this case, the MDN-RNN tried to predict one frame in the future. The images show a slight change (i.e. enemies moving to the right), which seems logical as it's just one frame ahead. This observation leads to the hypothesis that the MDN-RNN could have had a bigger impact on the model's performance if the VAE had encoded important features like lasers and the correct position of the agent. If the VAE was able to encode such features, then the MDN-RNN could have predicted when lasers where going to be shot. This could have made the future prediction of the MDN-RNN more significant than they were during the experiments.



(a) Input image

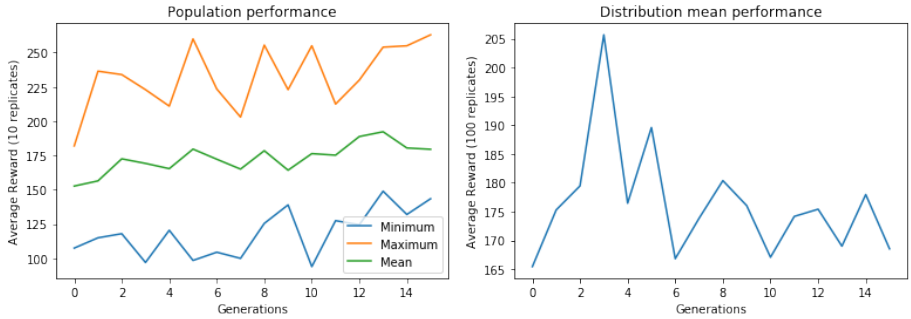


(b) Output RNN

Figure 8: Input and output for RNN after reverse engineering.

5.3 Controller & Genetic Algorithm Optimization

Section 4 (Testing Procedures) describes the experiments that were performed in this section. The controller that was trained on randomly generated data did not learn to obtain a particularly high score, which is probably caused by the reported static representation learned by the Variational Autoencoder. In essence, this controller model has no information about the state of the game, and can thus only learn a general policy that optimizes the reward without input information. This still results in a mean performance of 205, which is the average of 100 episodes (games), as can be observed in Table 2. This means that it outperformed a random agent (which achieves a score of 160). The aforementioned training procedure is reported in Figure 9. When the performance of the algorithm is reported, the 'population performance' means that data that is collected is based on 12 individuals, all tested on 10 games, whereas when 'distribution mean performance' is reported, the mean of the distribution found by CMA-ES is treated as an individual and tested on 100 games. The last metric provides a stable estimate of the position of the parameter settings within the fitness space.



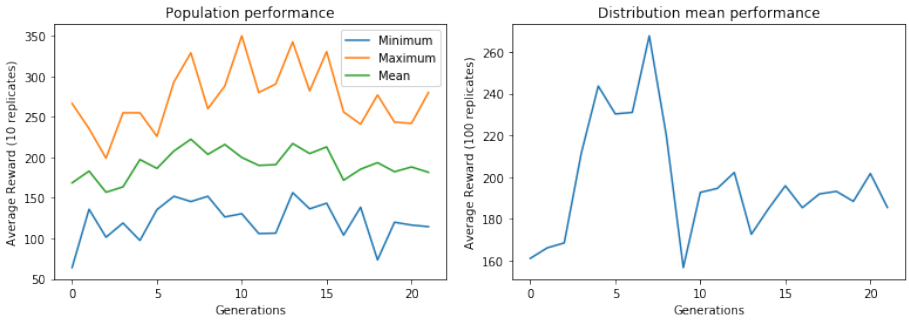
(a) Population performance (b) Mean of distribution performance

Figure 9: Training procedure for controller trained on randomly generated data. The model does not seem to learn very much from the data, but is does perform notably better than random actions.

Table 2: Results of experiments where controller was trained on data generated by a first iteration controller. During this experiment, the controller did not have access to the timestep variable. The distribution mean is based on 100 played game episodes, and the best individual score is based on 10 evaluations / games played by each individual.

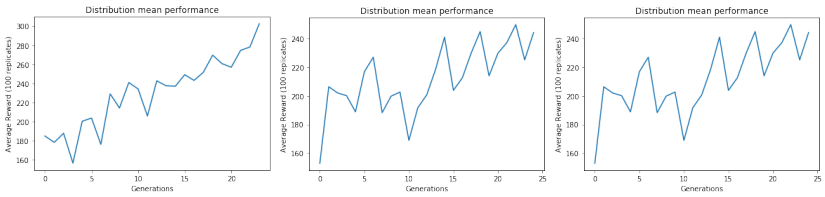
	DeepQN	0th iteration	σ=0.5	σ=0.1	σ=0.1	σ=0.1
Best distribution mean	N/A	205	267	302	249	271
Best individual	581	263	350	**443**	367	390
Best single episode	1075	765	950	905	945	900
Longest survival	N/A	1916	2195	2075	2100	2034

The last four columns of Table 2 show the experiments that were performed where the 1st iteration controller was used to generate data to train the corresponding models. The logic behind this is that the 1st iteration controller was slightly better than random, meaning that the data generated would encode more game-states than data generated by a random agent. This intuition is confirmed by the results shown in the table, as the distribution means and the best individuals are all higher in the 2nd iteration models. For the first experiment, an initial step size of 0.5 was used, but based on the sudden collapse of the performance (see Figure 10, the initial step size was lowered to 0.1, as it was suspected that the large step size caused the algorithm to travel out of the local optimum. Based on the performances of the three replicate experiments in the last three column, a step size of 0.1 permits learning (as can be observed in Figure 11). This was therefore chosen as the initial step size from this point on.



(a) Population performance (b) Mean of distribution performance

Figure 10: Training procedure for second iteration controller with initial step size of 0.5. The model's distribution performance seems to improve, but then collapses.



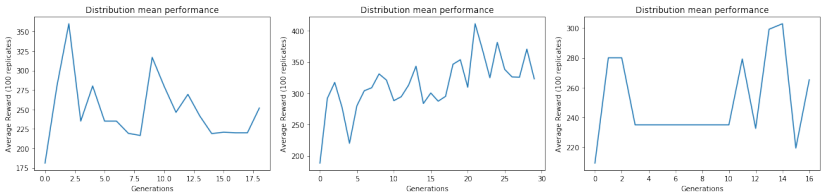
(a) Replicate 1 (b) Replicate 2 (c) Replicate 3

Figure 11: Training procedure for second iteration controller with initial step size of 0.1. The line shown is the performance of the mean parameters of the distribution found by CMA-ES, when treated as an individual. The lower step size seems to prevent the collapse of the controller's performance.

The controller was then given an extra input: the number of timesteps that have passed since the beginning of the game. It was expected that this would be used as a linear scaling factor. Based on our experiments, of which the results can be observed in Table 3, the timestep information is very useful. Note that because the model only has a single layer of variables, and because Space Invaders is non-deterministic, it cannot learn a deterministic optimal path.

Table 3: Results of experiments where controller was trained on data generated by a first iteration controller. During this experiment, the controller had access to the timestep variable. The distribution mean is based on 100 played game episodes, and the best individual score is based on 10 evaluations / games played by each individual.

	DeepQN	iteration1	it2_time1	it2_time2	it2_time3
Best distribution mean	N/A	205	360	411	302
Best individual	**581**	263	360	**544**	398
Best single episode	1075	765	1040	955	985
Longest survival	N/A	1916	2474	2190	2472



(a) Replicate 1 (b) Replicate 2 (c) Replicate 3

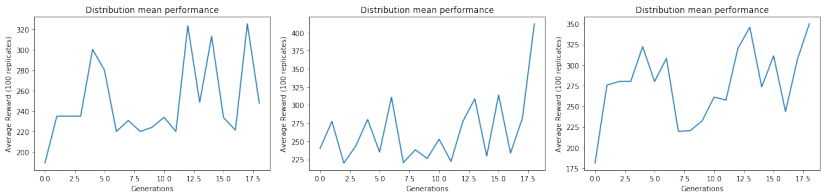
Figure 12: Training procedure for second iteration controller with initial step size of 0.1 and access to the timestep parameter. The line shown is the performance of the mean parameters of the distribution found by CMA-ES, when treated as an individual. The model was tested on 100 games. The lower step size seems to prevent the collapse of the controller's performance.

The addition of timestep information seems to help in the general performance of the algorithm, but it also seems to make the training procedure of the algorithm less reliable (see Figure 12). This may be related to the way the timesteps are given to the controller: the timesteps can be a value in the thousands, while the other outputs are typically much closer to zero. This discrepancy may cause difficulty when trying to find optimal parameter settings.

The last experiment that was performed was training a controller on data that was generated by a second iteration model, i.e. a third iteration model.

Table 4: Results of experiments where controller was trained on data generated by a second iteration controller. During this experiment, the controller had access to the timestep variable. The distribution mean is based on 100 played game episodes, and the best individual score is based on 10 evaluations / games played by each individual.

	DeepQN	random	it2_time1	it2_time2	it2_time3
Best distribution mean	N/A	205	325	412	350
Best individual	**581**	263	427	**513**	472
Best single episode	1075	765	1060	1060	1060
Longest survival	N/A	1916	2119	2408	2119



(a) Replicate 1 (b) Replicate 2 (c) Replicate 3

Figure 13: Training procedure for third iteration controller with initial step size of 0.1. The line shown is the performance of the mean parameters of the distribution found by CMA-ES, when treated as an individual. The model was tested on 100 games.

As can be observed in Table 4 and Figure 13, the third iteration models perform quite well and seem on par with the best controller individual from the second generation. There is however no clear benefit, which is probably due to the fact that the VAE does not encode the environment fully in any case.

6 Conclusion

In this project, we described an implementation of the world model architecture of Space Invaders. Our goal was to expand on the experiments done in the World Models paper, by playing a more complex game and performing multiple iterations of the process, as proposed in the Future Work section of original World Models paper. In general, the linear controller that makes use of the unsupervised representation of the world models, encoded in the variational autoencoder and recurrent neural network is clearly performing better than a random agent, and in some cases its performance is almost on par with Deep Q Network. This is an interesting result because the agent, as far as we can infer from the decoded latent states, has no information on where the projectiles fired by the opponents are. Apparently it is possible to learn a quite effective control policy, even in the absence of this seemingly crucial information. We imagine that with a better representation of the input space, including projectiles, the agent will be able to come up with better policies, even without being explicitly given the timestep variable. In general we find that this approach is quite promising due to the possibility of using genetic algorithm optimization.

7 Future Work

7.1 VAE

As can be observed in section 5.1, the VAE is not able to encode all the important features of the data. To further improve on this, it would be interesting to try a variation of the VAE, which is called InfoVAE. This variation focuses

focuses on information maximization by using a different loss function.

Besides experimenting with InfoVAE, it would also be interesting to experiment with a custom loss function. One thing that was observed is that the black background is not important for the agent, but the VAE will consider it important as it takes up a big part of the environment. So if the VAE predicts most of the environment as black, then the loss will still be considerably low. This could be prevented by punishing the loss function for predicting something as black when it isn't supposed to be black. This could be generalized by checking the environment for colors that take up a large amount of the screen and punishing the model for classifying pixels that aren't of that color, more severely.

During the experiments, the exploding gradient problem was observed when using 16 latent variables. This is also an issue that GANs (Generative Adversarial Networks) experience [16, 17]. A method for preventing this is clipping the gradients. If the experiments with clipping gradients prove to be successful, the model would be able to use a lower amount of latent variables, which in turn could result in a better representation of important variables when combining it with the methods mentioned above.

7.2 RNN

The RNN currently predicts one frame ahead of the current timestep. It would be interesting to see if it is able to predict multiple timeframes ahead such that the controller can take this into account. To achieve this feature, it would be interesting to look into the IndRNN architecture as this this is a more complex architecture which would enable the model to encode more complex patterns/features when e.g. predicting several frames ahead of time, or based on patterns further back in time.

Increasing the predicted amount of frames, also increases the complexity. To prevent the model from becoming overly complex, it would be interesting to experiment with a lower amount of hidden states such that the complexity of the model can be limited. Reducing the complexity of the model helps when predicting several frames ahead, as well as with the training of the CMA-ES (due to lower amount of parameters).

Predicting several frames ahead, could result in the model being able to predict when an agent will die or not. This could be possible as there are indications of the agent dying (See Figure 14, which displays the number of lives[2] which are left at the beginning of the game and just after dying).

[2]The number of lives is not clearly visible due the resizing of the environment, however, the environment only shows yellow-like pixels at that position after losing a life or when starting the game. This means that even a few of those pixels indicate a certain type of event, which the agent could learn from

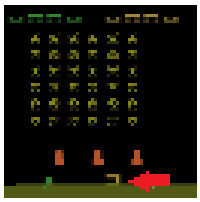


Figure 14: The yellow shape at the bottom of the frame is the 'life counter', that the VAE encodes.

7.3 Controller

Our controller currently consists of a single layer network. For future uses, it would be interesting to experiment with more layers as this would allow the controller to learn nonlinear functions.

At the beginning of the game the agent does not receive any rewards as it hasn't killed any enemies yet. An interesting alternative would be to reward the agent for staying alive. To achieve this, we would like to reward the agent for staying alive by adding the timesteps to the reward. This means that the agent gets rewarded more and more overtime for staying alive. To prevent the agent from focusing solely on surviving (and thus not killing any enemies), we imagine it would be useful to use a logarithmic function such that the reward it can achieve from surviving (i.e. the timesteps) is limited overtime.

8 Implementation

The code that was used for this project can be found on GitHub [3]. The code that was used is based on the original World Model paper [4] and several other codebases such as the PyTorch implementation [5], AppliedDataScience [6] and Hardmaru [David Ha] [7]. Furthermore, we've used the code from AppliedDataScience for multithreading the training of our controller.

Our codebase is divided into several files. The ReadMe file on our GitHub explains how to apply the model with the Space Invaders environment.

[3] https://github.com/mickvanhulst/world_models
[4] https://github.com/worldmodels/worldmodels.github.io
[5] hhttps://github.com/JunhongXu/world-models-pytorch
[6] https://github.com/AppliedDataSciencePartners/WorldModels
[7] https://github.com/hardmaru/WorldModelsExperiments

9 Individual Contribution

9.1 Mick

Mick was responsible for setting up the game environment, integrating the various components of the model (controller, VAE and RNN) and implementing the VAE. Furthermore, the experiments that were performed for the VAE and RNN were done by Mick.

9.2 Jelle

Jelle was responsible for the experimental setup and performing the experiments, including multiple iterations of data generation and model/controller/-VAE/RNN training as well as the (optimization of) the controller.

9.3 Jorrit

Jorrit was responsible for implementing an independent recurrent neural network [18], which could replace the RNN used currently. Unfortunately, due to time constraints and complexity, this is not fully implemented as of right now.

10 References

References

[1] David Ha and Jürgen Schmidhuber. World models. *arXiv preprint arXiv:1803.10122*, 2018.

[2] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin Riedmiller. Playing atari with deep reinforcement learning. *arXiv preprint arXiv:1312.5602*, 2013.

[3] Richard S Sutton. Introduction: The challenge of reinforcement learning. In *Reinforcement Learning*, pages 1–3. Springer, 1992.

[4] Volodymyr Mnih, Adria Puigdomenech Badia, Mehdi Mirza, Alex Graves, Timothy Lillicrap, Tim Harley, David Silver, and Koray Kavukcuoglu. Asynchronous methods for deep reinforcement learning. In *International Conference on Machine Learning*, pages 1928–1937, 2016.

[5] OpenAI. A toolkit for developing and comparing reinforcement learning algorithms. https://gym.openai.com/.

[6] Diederik P Kingma and Max Welling. Auto-encoding variational bayes. *arXiv preprint arXiv:1312.6114*, 2013.

[7] Felix A Gers, Jürgen Schmidhuber, and Fred Cummins. Learning to forget: Continual prediction with lstm. 1999.

[8] Irhum Shafkat. Intuitively understanding variational autoencoders, Feb 2018. https://towardsdatascience.com/intuitively-understanding-variational-autoencoders-1bfe67eb5daf.

[9] Kevin Frans. Variational autoencoders explained, May 2017. http://kvfrans.com/variational-autoencoders-explained/.

[10] Christopher Bonnett. Mixture density networks. http://edwardlib.org/tutorials/mixture-density-network.

[11] mixture-density-networks-with-tensorflow. http://blog.otoro.net/2015/11/24/mixture-density-networks-with-tensorflow/.

[12] 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.

[13] A visual guide to evolution strategies. http://blog.otoro.net/2017/10/29/visual-evolution-strategies/.

[14] Wikipedia. "`https://en.wikipedia.org/wiki/CMA-ES#/media/File:Concept_of_directional_optimization_in_CMA-ES_algorithm.png`, Jun 2018.

[15] Sergey Ioffe and Christian Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift. *arXiv preprint arXiv:1502.03167*, 2015.

[16] Ishaan Gulrajani, Faruk Ahmed, Martin Arjovsky, Vincent Dumoulin, and Aaron C Courville. Improved training of wasserstein gans. In *Advances in Neural Information Processing Systems*, pages 5769–5779, 2017.

[17] Ian Goodfellow, Jean Pouget-Abadie, Mehdi Mirza, Bing Xu, David Warde-Farley, Sherjil Ozair, Aaron Courville, and Yoshua Bengio. Generative adversarial nets. In *Advances in neural information processing systems*, pages 2672–2680, 2014.

[18] Shuai Li, Wanqing Li, Chris Cook, Ce Zhu, and Yanbo Gao. Independently recurrent neural network (indrnn): Building a longer and deeper rnn. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 5457–5466, 2018.