Universiteit Leiden

Reinforcement Learning
Practical Assignment 3: Open Project

Marius M. Kästingschäfer
s3058301

Adéla Šterberová
s2732815

Eric Prehn
s2724731

Abstract

In this paper we compare the world-model based learning agent Dreamerv2 with both an evolutionary strategies and the monte-carlo policy gradient (REINFORCE) method. We also provide the results of an random agent as baseline. We compare the different implementations in terms of model complexity, training time and final performance. We further investigate Dreamers sensitivity to changing the size of the categorical latent space. All models are trained either on the Pong-v0 or Pinball Atari environments.

Word count: 5974

Contents

1 Introduction

Within the last years semi-supervised and unsupervised learning has made tremendous progress. An interesting intersection between reinforcement learning and unsupervised learning has emerged. It is now possible to learn rich representations from high dimensional observations [8] [13]. An artificial agent benefits from having a good representation of dynamics of its surroundings and oneself. Especially advantageous is such a model if it includes a good predictive model of the future [15]. For the purpose of this report we will investigate and compare the Dreamerv2 model-based algorithm.

The main alteration in Dreamerv2 [7] from Dreamerv1 [6] is that the latent space is represented by a collection of 32 categorical variables each with 32 classes, instead of using Gaussian latent variables. The model 'decides' for itself or learns what the categorical variables are for and what each of the classes mean. For example, one categorical variable could encode the location of the agents paddle in Pong-v0 (then this categorical variable can take one of 32 classes to describe the paddles location). However the encoding of the variables that the model learns cannot be known or interpreted. For the purpose of this report we considered the following:

1. How does the Dreamerv2 model compares with more standard methods (such Monte-Carlo policy gradient (REINFORCE) or evolutionary strategies) in terms of training accuracy, sample efficiency and final model performance?
2. What impact does the size of the stochastic latent space have on the performance of the Dreamerv2?

The first question is of high relevance due to the large amount of modelling alternatives that are possibly less computationally expensive. To compare the Dreamerv2 we chose a simple evolutionary strategy (low model complexity) and the Monte-Carlo policy gradient (REINFORCE) algorithm (medium model complexity). We consider the Dreamerv2 itself to have high to very high model complexity. The second question is important for latent space representations in general and of course the answer will depend on the atari environment considered. An attempt was made to investigate this question for the Pong-v0 and the Pinball atari games.
Unfortunately the Dreamerv2 paper refers to itself as a single-GPU model, whereby the authors used one NVIDIA V100 GPU and trained the model for 10 days. This computational requirement limited the extent to which our research question could be investigated. Despite this, the theory behind the Dreamerv2 model was studied and understood and with the help of the LIACS labs a few runs of the Dreamerv2 model were possible.

2 Background Theory

2.1 Agent-environment loop

Reinforcement learning (RL) is often defined as the science of learning to make decisions [14]. 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.

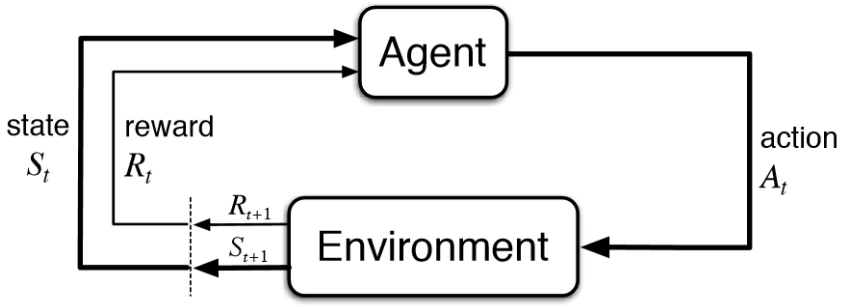


Figure 1: Showing the canonical agent-environment-loop. The agent acts upon the environment and obtains state and reward information from the environment [14].

2.2 Markov Decision Process

A Markov Decision Process (MDP) provides a mathematical framework for defining sequential decision making tasks. Formally a MDP is given by the tuple:

$$\{S, A, T(s'|s,a), R(s,a,s'), \gamma, p_0(s_0)\} \tag{1}$$

Where S and A denote the state and action spaces, respectively. $T(s'|s,a)$ is the transition probability of the environment being in state s' after action a occurs in state s. The reward function $R(s,a,s')$ determines the reward of the transition and $\gamma \in [0,1]$ is a discount factor that can determines how much long-term rewards are decreased. Finally $p_0(s_0)$ is an initial state distribution over S, which determines the starting state of the process.

2.3 Policy, Traces and Value

Agents require a policy π to decide which actions to take. The policy π is a conditional probability distribution that for each possible state specifies the probability of each possible action [ref]. This maps:

$$\pi : S \longrightarrow p(A) \tag{2}$$

Where $p(A)$ can be a discrete or continuous probability distribution. $\pi(a|s)$ then represents a particular probability (density). A parameterized policy $\pi_\theta(a|s)$ then depends on parameters θ and a deterministic policy selects only a single action in every state and maps:

$$\pi : S \longrightarrow A \tag{3}$$

Throughout a MDP the consecutive states, actions and rewards resulting from the MDP can be stored in a trace:

$$h_t^n = \{s_t, a_t, r_t, s_{t+1}, .., a_{t+n}, r_{t+n}, s_{t+n+1}\} \tag{4}$$

Where $r_t = R(s_t, a_t, s_{t+1})$ and taking a trace until the process terminates (reaching a terminal state) is denoted $h_t^\infty = h_t$. Since both the policy and the transition dynamics can be stochastic, the traces can differ from the same starting state s_t. This results in a distribution over traces denoted by $p_\theta(h_0)$, which depends on the policy parameters θ:

$$p_\theta(h_0) = p_0(s_0) \prod_{t=0}^{\infty} \pi_\theta(a_t|s_t) T(s_{t+1}|a_t, s_t) \tag{5}$$

Given a trace h_t its return or cumulative reward is then given by:

$$R(h_t) = \sum_{i=0}^{\infty} \gamma^i r_{t+i} \tag{6}$$

Importantly the outcomes of an MDP can be stochastic and therefore the average, or expected, cumulative reward that a certain policy achieves is key. The average cumulative reward is better known as the value.

The State Value is the average return expected when an agent starts in state s and follows policy π, it is given by:

$$V^\pi(s) = \mathbb{E}_{h_t \sim p(h_t)}[\sum_{i=0}^{\infty} \gamma^i r_{t+i} | s = s_t] \tag{7}$$

2.4 λ-target

Calculating the State Value over an entire episode can be lengthy and computationally expensive. One can consider a trace up to timepoint referred to as a horizon H, instead of trace up to the terminal state. The λ-target is a weighted average of n-step returns for different horizons, where longer horizons are weighted exponentially less. A value of $\lambda \in [0,1]$ can be chosen, were a larger value corresponds to focusing more on long horizons. The formula is given by:

$$V_t = r_t + \gamma_t \begin{cases} (1-\lambda)\mathbb{E}[\sum_{\tau \geq t} \gamma^{\tau - t} r_\tau] + \lambda V_{t+1} & \text{if} \quad t < H, \\ 0 & \text{if} \quad t = H. \end{cases} \tag{8}$$

This is in a general format where the discount factor γ_t is time-dependent. Using this method multiple horizon lengths can be considered and averaged. This can help with reducing variance and also the tuning of λ gives the model the ability to focus more/less on long horizons, as desired.

2.5 Reinforcement Learning Objective Function

The objective of reinforcement learning is to achieve the highest possible average return from the start state [source]:

$$J(\pi) = V^{\pi}(s_0) = \mathbb{E}_{h_0 \sim p(h_0|\pi)}[R(h_0)] \tag{9}$$

It turns out that there is only one optimal value function, which achieves higher or equal value than all other value functions. The aim is to search for a policy that achieves this optimal value function:

$$\pi^*(a|s) = \arg\max_{\pi} V^{\pi}(s) \tag{10}$$

2.6 Actor-Critic Theory

Actor-critic learning is a combination of the value-based approach and policy-based approach. The actor represents the policy approach by calculating the policy function at each state. Then the critic, which represents the value approach, calculates the action-value function at each state. Combining these two approaches brings two improvements. First, in the convergence of the model, which is better than in value methods and the second in better model policies, thanks to reduced variance [11].

2.7 World-model loop

Within world model learning the standard agent-environment loop remains intact, but another 'layer' is added consisting of the agents internal world model. The process is depicted in figure 2.

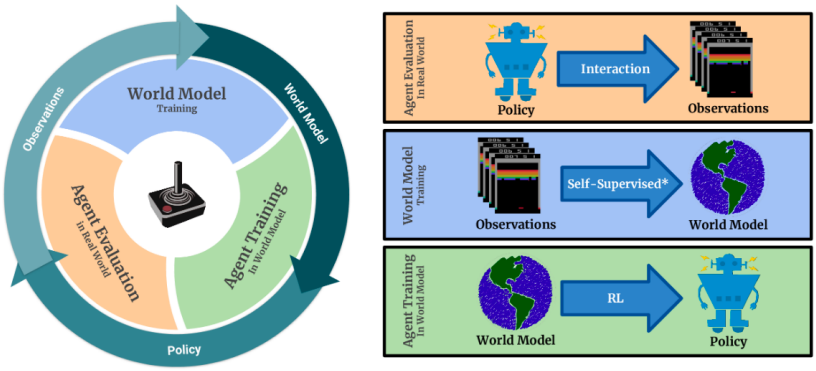


Figure 2: Illustrative description of model-based reinforcement learning [1].

2.8 Environment: OpenAI Gym

Throughout this assignment we will make use of the OpenAI gym [2]. Those games are well suited for testing different algorithms since the problem is not easy to solve but the action space is still limited. We will in particular make use of the Pong-v0 game. Pong is a simple "tennis like" game that features two paddles and a ball. Goal of the game is get as many points as possible, points are obtained when the opponent misses a ball. The game was originally developed by Allan Alcorn and released in 1972 by Atari corporations. The rendered game screen is shown in figure 3.

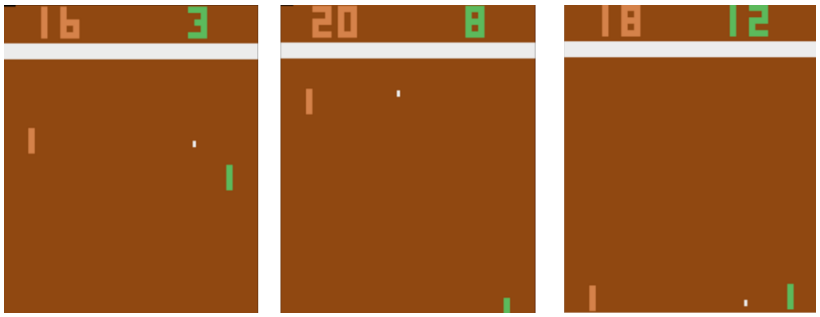


Figure 3: Example frames of the Pong-v0 game [2].

The second game considered within this report is the Video pinball. It is a single-player game. The goal is to collect as many points as possible by keeping the ball in the environment. The original game was programmed by Bob Smith and released by Atari in 1980. The environment can be seen in figure 4.

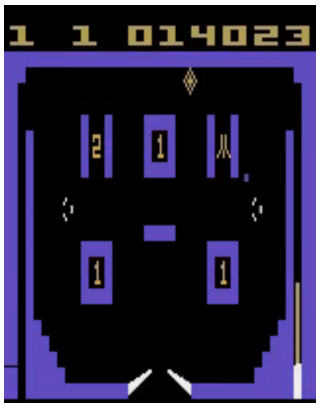
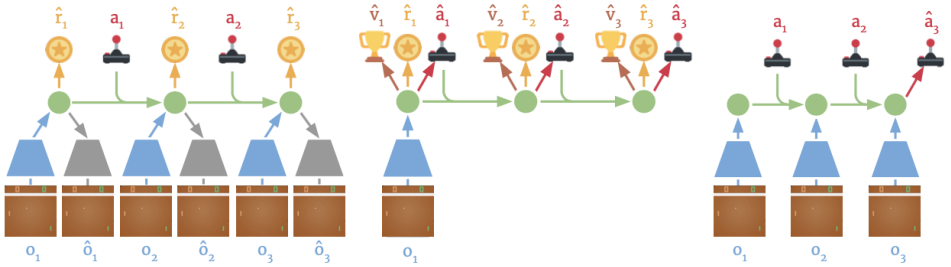


Figure 4: Example frame of pinball [2].

2.9 Dreamer

Dreamer is a model-based reinforcement model that learns a so called 'world model'. The world model is a compact latent space representation of the complex environment dynamics. This world model allows the agent to learn optimal policies within this latent space before applying them in the actual environment. The learning process can thus be separated into three distinct phases: the world model learning, the actor-critic learning and of course applying the learned dynamics in the actual environment. The 3 steps are depicted in figure 5.



(a) Learn dynamics from experience (b) Learn behavior in imagination (c) Act in the environment

Figure 5: Procedure followed by the dreamer algorithm. The image is adopted from [6].

2.9.1 World model learning

World model learning is closely related and enabled through the advancements made within the fields of semi-supervised learning and representation learning. The world model consists of a recurrent model, a representation model, a transition predictor, an image predictor, a reward predictor and a discount predictor. The models are formally introduced in the equation below.

$$\text{World model} \begin{cases} \text{Recurrent model:} & h_t = f_\phi\left(h_{t-1}, z_{t-1}, a_{t-1}\right) \\ \text{Representation model:} & z_t \sim q_\phi\left(z_t \mid h_t, x_t\right) \\ \text{Transition predictor:} & \hat{z}_t \sim p_\phi\left(\hat{z}_t \mid h_t\right) \\ \text{Image predictor:} & \hat{x}_t \sim p_\phi\left(\hat{x}_t \mid h_t, z_t\right) \\ \text{Reward predictor:} & \hat{r}_t \sim p_\phi\left(\hat{r}_t \mid h_t, z_t\right) \\ \text{Discount predictor:} & \hat{\gamma}_t \sim p_\phi\left(\hat{\gamma}_t \mid h_t, z_t\right) \end{cases} \tag{11}$$

The input of the model x_t consists of an environment state generated at time point t. h_t are the deterministic recurrent state of the model at time point t. z_t is the computed posterior stochastic state (which also incorporates information

about the current image). $\hat{z}_t$ is a prior stochastic state that tries to predict the posterior without access to the current image. a_t describes the action taken at time point t. The recurrent model, the representation model and the transition predictor are part of the Recurrent State-Space Model (RSSM). The RSSM is the 'backbone' of the world model, it can be understood as a sequential generative model.

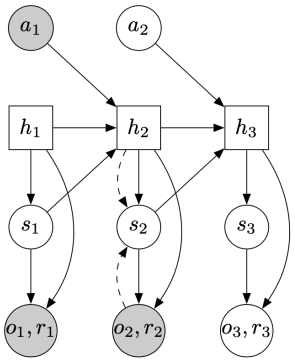


Figure 6: Recurrent State-Space Model [5].

We can now put the different pieces together and describe the full process:

1. An image x_t is together with an recurrent state h_t fed into the *representation model*. We obtain the stochastic state z_t.
2. The *transition predictor* performs the same task as the representation model and uses only the recurrent state h_t.
3. The recurrent state h_t is obtained via the *recurrent model* using the prior recurrent state h_{t-1}, prior stochastic state z_{t-1} and the prior action a_{t-1}.
4. The recurrent state h_t and the stochastic state z_t are feed into the *image predictor* to obtain the reconstructed image $\hat{x}_t$.

For the understanding of the world model the discount predictor and the reward predictor is more straightforward and for the generative model part less important. As the reconstructed image the reward predictor and the discount predictor use the recurrent state h_t and the stochastic state z_t to predict the associated reward $\hat{r}_t$ and the discount $\hat{\epsilon}_t$. The full dreamerv2 world model is displayed in figure 7. All model parts are parameterized using neural networks.

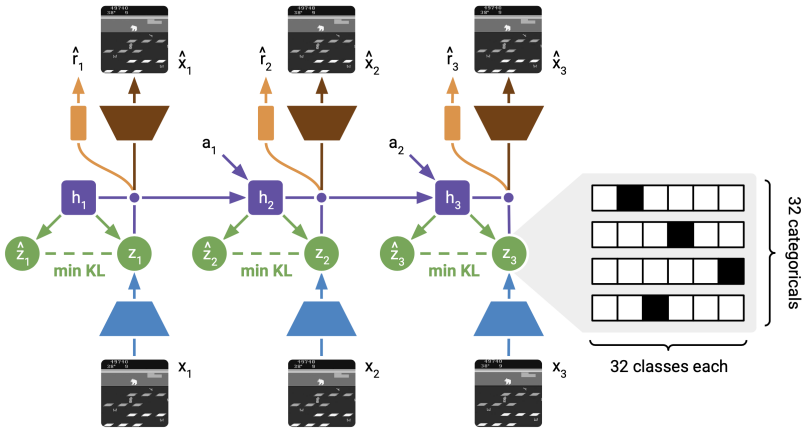


Figure 7: World model learning within the DreamerV2 [7].

Kullback Leibler (KL) Divergence is also known as relative entropy and it measures how different one probability distribution Q is to another probability distribution P. It is given by:

$$KL(Q||P) = \sum_{i \neq j} q_{i,j} \log(\frac{q_{i,j}}{p_{i,j}}) \tag{12}$$

In Dreamerv2 the corresponding KL divergence is between the representation model z_t and the predictor $\hat{z}_t$. This is minimised in an alternative way via KL balancing with the use of extra scaling hyperparameters η_t and η_q. This appears in the loss equation 13 in the transition log loss and the entropy regulariszer, encouraging the model to create similar latent space representations.

Loss function All parts of the world model are trained jointly. This allows to write the single loss function in the following way:

$$\begin{aligned}\mathcal{L}(\phi) \doteq \mathrm{E}_{q_\phi(z_{1:T}|a_{1:T},x_{1:T})}\Big[\frac{1}{T}\sum_{t=1}^{T}\Big(&\underbrace{-\eta_x \ln p_\phi\left(x_t \mid h_t, z_t\right)}_{\text{image log loss}}\underbrace{-\eta_r \ln p_\phi\left(r_t \mid h_t, z_t\right)}_{\text{reward log loss}}\\ &\underbrace{-\eta_\gamma \ln p_\phi\left(\gamma_t \mid h_t, z_t\right)}_{\text{discount log loss}}\underbrace{-\eta_t \ln p_\phi\left(z_t \mid h_t\right)}_{\text{transition log loss}}\underbrace{+\eta_q \ln q_\phi\left(z_t \mid h_t, x_t\right)}_{\text{entropy regularizer}}\Big)\Big]\end{aligned} \tag{13}$$

The image log loss, the reward log loss, the discount log loss and the transition log loss can all be computed in a semi-supervised fashion. The goal is to get the actual model predictions as close as possible to the actual outcomes.The world model can also be interpreted as a sequential variational autoencoder (VAE). Background VAE theory is provided in the Appendix.

2.9.2 Policy learning

The future behaviour of the DreamerV2 agent is determined by an actor-critic model trained only on its world model. The actor-critic works as was already described in 2.6. The actor determines the actions to predict the future sequences of model states, and the critic calculates the future rewards. Both methods work with the learned world model, which stays unchanged during this phase [7]. The actor-critic learning can be seen in figure 8. The learned world model is used to predict the future rewards for each state by the critic, and then the actor maximises the predictions.

Actor critic The actor and critic part of the model are trained together. Both are trained as MLP with 1 million parameters. The actor part is stochastic, trying to maximize the output from the critic. It uses the parameter vector ψ, equation 14, and the result from the actor is a categorical distribution over the actions. The critic part of the model is deterministic, estimating the sum of future rewards and using the parameter vector ξ, equation 15 [7].

$$\text{Actor: } \hat{a}_t \sim p_\psi\left(\hat{a}_t \mid \hat{z}_t\right) \tag{14}$$

$$\text{Critic: } v_\xi\left(\hat{z}_t\right) \approx \mathrm{E}_{p_\phi, p_\psi}\left[\sum_{\tau \geq t} \hat{\gamma}^{\tau-t} \hat{r}_\tau\right] \tag{15}$$

Actor loss function As was already mentioned, the actor's purpose is to predict future actions based on the expected rewards by the critic. The DreamerV2 combines two types of gradient estimators - the Reinforce gradients and the backpropagation gradients, which brings a low variance biased gradient estimate. The entropy regularization term allows for exploration. The function $sg()$ denotes the stop-gradient operation that acts as the identity function in the forward direction, but stops the accumulated gradient from flowing through that operator in the backward direction.

$$\mathcal{L}(\psi) \doteq \mathrm{E}_{p_\phi, p_\psi}\Big[\frac{1}{H-1}\sum_{t=1}^{H-1}\underbrace{\left(-\eta_s \ln p_\psi\left(\hat{a}_t \mid \hat{z}_t\right) \operatorname{sg}\left(V_t^\lambda - v_\xi\left(\hat{z}_t\right)\right)\right.}_{\text{reinforce}}\underbrace{-\eta_d V_t^\lambda}_{\text{dynamics backprop}}\underbrace{\left.-\eta_e \mathrm{H}\left[a_t \mid \hat{z}_t\right]\right)}_{\text{entropy regularizer}}\Big] \tag{16}$$

Critic loss function The critic that predicts the future rewards for the actor for every given state. For better learning of the critic, a λ-target policy is used. The λ-target function is called recursively 17 and returns the weighted average of future steps in horizon H. To focus more on future targets, λ is set to $\lambda = 0.95$.

$$V_t^\lambda \doteq \hat{r}_t + \hat{\gamma}_t \begin{cases} (1-\lambda) v_\xi\left(\hat{z}_{t+1}\right) + \lambda V_{t+1}^\lambda & \text{if} \quad t < H \\ v_\xi\left(\hat{z}_H\right) & \text{if} \quad t = H \end{cases} \tag{17}$$

Therefore, the critic loss function is defined as follows 18:

$$\mathcal{L}(\xi) \doteq \mathrm{E}_{p_\phi, p_\psi}\left[\frac{1}{H-1}\sum_{t=1}^{H-1}\frac{1}{2}\left(v_\xi\left(\hat{z}_t\right) - \mathrm{sg}\left(V_t^\lambda\right)\right)^2\right] \tag{18}$$

Equation 18 involves averaging over horizons of different lengths, here the critic is attempting to accurately estimate the sum of future rewards achieved by the actor from each imagined state.

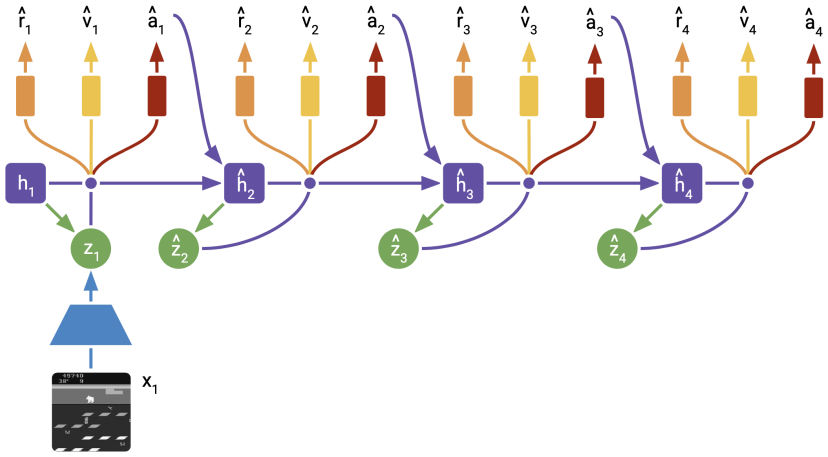


Figure 8: Actor-critic learning within the DreamerV2. [7]

2.10 Evolutionary Strategies

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 optimization algorithm is roughly inspired by natural evolution [4]. The team from OpenAI [12] in their 2017, paper showed that evolutionary strategies can be a viable alternative when compared to standard reinforcement learning algorithms.

Due to great training success with evolutionary strategies for Cartpole we applied the comparable method to learn both Pong and Pinball. Since learning from pixels directly (210x160x3) would be to expensive and since even the cropped and downsampled version of the input would result in a 1-D input of size 6400 we decided to use the ram state. The ram state provides all the important state information and consists of a non-binary 1-D vector of length 128. 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 initialized 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:

$$\mathrm{r} = \sum_i x_i * w_i \tag{19}$$

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:

$$\text{Actions} \begin{cases} \text{Action 1: } a_1 & r > a \text{ and } r <= b \\ \text{Action 2: } a_2 & r > b \text{ and } r <= c \\ \ldots & \\ \text{Action n: } a_n & r > z \text{ and } r <= z \end{cases} \tag{20}$$

Here $a, b, ..., z$ define predefined cutoff-values. The full model architecture is also depicted in figure 9.

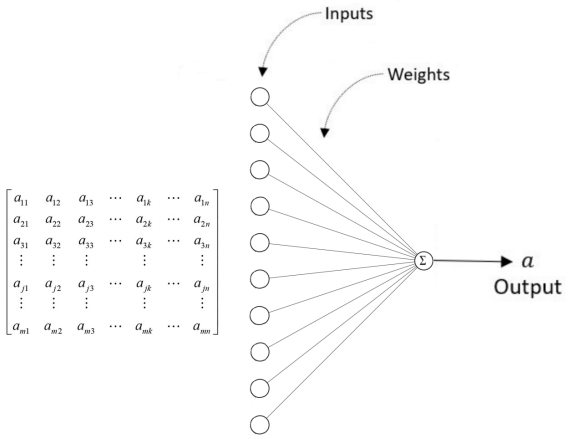


Figure 9: Architecture of the agent trained using evolutionary strategies. The ram state consists of a 1-D vector of length 128 which are weighted using the trained weights. The resulting score is summed and depending on the sum an action is chosen.

2.11 Policy Search

Policy Search involves directly optimising the explicit policy. The objective then becomes an optimisation problem seeking:

$$\theta^* = \arg\max_\theta J(\theta) \tag{21}$$

This report focuses on gradient-based optimisation, whereby the derivative of the objective is used to find the optimum. In the case of a maximization (gradient ascent) each iteration i of the algorithm makes the following update:

$$\theta_{i+1} = \theta_i + \eta \nabla_\theta J(\theta), \tag{22}$$

for learning rate $\eta \in \mathbb{R}^+$.

2.12 Monte-Carlo policy gradient (REINFORCE)

In order to calculate the gradient $\nabla_\theta J(\theta)$ in equation 22, the gradient of the expectation value is required:

$$\nabla_\theta J(\theta) = \nabla_\theta \mathbb{E}_{h_0 \sim p(h_0|\pi)}[R(h_0)] \tag{23}$$

Moving the gradient inside the expectation value and applying the log-derivative trick gives:

$$\nabla_\theta J(\theta) = \mathbb{E}_{h_0 \sim p(h_0|\pi)}[R(h_0) \nabla_\theta \log(p_\theta(h_0))] \tag{24}$$

Noting that the trace probability distribution given by equation 5 only depends on θ in the policy term and that the logarithm of product terms is a sum of terms, the gradient can be expressed as:

$$\nabla_\theta J(\theta) = \mathbb{E}_{h_0 \sim p(h_0|\pi)}[\sum_{t=0}^{\infty} R(h_0) \nabla_\theta \log(\pi_\theta(a_t|s_t))] \tag{25}$$

A popular formulation of the policy gradient only considers the remaining return from the specific timepoint [notes]. The final expression becomes:

$$\nabla_\theta J(\theta) = \mathbb{E}_{h_0 \sim p(h_0|\pi)}[\sum_{t=0}^{\infty} R(h_t) \nabla_\theta \log(\pi_\theta(a_t|s_t))] \tag{26}$$

Given a policy π_θ the optimal parameters θ are sought after, which maximise $J(\theta)$ using gradient ascent. This method is known as the REINFORCE estimator.

3 Implementation Details

Below we clarify our implementation choices. We refer the reader to the information in the README.pdf for futher information.

3.1 Random agent (Baseline)

The implementation of the random agent was trivial. For the pong game agent draws a random number between 0 and 1, if the number is smaller than 0.5 the agent goes left, otherwise the agent goes right. For the pinball game the agent directly draws the possible actions from the action space and executes them.

3.2 Dreamerv2

For the experiments with the DreamerV2 agent, we used code written by Danijar Hafner (Copyright (c) 2020 Danijar Hafner) [3]. The hyperparamater space is large and due to the infeasible computational cost of searching this space, the default hyperparameters were not altered. In Table 1 a summary of default hyperparamaters is provided.

Hyperparameter Default	Origin	Type	
$\eta_x = \frac{1}{64\times64\times3}$	World Model	Constant	
$\eta_r = 1$	World Model	Constant	
$\eta_\gamma = 1$	World Model	Constant	
$\eta_t = 0.08$	World Model	Constant	
$\eta_q = 0.02$	World Model	Constant	
$\eta_s = 0.9$	Actor Loss Function	Constant	
$\eta_d = 0.1 \rightarrow 0.0$	Actor Loss Function	Varies (annealing)	
$\eta_e = \cdot 10^{-3} \rightarrow 3 \cdot 10^{-4}$	Actor Loss Function	Varies (annealing)	
$\lambda = 0.95$	λ-target	Constant	

Table 1: Hyperparameter Details

The Actor loss function (parameters ψ) and Critic loss function (parameters ξ) optimization avail of the Adam optimizer.

The sequence of image inputs x_t are down-scaled to be grayscale images of size (64×64).

In the world model the training sequence of images x_t are encoded by a Convolutional Neural Network. This image embeddding then enters a Multi-Layer Percpetron (MLP) network along with the deterministic recurrent state h_t. The RSSM uses a Gated Recurrent Unit and the image predictor is a De-Convolutional Neural Network (which can be seen as the inverse of a CNN). MLPs are used for the transition predictor $\hat{z}_t$, the reward predictor $\hat{r}_t$ and discount predictor $\hat{\gamma}_t$. Overall the world model uses a total of 20M trainable parameters [7]. The model is involved and complex, in order to be a self-contained report, an overview of Neural Networks theory is provided in the Appendix.

3.3 Evolutionary Strategies

The method was implemented using python's standard library. The pseudocode can be found in appendix figure 19. 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 seperate 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 for pong and across 3 games for pinball. This decreased fluctuations and improved the chance that a generation used for weight updating is indeed stronger in terms of performance.

3.4 Monte-Carlo policy gradient (REINFORCE)

As another means to evaluate the Dreamerv2 world model, our favorite algorithm REINFORCE was implemented for Pong-v0. REINFORCE was not implemented for Pinball because the episode lengths are very long. Considering this along with it being a more complex environment than Pong-v0, along with the very small pixel size of the ball in Pinball; the training time/success would have been sub-optimal. Still, implementing REINFORCE on Pong-v0 (less complex environment) would allow comparisons to be drawn with the Dreamerv2 model.

The Monte-Carlo policy gradient method was implemented using a neural network to model the probabilistic policy function π_θ. PyTorch software [10] was used to train the network where the inputs are the state s_t and the output is

a vector of probabilities for each action. For the REINFORCE algorithm two possible Pong-v0 actions were chosen (move up or down). The corresponding 2 outputs of the network are probabilities determined by the softmax function:

$$\sigma(\vec{z_i}) = \frac{e^{\vec{z_i}}}{\sum_j^K e^{\vec{z_j}}} \tag{27}$$

Where K is the total number of classes and $\vec{z_j}$ is the j'th value of the input vector to the softmax function.

The next action is selected by sampling from these probabilities, producing a new state s_{t+1}, this next state s_{t+1} is fed into the network and the process repeats until a terminal state has been reached.
This produces a trace h and by keeping track of the cumulative rewards and gradients at every step, the automatic differentiation software (PyTorch) backpropagates the loss and updates the weights (parameters θ) of the network according to equation 22. The Loss function $L(\theta)$ used for backpropagation following every trace is:

$$L(\theta) = -\sum_{t=0}^{\infty} R(h_t) \log(\pi_\theta(a_t|s_t)) \tag{28}$$

Which follows clearly from equation 26. PyTorch seeks the minimum of a loss function, a minus sign is included to account for the desired maximisation of $J(\theta)$.

3.5 Monte-Carlo policy gradient (REINFORCE): Pre-processing

Initially the REINFORCE algorithm was implemented without pre-processing the observations and the model failed to learn. The Pong-v0 observations or states consist of (210x160x3) RGB images which contain redundant information such as the upper section which displays the current score. The images were gray-scaled as the color of objects is not relevant, only that the background differs from the ball and bats. The red channel was kept and the then (210x160) image was cropped and downsampled to a (80x80) image, which was unrolled into a 1-D input of size 6400.

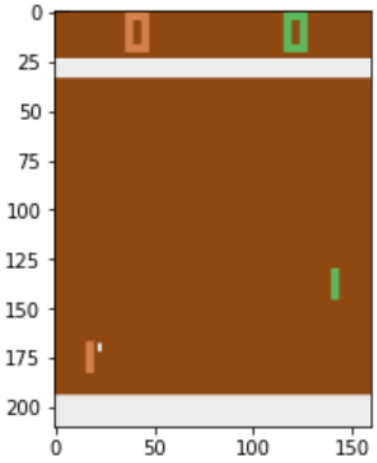


Figure 10: Example state before cropping, gray-scaling and downsampling.

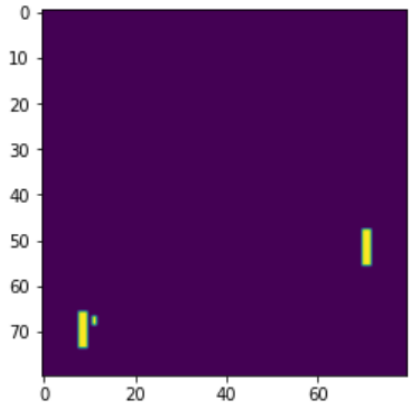


Figure 11: Corresponding state which gets unrolled into 1-D input vector.

After being unrolled into a 1-D vector of length 6400, additional pre-processing had to be undertaken due to the fact that single observations s_t don't provide information about the velocity of the ball in Pong-v0. Only by taking the difference between s_t and s_{t-1} can this information be included. Therefore as a final pre-processing step the input vector was made to be the difference between the current state and the previous state.

3.6 Monte-Carlo policy gradient (REINFORCE): Final Network

The Final Network (most successful network created), intended for comparison with the Dreamerv2 model, consisted of

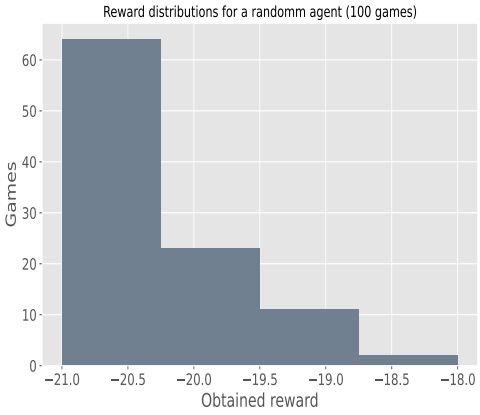
- Input Layer: 1-D state difference vector of length 6400 .
- Fully-Connected Layer of size 200 with no bias.
- Dropout Layer with dropout rate of 0.1 followed by ReLu (equation 30) activation.
- Output Layer: Softmax activation with two outputs.

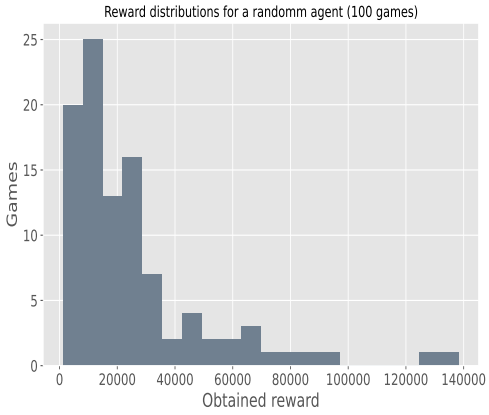
The discount factor was set to $\gamma = 0.995$ and the RMSProp optimiser learning rate was set to 0.001. In order to account for the fact that one Pong-v0 game consists of the best of 21 rounds, the cumulative rewards were reset after every round throughout game episodes. The pseudocode for the REINFORCE algorithm is provided in the Appendix Section.

4 Results

4.1 Random agent (Baseline)

To obtain a valid 'lower-bound' for our experimental results we evaluated a random agent in both environments. For the pong game we a mean reward of -20.49 over 100 games. The full reward distribution is displayed in figure 4.1. The mean reward obtained by the random agent within the pinball game across 100 games is 24973.2. The reward distribution is depicted in figure 4.1.





4.2 Dreamerv2

The computational cost of running the Dreamerv2 is considerable and unfortunately a comprehensive analysis was not achievable. Despite this the Dreamerv2 model was implemented on Pong-v0 and Pinball for as long as was possible using LIACS lab machines. We let the DreamerV2 agent run for an accumulated amount of 192 hours which resulted in 20-25 million steps. In figure 12 we can see results of the DreamerV2 agent with default parameters on the Pong-v0 game. And figure 13 shows the results from the DreamerV2 with a smaller latent space (DreamerV2-24) on the same game. We used 24 classes and 24 categories for learning the world model and classification of learning images. And the difference between these two agents is as expected:
The classical DreamerV2 12 obtained higher rewards in the first steps of the training than the DreamerV2-24. However, the training of the DreamerV2-24 agent was significantly faster than the training of the classical DreamerV2.

Figure 14 shows the results of the DreamerV2 agent on the Video Pinball game. This game was chosen because it demonstrates the shortcomings of the DreamerV2. We can observe that the obtained reward of the DreamerV2 is meagre, close to the zero or the average of the human player. The score does not approach the average of the Rainbow agent even close and the DreamerV2 agent does not improve in time. The temporary improvements in obtained reward are caused more likely by random factors. A possible explanation for this behaviour is that the environment of the Video Pinball is changeable, and most of the changes does not have any influence on the score of the game. On the other hand, the most considerable impact on the game's score (reward) and prediction of the future action of the bats has the ball whose size is only one pixel. Hence, the classification of the training images and prediction of the future action based on the one pixel in the image can not be precise.

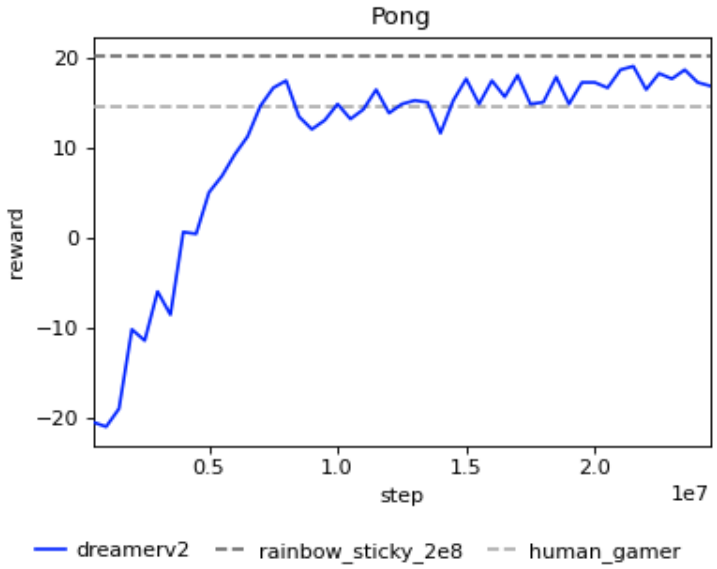


Figure 12: Reward obtained by the DreamerV2 agent in the Pong game, compared to human gamer average score and Rainbow [9] agent average.

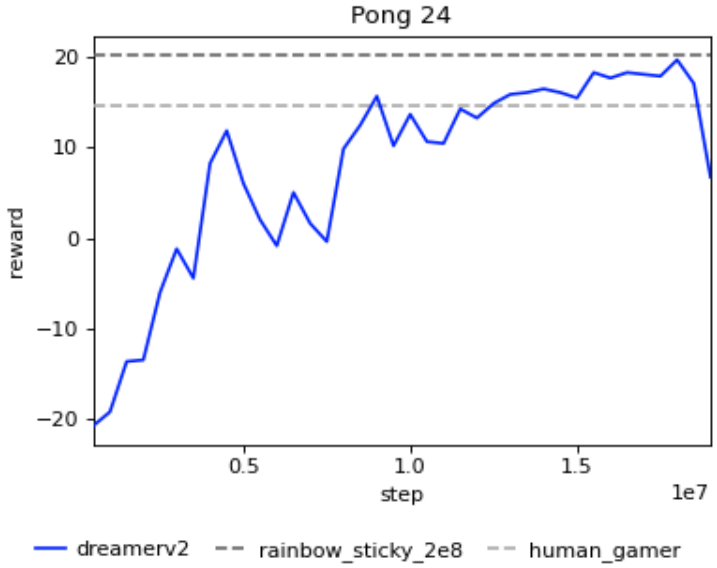


Figure 13: Reward obtained by the DreamerV2-24 agent in the Pong game, compared to human gamer average score and Rainbow [9] agent.

4.3 Evolutionary Strategies

For the evolutionary strategies we ran several different parameter configurations, 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 are provided within the appendix. In this section we will mainly focus on the aggregated result overview and on the high level insights that we gained. 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.

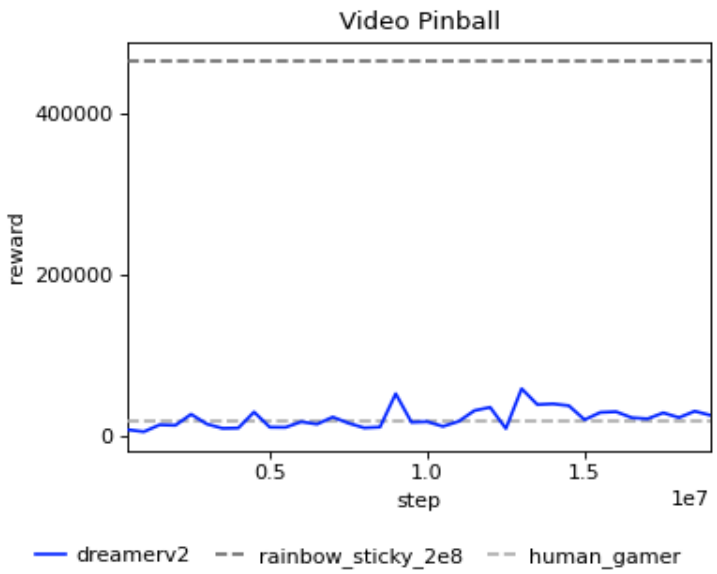


Figure 14: Reward obtained by the DreamerV2 agent in the Video Pinball game, compared to human gamer average score and Rainbow [9] agent.

Results Pong

Model	Training Episodes	Generational changes	Best Result
Noise 0.4	1000	7	-13
Noise 0.4 + sparse	1000	4	-12
Noise 0.8	1000	7	-13.5
Noise 0.8 + sparse	1000	8	-13.5
Noise 1.2	1000	10	-15
Noise 1.2 + sparse	1000	3	-14

Overall it seems like smaller weight 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.

Results Pinball

Model	Training Episodes	Generational changes	Best Result
Noise 0.4	1000	12	60.000
Noise 0.4 + sparse	1000	14	160.000
Noise 0.8	1000	16	60.000
Noise 0.8 + sparse	1000	0	0
Noise 1.2	1000	22	120.000
Noise 1.2 + sparse	1000	10	80.000

For the pinball game we found a higher amount of weight updates. This can be explained to the wider range of obtainable reward scores which made differentiating individual agents easier. For on run (using noise level 0.8 and a sparse weight update) we observed the 'collapse' of weight updates due to missing reward. As initially mentioned the sensitivity towards the initial weight configuration might have caused this problem. Especially during the Pinball game it was difficult to find initial weight configurations that lead to a diverse enough action spectrum to generate any kind of reward. The best agent nevertheless obtained a reward of 160.000.

4.4 Monte Carlo Policy Gradient (REINFORCE)

The computational cost of training the model increases as the models performance improves. This is because initially the agent loses badly and there are approximately 20 rounds per episode. As it learns the agent performs better and hits the ball more often and wins more rounds, resulting in increased lengthy rounds and more rounds per episode. The purpose of running REINFORCE experiments was to provide a comparison with the Dreamerv2 world model. The REINFORCE Final Network results are displayed in figure 15.

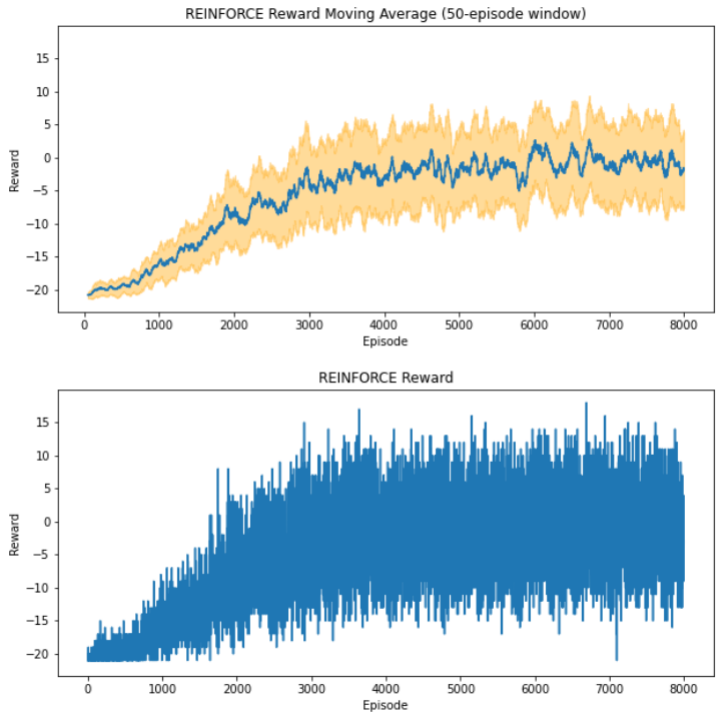


Figure 15: REINFORCE agent reward. Upper plot considers the 50-episode window moving average and shaded orange represents window standard deviation. Lower plot is exact results displaying the instability of the REINFORCE agents performance.

The agent is successful in being an equal opponent, however the performance of the REINFORCE agent plateaus after approximately 4000-5000 episodes of training. The models inability to continue improving performance despite the Pong-v0 specific pre-processing is evident, specifically the frame difference $s_t - s_{t-1}$ as input. With further optimisation and hyperparameter tuning the model could marginally improve, however overall the Dreamerv2 world model strongly outperforms REINFORCE.

4.5 Summary of the results

Here we compare the three trained models, namely the evolutionary strategies, REINFORCE and the Dreamerv2 together with a baseline in terms of amount of parameters, the model complexity, the obtained results and the input the models require.

Results Pong

Model	Parameters	Complexity	Approximate Result	Training Time	Input Size
Baseline	n/a	n/a	-20	n/a	n/a
Evolutionary strategies	128	low	-12	6h	128
REINFORCE	1.280.400	medium	0	13h	6.400
Dreamerv2	22.000.000	very high	+15	192h	100.800

The amount of parameters is by a large margin highest for the Dreamerv2, followed by the REINFORCE algorithm and the evolutionary strategies. The amount of parameters for the different models differs by several orders of magnitude. Within the table the complexity is referring to the model complexity. We took into account factors such as implementation time, amount of parameters and how difficult it is to fine-tune the model (the amount of hyperparameters). The input refers to the dimensionality of the input that was feed into the model during training. The evolutionary agent was not trained using pixel values but received 'precompressed' ram inputs. The input for the REINFORCE agent was preprocessed, resulting in a 1-D vector. Only the Dreamer used the full pixel input of 210x160x3 pixels resulting in an input size of 100.800.

Results Pinball

Model	Parameters	Model complexity	Best Result	Input size
Baseline	n/a	n/a	24973.2	n/a
Evolutionary strategies	128	low	160.000	128
Dreamer	22.000.000	very high	50.000	100.800

5 Conclusion

1. How does the Dreamerv2 model compares with more standard methods (such Monte-Carlo policy gradient (REINFORCE) or evolutionary strategies) in terms of training accuracy, sample efficiency and final model performance?
2. What impact does the size of the stochastic latent space has on the performance of the Dreamer?

Overall the amount of parameters and the model complexity directly correspond to the training time the models required. While a more complex model such as the Dreamer is able to obtain reasonable results within both the pinball and the pong game, less complex models such as the evolutionary strategy fail to do so for the pong game. The REINFORCE algorithm which could be considered a model of 'medium' complexity significantly outperformed the baseline and the evolutionary algorithm on the Pong game but failed to fully solve the game. While the dreamer is an architecture able to solve all 52 Atari games (not including Montezuma's revenge) this generality comes at the price of low sample efficiency. The main reason for this is the expensive process of world model learning which required the agent to process $2e^7$ environment frames.

For the second question we observed that the smaller latent space did not significantly alter the final agents performance. This however is likely due to the low complexity of the pong game itself. In this regard the size of the latent space representation becomes another hyperparameter that needs to be fine-tuned to the task at hand. After having corresponded with the original author of the Dreamerv2 paper this assumption was indeed confirmed.

Acknowledgement: We would like to thank Aske for introducing us to the topic of reinforcement learning and making his book freely available. We also thank the TA's for helpful discussions and guidance. We learned a lot during this course and had fun along the way!

References

[1] Model-based reinforcement learning for atari. URL https://sites.google.com/view/modelbasedrlatari/home.

[2] Greg Brockman, Vicki Cheung, Ludwig Pettersson, Jonas Schneider, John Schulman, Jie Tang, and Wojciech Zaremba. Openai gym, 2016. URL http://arxiv.org/abs/1606.01540. cite arxiv:1606.01540.

[3] Danijar Hafner. Dreamerv2 (version 2). URL https://github.com/danijar/dreamerv2. accessed:2021-04-01.

[4] 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.

[5] Danijar Hafner, Timothy P. Lillicrap, Ian Fischer, Ruben Villegas, David Ha, Honglak Lee, and James Davidson. Learning latent dynamics for planning from pixels. *CoRR*, abs/1811.04551, 2018. URL http://arxiv.org/abs/1811.04551.

[6] Danijar Hafner, Timothy Lillicrap, Jimmy Ba, and Mohammad Norouzi. Dream to control: Learning behaviors by latent imagination. 2020. URL https://arxiv.org/pdf/1912.01603.pdf.

[7] Danijar Hafner, Timothy P Lillicrap, Mohammad Norouzi, and Jimmy Ba. Mastering atari with discrete world models. In *International Conference on Learning Representations*, 2021. URL https://openreview.net/forum?id=0oabwyZbOu.

[8] Irina Higgins, Arka Pal, Andrei Rusu, Loic Matthey, Christopher Burgess, Alexander Pritzel, Matthew Botvinick, Charles Blundell, and Alexander Lerchner. Darla: Improving zero-shot transfer in reinforcement learning. In *Proceedings of the 34th International Conference on Machine Learning - Volume 70*, ICML'17, page 1480–1490. JMLR.org, 2017.

[9] H. Van Hasselt T. Schaul G. Ostrovski W. Dabney D. Horgan B. Piot M. Azar M. Hessel, J. Modayil and D. Silver. Rainbow: Combining improvements in deep reinforcement learning. *Thirty-Second AAAI Conference on Artificial Intelligence, 2018.*

[10] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Kopf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. Pytorch: An imperative style, high-performance deep learning library. In H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett, editors, *Advances in Neural Information Processing Systems 32*, pages 8024–8035. Curran Associates, Inc., 2019. URL http://papers.neurips.cc/paper/9015-pytorch-an-imperative-style-high-performance-deep-learning-library.pdf.

[11] Aske Plaat. *Learning to Play - Reinforcement Learning and Games*. Springer, 2020. ISBN 978-3-030-59237-0. doi: 10.1007/978-3-030-59238-7. URL https://doi.org/10.1007/978-3-030-59238-7.

[12] Tim Salimans, Jonathan Ho, Xi Chen, Szymon Sidor, and Ilya Sutskever. Evolution Strategies as a Scalable Alternative to Reinforcement Learning, September 2017. URL http://arxiv.org/abs/1703.03864.

[13] Pierre Sermanet, Corey Lynch, Yevgen Chebotar, Jasmine Hsu, Eric Jang, Stefan Schaal, Sergey Levine, and Google Brain. Time-contrastive networks: Self-supervised learning from video. In *2018 IEEE International Conference on Robotics and Automation (ICRA)*, pages 1134–1141, 2018. doi: 10.1109/ICRA.2018.8462891.

[14] Richard S. Sutton and Andrew G. Barto. *Reinforcement Learning: An Introduction*. A Bradford Book, Cambridge, MA, USA, 2018. ISBN 0262039249.

[15] P. J. Werbos. Applications of advances in nonlinear sensitivity analysis. In *Proceedings of the 10th IFIP Conference, 31.8 - 4.9, NYC*, pages 762–770, 1981.

6 Appendix

6.1 Deep Learning

6.1.1 Supervised Learning

In Supervised Learning the data set of inputs and matching outputs is subject to an algorithm that attempts to learn this mapping or functional,

$$f_{ML} : \vec{v_i} \rightarrow y_i, \tag{29}$$

where each input of the data set, $\vec{v_i}$, called an **example**, (whose individual components are called **features**) is mapped to the corresponding output of the dataset, called a **target**. The algorithm attempts to create a one-to-one correspondence between inputs and outputs. The form of inputs and outputs can vary, for example the **target** may also be a vector, rather than a scalar.

6.1.2 Neural Networks

Artificial Neural Networks (ANN's) provide a general, practical method for learning real-valued, discrete-valued, and vector-valued functionals from **examples**. A Neural Netwrok consists of an input layer, hidden layer(s) and a final output layer. The layers consist of many units, which connect to all units in adjacent layers. Data enters the input layer, each unit in this layer corresponds to a **feature** of the **example**. Every unit receives inputs adjusted by connection weights and processes these to form outputs. This output is activated or sent to the next layer's units, if a weighted sum (Σ) satisfies a condition specified by the activation function.

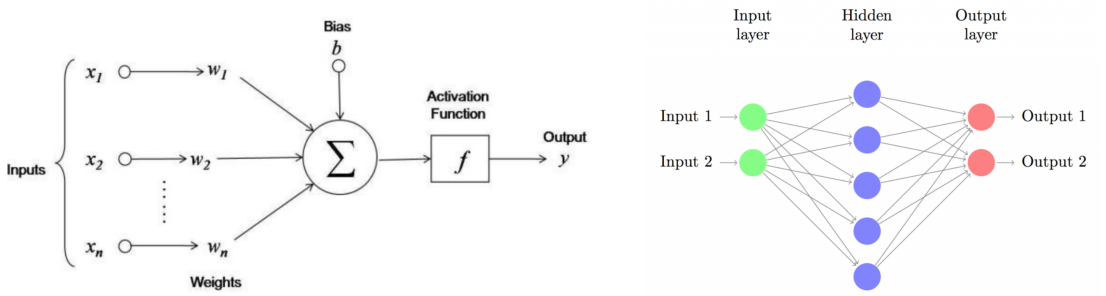


Figure 16: (a) An example of a Neural Network with no hidden layers and a single output. (b) An overview of a Neural Network with one hidden layer and two outputs.

There exist many forms of activation functions, the most common being the Rectified Linear Unit (ReLU) function,

$$g(s) = max[0, s]. \tag{30}$$

For a general neural network with K hidden layers, with input vector $\mathbf{x}$ (layer 0), the first and second layer are expressed as:

$$\mathbf{h_1} = w_{ij}^{0,1}\mathbf{x} + \mathbf{b_0} \tag{31}$$

$$\mathbf{h_2} = w_{ij}^{1,2}\mathbf{h_1} + \mathbf{b_1} \tag{32}$$

The weights between unit *i* in the previous layer and unit *j* in the current layer are labelled w_{ij}. The bias vector ensures that a constant term is present in the layers, which may be required for successful ML (by shifting the activation function). The final output is then given by,

$$\hat{\mathbf{y}} = w_{ij}^{K(K-1)}\mathbf{h_K} + \mathbf{b_K}. \tag{33}$$

Thus for such a fully-connected Neural Network the number of parameters to "learn" grows rapidly with input size and number of layers.

6.1.3 Convolutional Neural Networks

CNN's are a specialised form of neural networks for processing data with a grid like topology. Examples of such data are images (pixels form a grid) or time-series data (1-D grid taking samples at equally spaced time intervals). CNN's employ a form of convolution in one or more layers rather than matrix multiplication. 1-D real valued convolution of a general function *f(t)* and a weighted function *w(t)* is mathematically defined as,

$$(f \circledast w)(t) = \int_{-\infty}^{\infty} f(\tau) w(t-\tau) d\tau. \tag{34}$$

In convolutional network terminology, the first argument to the convolution is referred to as the **input**, and the second as the **kernel**. The output is referred to as the **feature map**.
For time-series data, time is discretised and the convolution can be expressed by,

$$(f \circledast w)(t) = \sum_{\tau=-\infty}^{\tau=\infty} f(\tau) w(t-\tau). \tag{35}$$

For ML purposes the inputs are usually a multidimensional array of data, and the kernel a multidimensional array of parameters (weights) that are adapted by the learning algorithm. The infinite sum is then replaced by finite summation over array elements. For example, if the input is a 2-D image (array of pixels labelled by $I(i,j)$), the kernel can be 2-D array $W(m,n)$, resulting in an output array S, satisfying,

$$S(i,j) = (I \circledast W)(i,j) = \sum_m \sum_n I(m,n) W(i-m, j-n). \tag{36}$$

In contrast to typical ANN layers, where every unit interacts with every unit in adjacent layers via matrix multiplication, CNN's can have fewer interactions or connections. This is accomplished by making the kernel size smaller than the input.
CNN's can therefore be more efficient as fewer parameters have to be stored. Convolution is an extremely efficient way of describing transformations that apply the same linear transformation of a small local region across the entire input [6].

Multi-layer ANNs can be constructed from combinations of CNN's layers and fully-connected layers. Further layers can be added to avoid **over-fitting** such as the Keras Dropout layer which randomly removes a fraction of the units of a layer (helping the model to not become too specialised to the training data).
Deep Learning involves applying ML algorithms to ANN's. The aim of these algorithms is to construct the desired functional or mapping by minimizing corresponding loss functions. The optimisation of the loss functions can be achieved via a number of methods, many of which are based on or similar to the method of stochastic gradient descent (SGD).

6.1.4 Method of Stochastic Gradient Descent

Noting that loss functions are functions of the ANNs' weights, a global minimum of the relevant loss functions w.r.t to these weights, over the training set, is sought after. This well-known method involves iteratively moving in the direction of the steepest descent, which is the negative of the gradient, towards a minimum. The weights vector is updated by ,

$$\theta_{i+1} = \theta_i + \eta \nabla_\theta J(\theta), \tag{37}$$

for learning rate $\eta \in \mathbb{R}^+$. In standard gradient descent, the error is summed over all examples before updating weights, whereas in stochastic gradient descent weights are updated upon examining each training **example**. A more advanced optimisation method that is based on SGD is the Adaptive Moment Estimation optimiser.

6.1.5 ADAM

Optimizers can take into the consideration the momentum by combining weight updates with the previous weight update. Adaptive Moment Estimation (ADAM) considers the momentum $\vec{m}$ and adaptive learning rates $\vec{v}$ as estimates of moments of $\nabla_{\vec{\theta}} L(\vec{\theta})$. The update rules are:

$$\vec{m}_{t+1} = \beta_1 \vec{m}_t + (1-\beta_1) \nabla_{\vec{\theta}} L(\vec{\theta}) \tag{38}$$

$$\vec{v}_{t+1} = \beta_2 \vec{v}_t + (1-\beta_2) (\nabla_{\vec{\theta}} L(\vec{\theta}))^2 \tag{39}$$

$$\vec{\theta}_{t+1} = \vec{\theta}_t - \frac{\eta}{\sqrt{\vec{v}_{t+1} + \epsilon}} \vec{m}_{t+1} \tag{40}$$

Where β_1 and β_2 are hyper-parameters.

6.1.6 RMSProp

Let $L(\vec{w})$ represent the loss function as a function of the weights $\vec{w}$, then RMSProp updates the weights by also normalizing each gradient by a moving average of squared gradients. The update rules are:

$$\vec{v}_{t+1} = \rho \vec{v}_t + (1 - \rho)(\nabla_{\vec{w}} L(\vec{w}))^2 \tag{41}$$

$$\vec{w}_{t+1} = \vec{w}_t - \frac{\eta}{\sqrt{\vec{v}_{t+1} + \epsilon}} \nabla_{\vec{w}} L(\vec{w}) \tag{42}$$

Where ϵ is a small value to avoid division by zero, ρ is the decay rate.

6.1.7 Dropout Layers

The Dropout layer which randomly removes a fraction of the units of a layer can be implemented to help the model to not become too specialised to the training data. This regularisation reduces overfitting. During training, a Dropout Layer randomly zeroes some of the elements of the input tensor with probability p using samples from a Bernoulli distribution [10].

6.2 Variational Autoencoders (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:

$$\vec{z} = \vec{\mu} + \vec{\varepsilon} \odot \vec{\sigma} \tag{43}$$

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 $n = 3$:

$$\vec{z} = \begin{pmatrix} z_1 \\ z_2 \\ z_3 \end{pmatrix} = \begin{pmatrix} \mu_1 \\ \mu_2 \\ \mu_3 \end{pmatrix} + \begin{pmatrix} \varepsilon_1 \sigma_1 \\ \varepsilon_2 \sigma_2 \\ \varepsilon_3 \sigma_3 \end{pmatrix} \tag{44}$$

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 17.

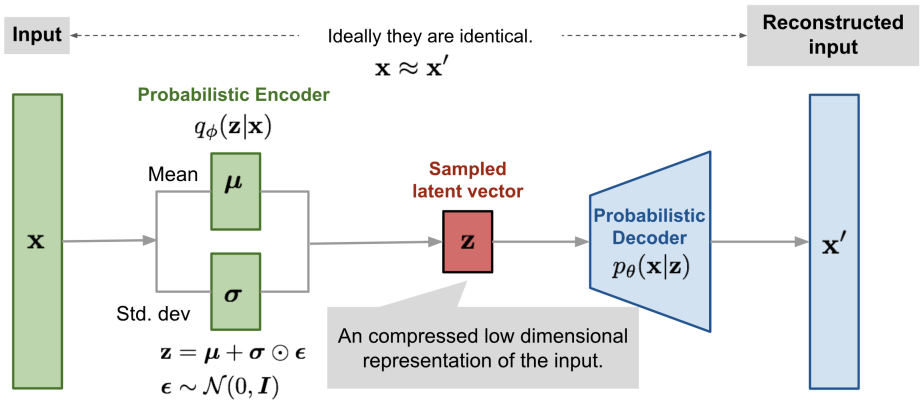


Figure 17: Illustrative example of VAE model.

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.

6.3 REINFORCE Psuedocode

Algorithm 3: Monte Carlo policy gradient (REINFORCE)

Input: A differentiable policy $\pi_\theta(a|s)$, parametrized by $\theta \in \mathbb{R}^d$.
A learning rate η.
Initialization: Randomly initialize θ in $\mathbb{R}^d$.
while *not converged* **do**
  grad $\leftarrow 0$
  for $m \in 1, .., M$ **do**
    Sample trace $h_0 = \{s_0, a_0, r_0, s_1, .., s_{n+1}\}$ following $\pi_\theta(a|s)$
    $R \leftarrow 0$
    for $t \in n, .., 1, 0$ **do**
      $R \leftarrow r_t + \gamma \cdot R$ `/* Backwards through trace */`
      grad $+= R \cdot \nabla_\theta \log \pi_\theta(a_t|s_t)$ `/* Add to total gradient */`
    end
  end
  $\theta \leftarrow \theta + \eta \cdot$ **grad**
end
Return $\pi_\theta(a|s)$

Figure 18: Pseudo code for Monte Carlo Policy gradient. Note in our implementation $m = 1$ and the rewards are standardized by removing the mean and scaling to unit variance.[11]

6.4 Evolutionary Strategies Pseudocode

Algorithm 1 Evolution Strategies

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

Figure 19: Pseudo code for Evolutionary Strategies [12]

6.5 Results Evolutionary Agent

6.5.1 Results Pong

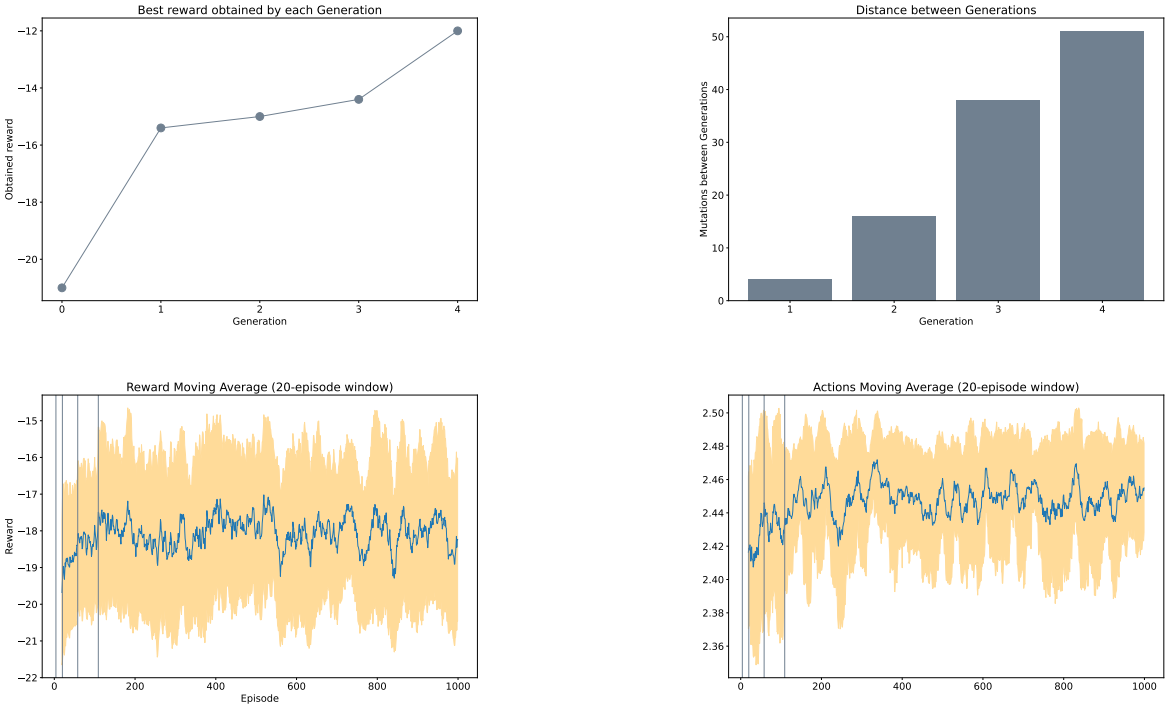


Figure 20: Pong: Noise term of 0.4 and sparse weight updates.

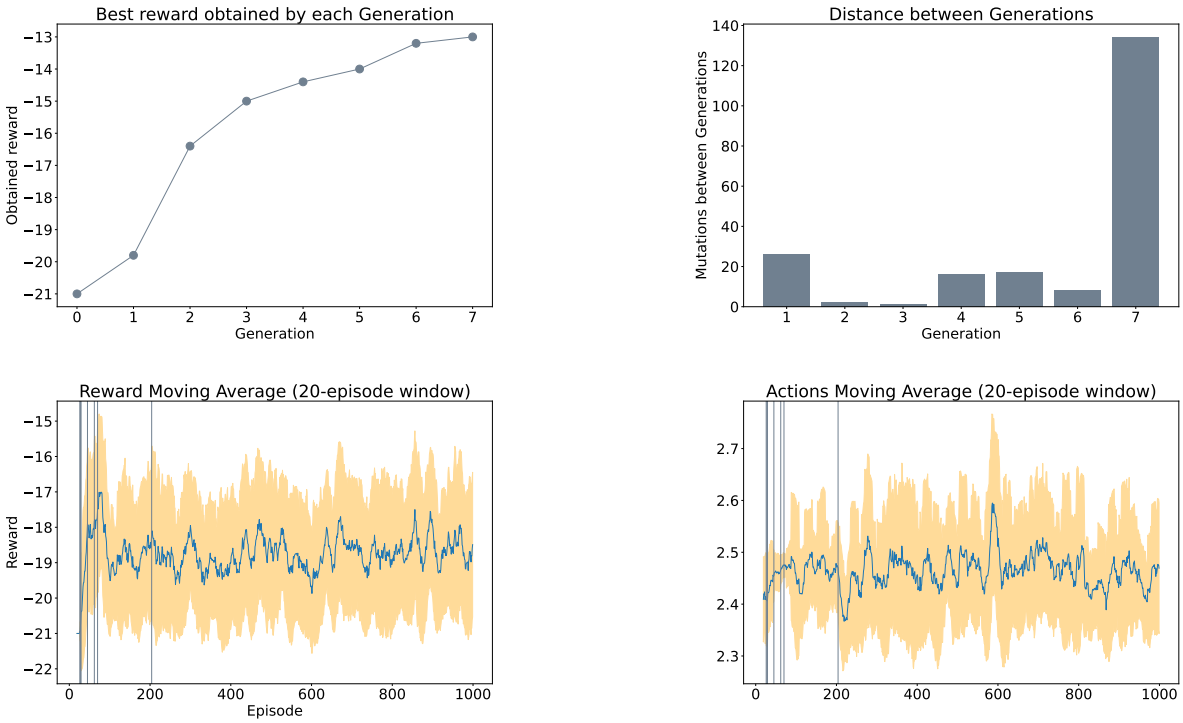


Figure 21: Pong: Noise term of 0.4 and no sparse weight updates.

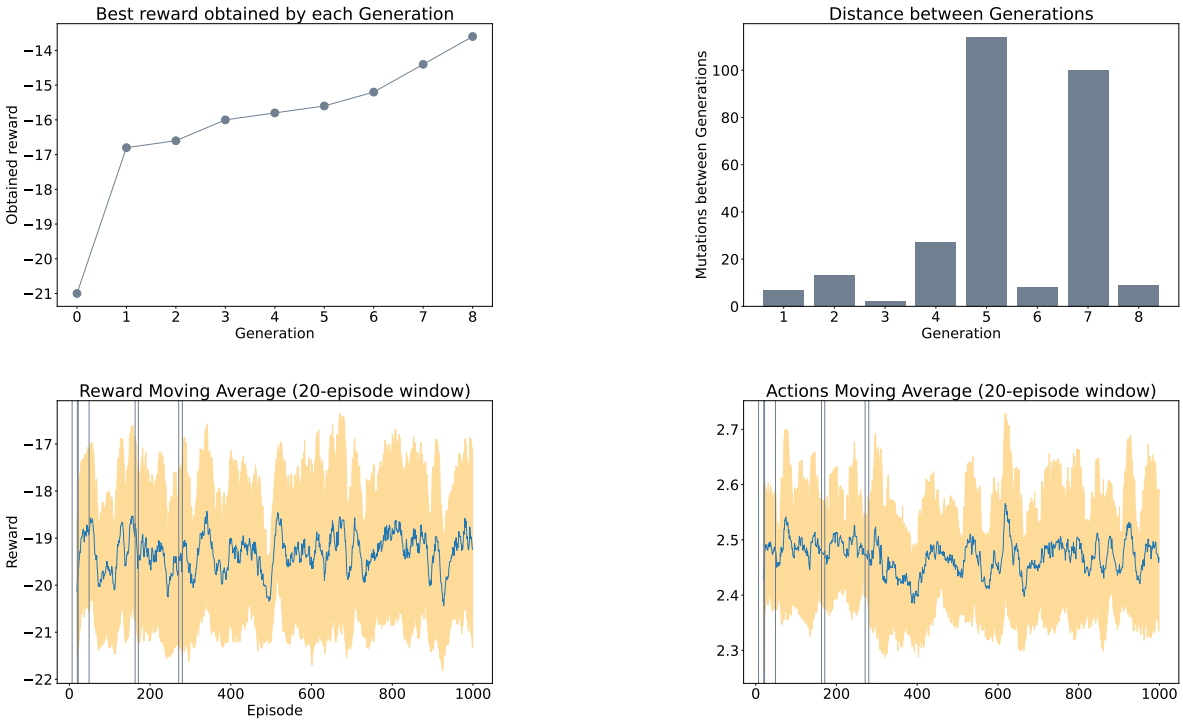


Figure 22: Pong: Noise term of 0.8 and sparse weight updates.

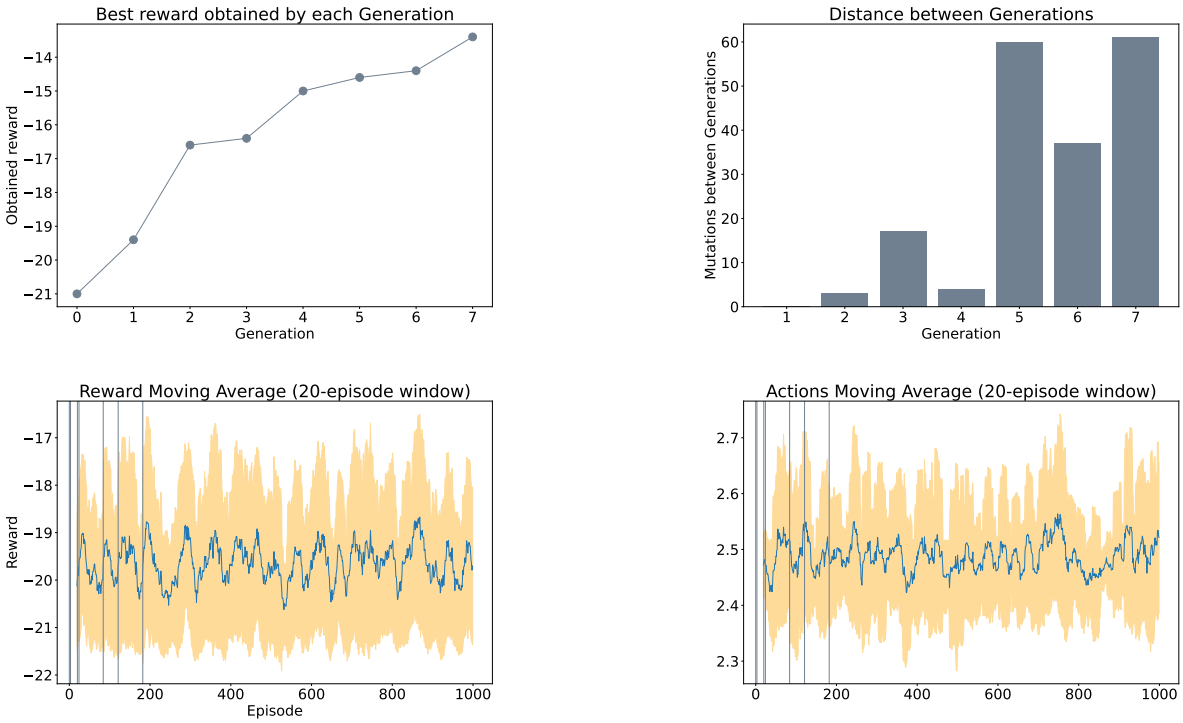


Figure 23: Pong: Noise term of 0.8 and no sparse weight updates.

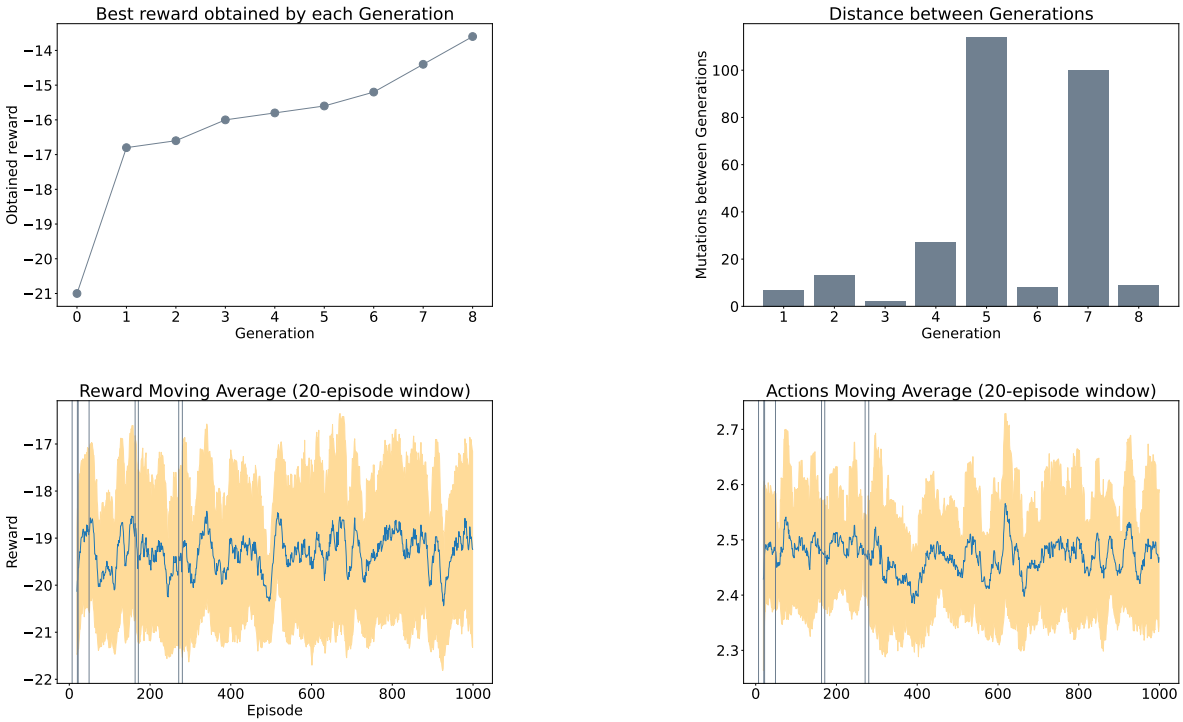


Figure 24: Pong: Noise term of 0.8 and sparse weight updates.

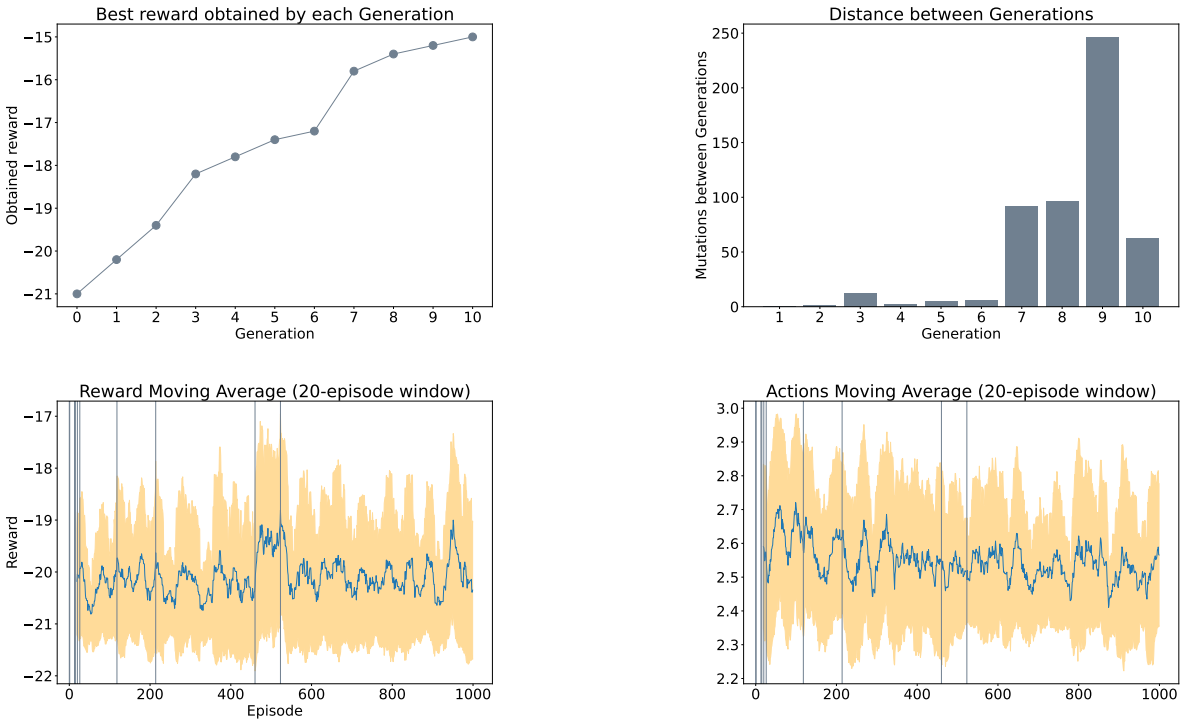


Figure 25: Pong: Noise term of 1.2 and no sparse weight updates.

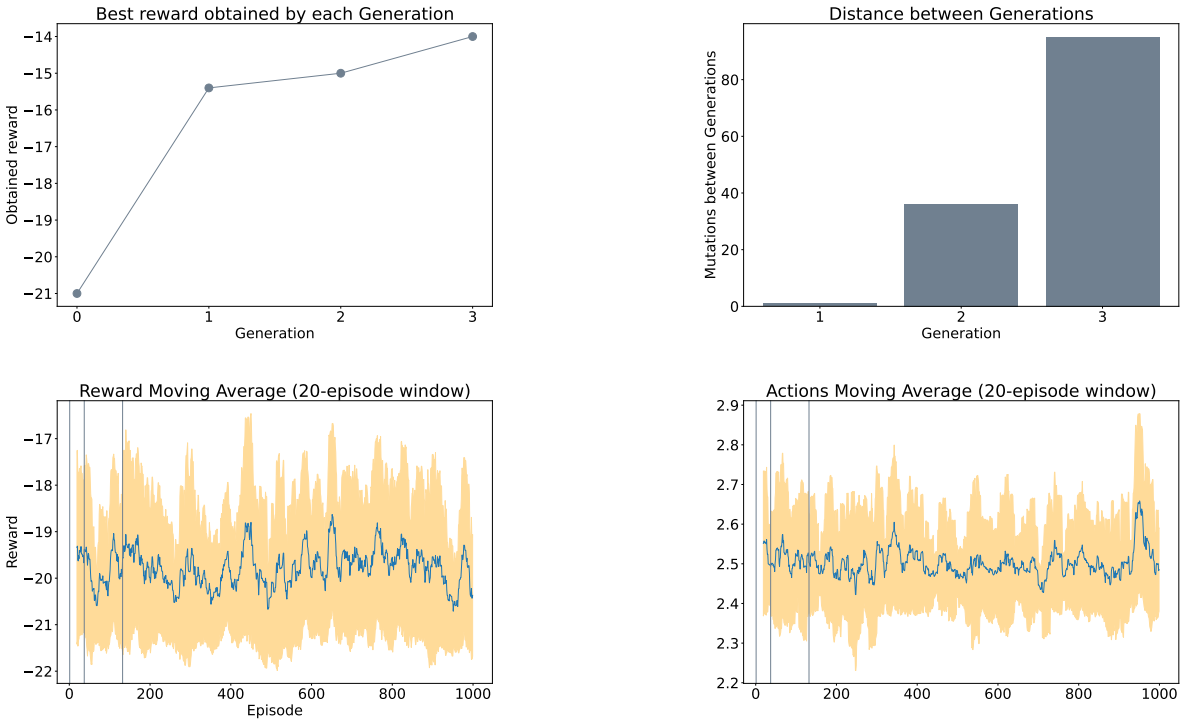


Figure 26: Pong: Noise term of 1.2 and sparse weight updates.

6.5.2 Results Pinball

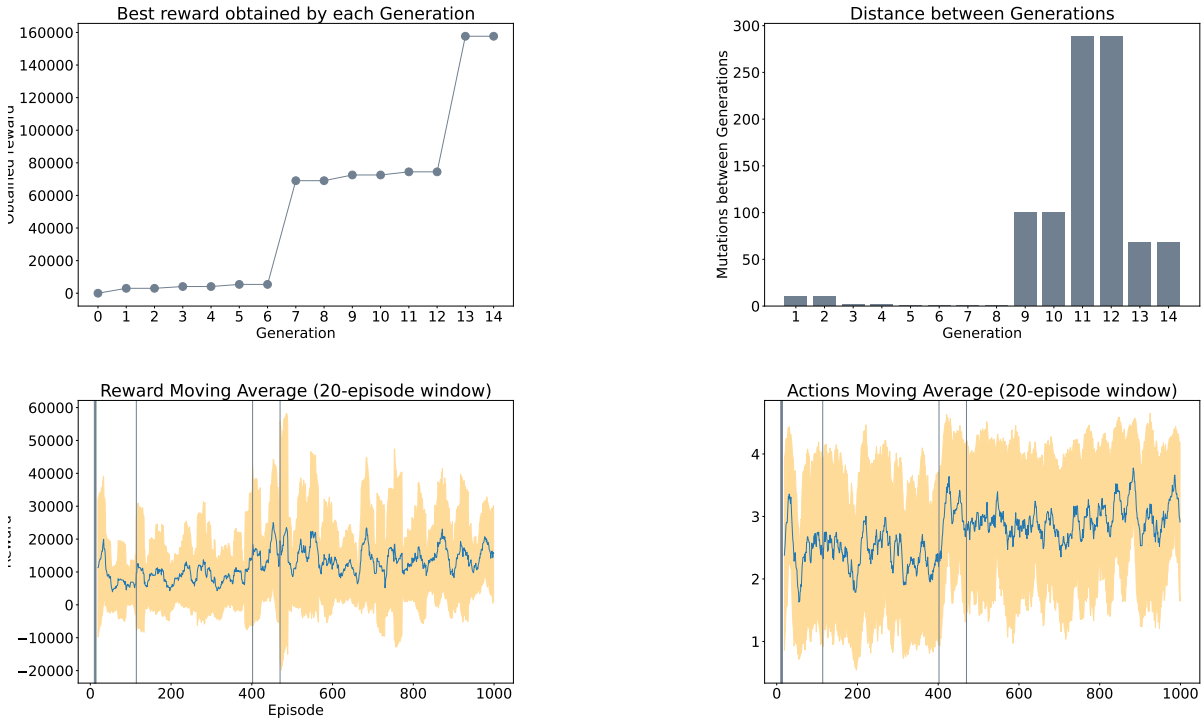


Figure 27: Pinball: Noise term of 0.4 and sparse weight updates.

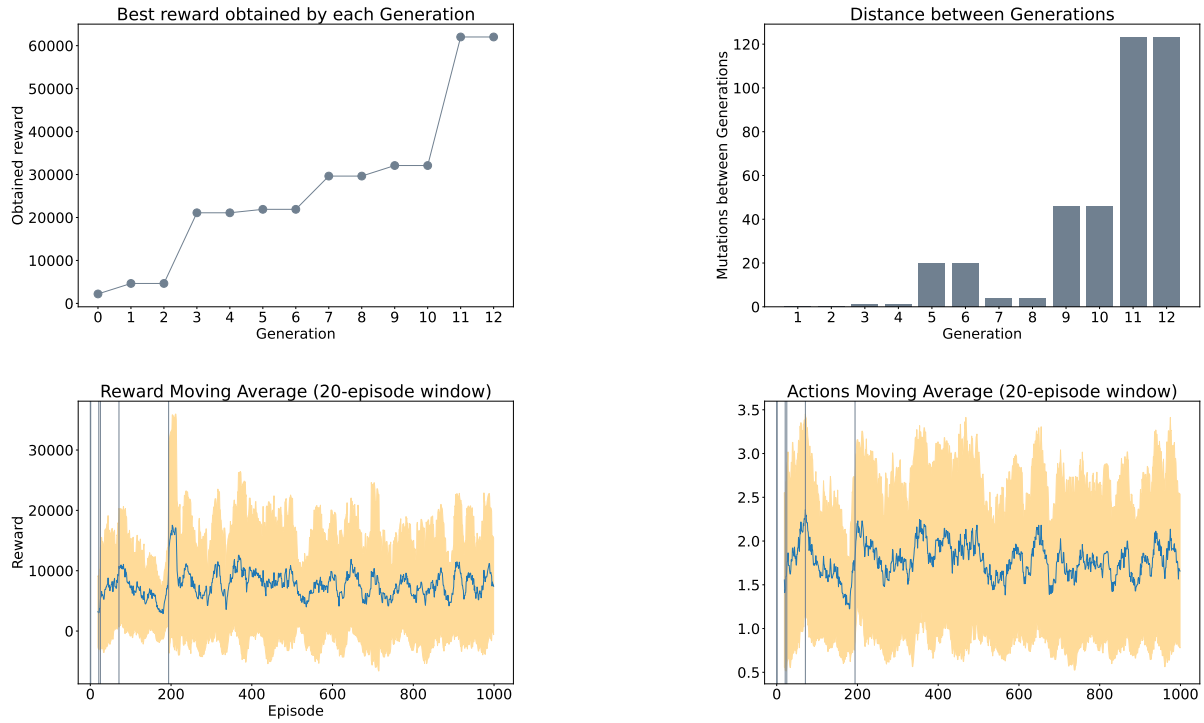


Figure 28: Pinball: Noise term of 0.4 and no sparse weight updates.

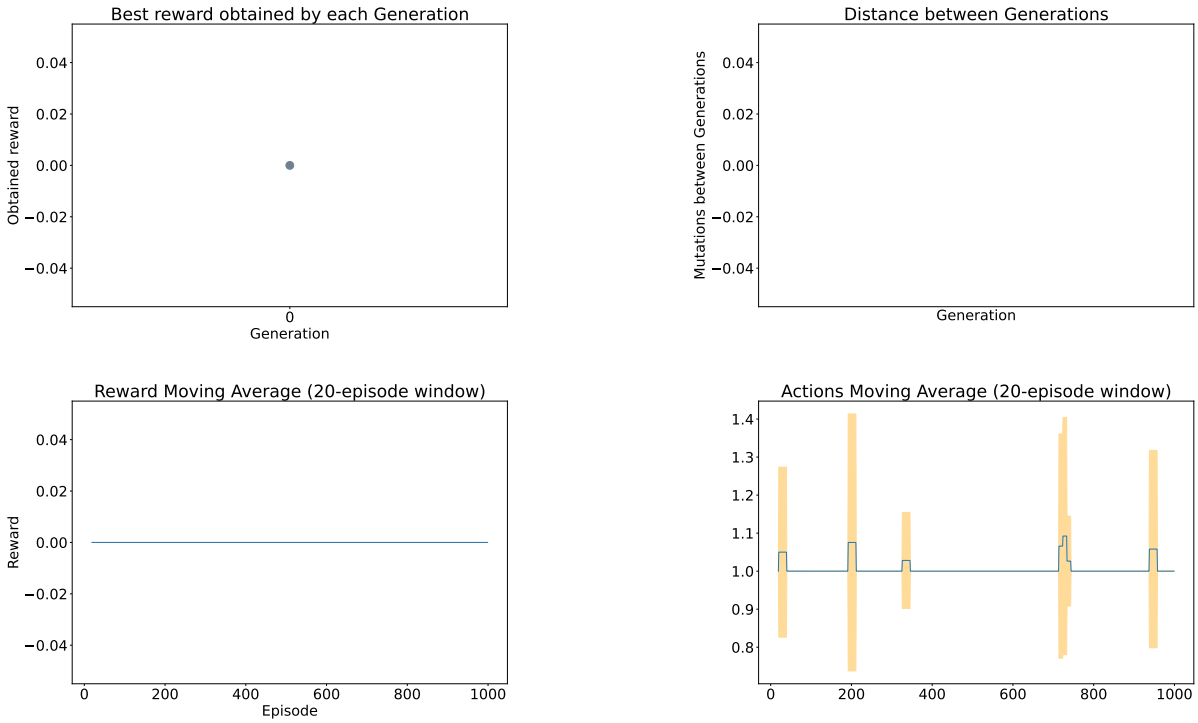


Figure 29: Pinball: Noise term of 0.8 and sparse weight updates.

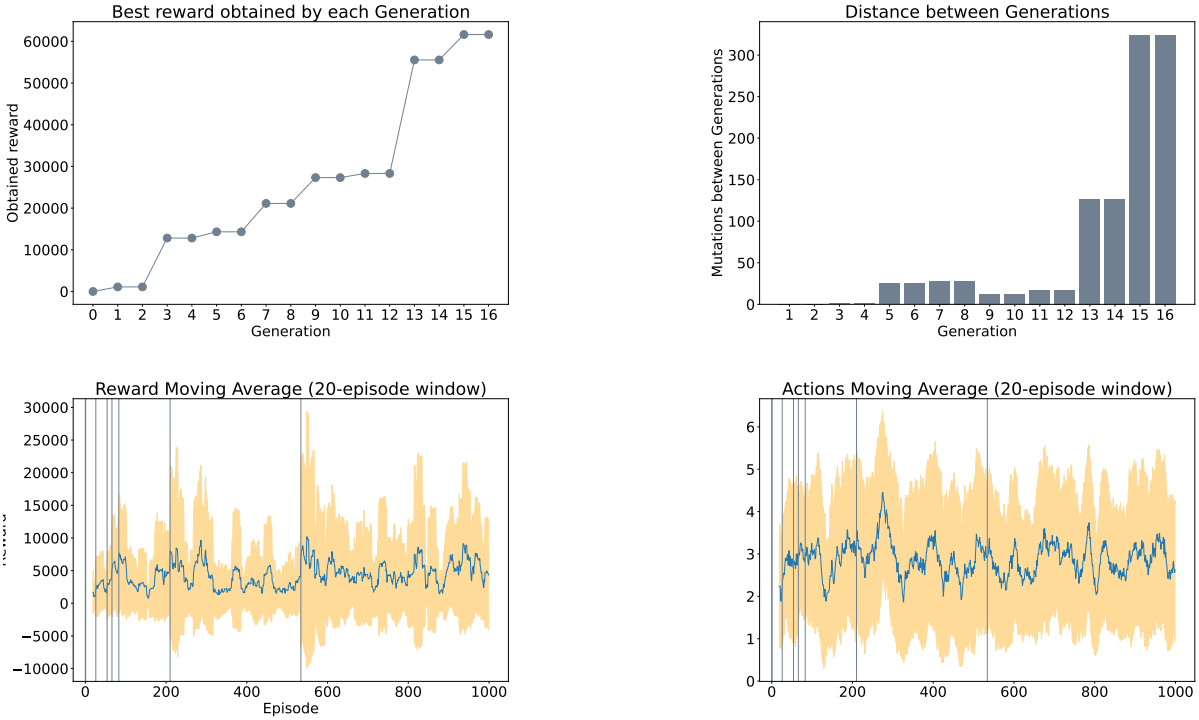


Figure 30: Pinball: Noise term of 0.8 and no sparse weight updates.

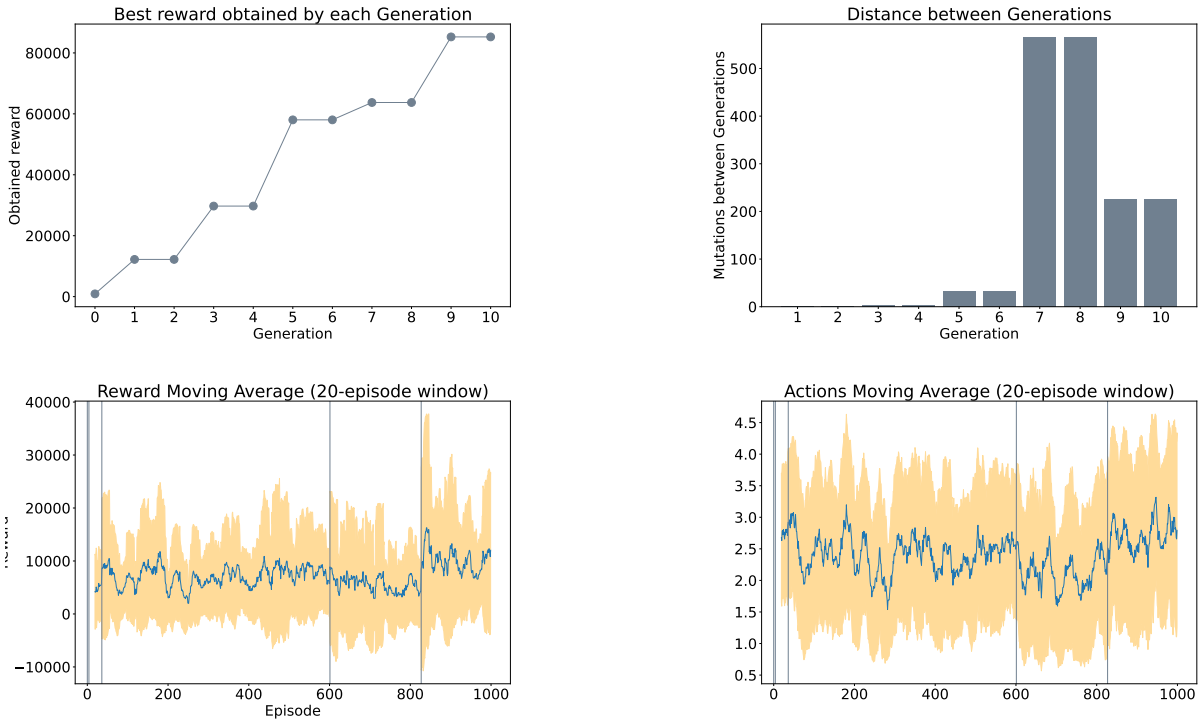


Figure 31: Pinball: Noise term of 1.2 and sparse weight updates.

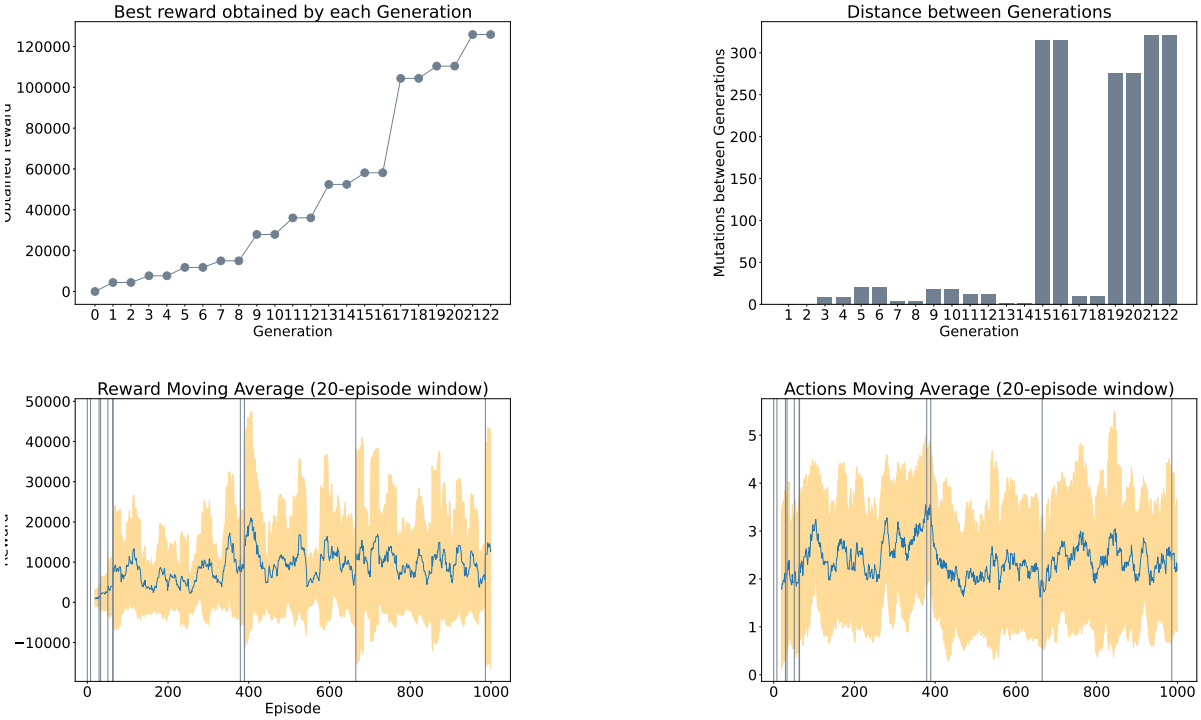


Figure 32: Pinball: Noise term of 1.2 and no sparse weight updates.