Evolutionary Planning in Latent Space

Thor V. A. N. Olesen[1], Dennis T. T. Nguyen[1], Rasmus B. Palm[1], and Sebastian Risi[1,2(✉)]

[1] IT University of Copenhagen, Copenhagen, Denmark
[2] modl.ai, Copenhagen, Denmark

Abstract. Planning is a powerful approach to reinforcement learning with several desirable properties such as sampling efficiency. However, it requires a world model, which is not readily available in many real-life problems. In this paper, we propose to learn a world model that enables *Evolutionary Planning in Latent Space* (EPLS). We use a Variational Auto Encoder (VAE) to learn a compressed latent representation of individual observations and extend a Mixture Density Recurrent Neural Network (MDRNN) to learn a stochastic, multi-modal forward model of the world used for planning. We use the Random Mutation Hill Climbing (RMHC) algorithm to find a sequence of actions that maximize expected reward in this learned model of the world. We demonstrate how to build a world model by bootstrapping it with rollouts from a random policy and iteratively refining it with rollouts from an increasingly accurate planning policy using the learned world model. After few iterations, our planning agents exceed standard model-free reinforcement learning approaches, which demonstrates the viability of our approach. Code to reproduce the experiments is available at https://github.com/two2tee/WorldModelPlanning and videos at https://youtu.be/3M39QgeF27U.

Keywords: World models · Evolutionary planning · Iterative training · Model-based reinforcement learning

1 Introduction

Planning by searching for action sequences that maximize expected reward is a powerful approach to reinforcement learning problems, which has recently lead to breakthroughs in complex domains such Go, Shogi, Chess, and Atari games [21–23]. To plan, the agent needs access to a model of the world which it can use to simulate the outcome of actions, to determine which course of action is best. Planning using a model of the world also allows you to introspect what the agent is planning and why it thinks certain actions are preferable. It even allows you to add constraints or change the objective at runtime.

In games, these world models are readily available and given by the rules of the game. However, for many real-world problems like driving a car, they are not available.

T. V. A. N. Olesen and D. T.T. Nguyen—Contributed equally

P. A. Castillo and J. L. Jiménez Laredo (Eds.): EvoApplications 2021, LNCS 12694, pp. 522–536, 2021.
https://doi.org/10.1007/978-3-030-72699-7_33

Fig. 1. Planed trajectories using evolutionary planning in the latent space of a learned world model.

In problems where a world model is not available, one can instead use model-free reinforcement learning, which learns a policy that directly maps from the environment state to the actions that maximize expected reward. However, these approaches often require many samples from the real environment, which is often expensive to obtain when learning the statistical relationship between states, actions, and rewards. That is especially true if the environment requires complex sequences of actions to observe any rewards.

Alternatively, a *learned* world model is used to find an optimal policy, which is the approach taken in this paper. This is known as model-based reinforcement learning. Learning a model of the world requires fewer samples of the environment than learning a policy directly from the state space since it can use supervised learning methods to predict the environment state transitions, regardless of the reward signal being sparse or not. Several models have been proposed for learning world models [6,8,21].

In this paper, we propose an extension to the Mixture Density Recurrent Neural Network (MDRNN) model [6], which makes it suitable for planning, and demonstrate how to do evolutionary planning in latent space by using the learned world model. See Fig. 1 for examples of planned trajectories and Fig. 2 for an overview of the proposed method.

We further show how to iteratively improve the world model by using the existing world model to do the planning and using the planned policy to sample better rollouts of the environment, which, in turn, are used to train a better world model. This process is bootstrapped by using an initial random policy. Given few iterations, we obtain results that outperform standard model-free approaches, demonstrating the viability of the approach.

2 Related Work

2.1 Planning

In planning, an agent uses a model of the world to predict the consequences of its actions and select an optimal action sequence accordingly. Planning is a powerful technique that has recently lead to breakthroughs in complex domains such as Go, Chess, Shogi, and Atari [21–23].

Monte-Carlo Tree Search (MCTS) is a state-of-the-art planning algorithm for discrete action spaces, which iteratively builds a search tree that explores the most promising paths using a fast, often stochastic, rollout policy [3].

Rolling Horizon Evolutionary Algorithms (RHEA) encode individuals as sequences of actions and uses evolutionary algorithms to search for optimal trajectories. *Rolling Horizon* (RH) refers to how the first action in a plan is executed before the plan is reevaluated and adjusted, looking one step further into the future and slowly expanding the horizon [5,10,11,18]. RHEA naturally handles continuous action spaces. Tong et al. [24] show how to learn a prior for RHEA by training a value and policy network. The value network reduces the required planning horizon by estimating the rewards of future states. The policy network helps initialize the population of planning action trajectories to narrow the search scope to a near-optimal local action policy-subspace. In our approach, we use a randomly initialized set of planning trajectories and improve them iteratively through evolution.

Random Mutation Hill-Climb (RMHC) is a simple and effective type of evolutionary algorithm that repeats the process of randomly selecting a neighbour of a best-so-far solution and accepts the neighbour if it is better than or equal to the current best-so-far solution. This local search method starts with a solution and iteratively tries to improve it by taking random steps or restarting from another region in the policy space.

Planning approaches that rely on imperfect models may plan non-optimal trajectories. The authors of [17] suggest incorporating uncertainty estimation into the forward model to improve the agent. In general, planning under uncertainty has been extensively studied [2,12,16].

2.2 Learning World Models

If a world model is unavailable, it is learned from environment observations. Using such a model to find a policy is generally known as model-based Reinforcement Learning (RL).

World Models [6] introduces a stochastic recurrent world model learned from environment observations under an initially random policy. The model uses a Variational Auto-Encoder (VAE) to encode pixel inputs into a low dimensional latent vector. A recurrent neural network (RNN) is trained to predict sequences of latent states using a Gaussian Mixture Model to capture the uncertain and multi-modal nature of the environment. Notably, the authors do not use this world model for planning, but rather for training a simple single-layer linear policy network.

In *MuZero* [21], the authors do planning with a learned model in video and board games by using tree-based search (MCTS) to enable imitation learning with a policy network.

In *PlaNet* [8], the authors have shown it is possible to do online planning in latent space using an adaptive randomized algorithm on a recurrent state-space model (SSM) with a deterministic and stochastic component and a multi-step prediction objective. *PlaNet* [8] is the approach that is most similar to

the work presented here (i.e., online planning on a learned model). However, it uses a rather complicated dynamics model and planning algorithm. In *Dreamer* [7], the authors use the *PlaNet* world model but no longer do online planning. Instead, their Dreamer agent uses an actor-critic approach to learn behaviors that consider rewards beyond a horizon. Namely, they learn an action model and value model in the latent space of the world model. Thus, their approach is similar to *World Models* [6] where they plan on a learned model by training a policy inside the simulated environment with backpropagation and gradient descent, instead of evolution. The novel part is using a value network to estimate rewards beyond a finite imagination horizon. Also, *World Models* [6] does not show how to do planning on a fully learned model, since the reward signal is not learned in their model. Finally, MuZero [21] relies on extensive training data and access to unrealistic GPU resources, which may not be feasible in practice.

In another related approach, *Neural Game Engine* [1], the authors show how to learn accurate forward models from pixels that can generalize to different size game levels. However, their methods currently only work on grid-based world games. The authors argue it does not work as a drop-in replacement for the kind of world models we need in real-life environments. For this purpose, the authors recommend looking into some of the previously presented methods that learn a latent dynamics model with a 2D state-space model (SSM) like shown in PlaNet and Dreamer that both use a Recurrent State Space Model (RSSM).

3 Approach

We use a model-based RL approach to solve a continuous reinforcement learning control task. We achieve this through online evolutionary planning on a learned model of the environment. Our solution combines a world model [6] with rolling horizon evolutionary planning [18]. See Fig. 2 for an overview.

Similar to the original world model [6], our model uses a visual sensory component (V) to compress the current state into a small latent representation. The memory component (M) is extended to predict the next latent state, the expected reward, and whether the environment terminates. In the original world model, the decision-making component uses a simple learned linear model that maps latent and hidden states directly to actions at each time step. In contrast, EPLS uses a random mutation hill-climbing (RMHC) algorithm as the decision-making component that exploits M to do online planning in latent space.

3.1 Learning the World Model

The **visual component** (V) is implemented as a convolutional variational autoencoder (ConvVAE), which learns an abstract, compressed representation $z_t \in \mathbb{R}^{64}$ of states (i.e., frames) $s_t \in \mathbb{R}^{64\times64\times3}$ using an encoder and decoder as shown in Fig. 3.

The VAE encoder is a neural network that outputs a compressed representation of a state s (i.e., frame) using a deep convolutional neural network (DCNN)

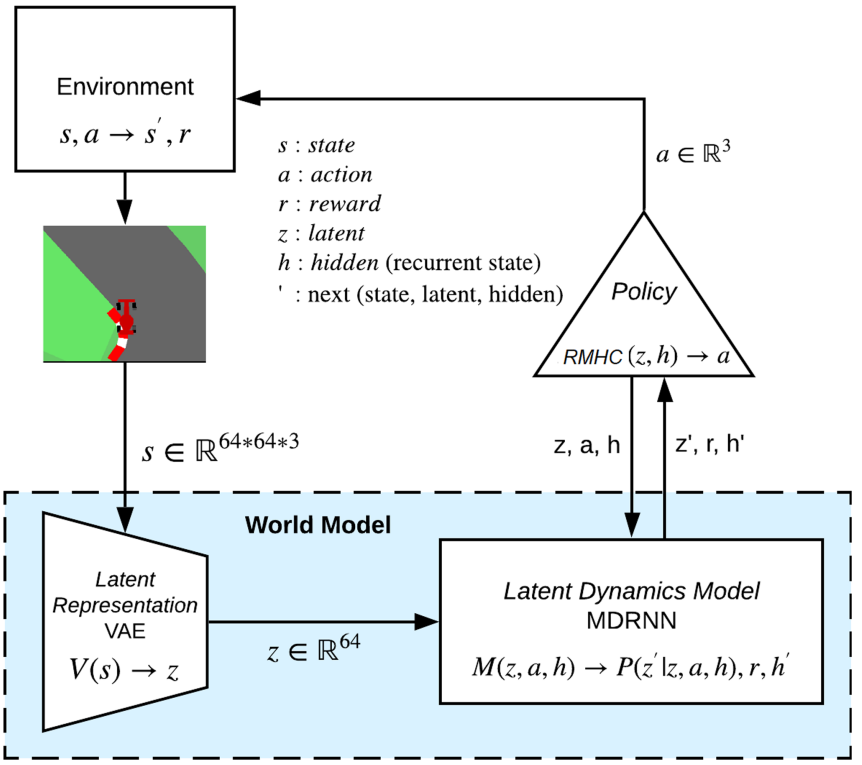

Fig. 2. Evolutionary Planning in Latent Space (EPLS). The raw observation is compressed by V at each time step t to produce a latent vector z_t. RMHC does planning by repeatedly generating, mutating, and evaluating action sequences $a_0, ..., a_T$ in the learned world model, M where T is the horizon. The learned world model, M, receives an action a_t, latent vector z_t and hidden state h_t and predicts the simulated reward r_t, next latent vector z_{t+1}, and next hidden state h_{t+1}. The predicted states are used with the next action as inputs for M to let the agent simulate the trajectory in latent space. The first action of the plan with the highest expected total reward in the simulated environment is executed in the real environment.

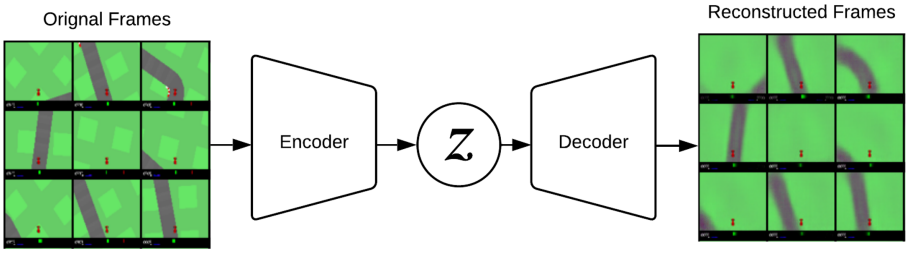

Fig. 3. Flow diagram of a Variational Autoencoder (VAE). The VAE learns to encode frames into latent vectors by minimizing the pixel-wise difference between input frames and reconstructed frames (i.e., L2 or MSE) generated by decoding the latent vectors.

of four stacked convolutional layers and non-linear relu activations to compress the frame and two fully-connected (i.e., dense) layers that encode the convolutional output into low dimensional vectors μ_z and σ_z:

$$encoder : s \in \mathbb{R}^{64\times64\times3} \rightarrow \mu_z \in \mathbb{R}^{64}, \sigma_z \in \mathbb{R}^{64}\,. \tag{1}$$

The means μ_z and standard deviations σ_z are used to sample a latent state z from a multivariate Gaussian with diagonal covariance:

$$z \in \mathbb{R}^{64} \sim \mathcal{N}(z|\mu_z, \sigma_z)\,. \tag{2}$$

The decoder is a neural network that learns to decode and reconstruct the state (i.e., frame) s given the latent state z using a deep CNN of four stacked deconvolution layers:

$$decoder : z \in \mathbb{R}^{64} \rightarrow s' \in \mathbb{R}^{64\times64\times3}\,. \tag{3}$$

Each convolution and deconvolution layer uses a stride of two. Convolutional and deconvolutional layers use relu activations. The output layer maps directly to pixel values between 0 and 1. The VAE is trained with the standard VAE loss [13].

We extend the **memory component** (M) of [6] to output an expected reward r and a binary terminal signal τ to obtain a fully learned world model that supports planning entirely in latent space. M is an LSTM with 512 hidden units, which jointly models the next latent state z_t, reward r_t, and whether or not the environment terminates, τ_t,

$$p(z_t, r_t, \tau_t|h_{t-1}) = p(z_t|h_{t-1})p(r_t|h_{t-1})p(\tau_t|h_{t-1})\,. \tag{4}$$

The LSTM hidden state h_t depends on the previous hidden state h_{t-1}, the current action a_t, and the current latent state z_t such that $h_t = \text{LSTM}(z_t, a_t, h_{t-1})$.

Most complex environments are stochastic and multi-modal so $p(z_t|h_{t-1})$ is approximated as a mixture of Gaussian distribution (MD-RNN). The output of the MDRNN are the parameters π, μ, σ of a parametric Gaussian mixture model where π represents mixture probabilities:

$$p(z_t|h_{t-1}) = \sum_{k=1}^{5} \pi_k \mathcal{N}(z_t|\mu_k, \sigma_k)\,, \tag{5}$$

where π, μ and Σ are linear functions of h_{t-1} and each mixture component is a multivariate Gaussian distribution with diagonal covariance.

We model the reward r using a Gaussian with a fixed variance of one such that

$$p(r_t|h_{t-1}) = \mathcal{N}(r_t|\mu_t^r, 1)\,, \tag{6}$$

where μ_t^r is a linear function of h_{t-1}. Finally we model the terminal state τ using a Bernoulli distribution,

$$p(\tau_t|h_{t-1}) = p^{\tau_t}(1-p)^{1-\tau_t}\,, \tag{7}$$

where $p = \text{sigmoid}(f(h_{t-1}))$ is the sigmoid of a linear function of h_{t-1}.

We train M by minimizing the negative log-likelihood of $p(z_t, r_t, \tau_t | h_{t-1})$ for observed rollouts of the environment,

$$\mathcal{L} = -\log p(z_t, r_t, \tau_t | h_{t-1}) = \text{MSE}(r_t, \hat{r}_t) + \text{BCE}(\tau_t, \hat{\tau}_t) + \text{GMM-NLL}(z_t, \hat{z}_t)\,, \quad (8)$$

where MSE is the mean squared error, BCE is the binary cross-entropy, GMM-NLL is the negative log likelihood of a gaussian mixture model, and $\hat{z}, \hat{r}, \hat{\tau}$ are the observed latent states, rewards and terminals. The learnable weights and biases in the different neural network-based modules are all initialized uniformly by default in PyTorch: $U(-\sqrt{(k)}, \sqrt{(k)})$ with $k = \frac{1}{in\_features}$ where $in\_features$ is the size of each input sample.

3.2 Evolutionary Planning in Latent Space

Once the world model is trained, it can enable planning (Fig. 4). We use Random Mutation Hill Climbing (RMHC), which is a simple evolutionary algorithm. RMHC works by iteratively mutating and evaluating individuals, and letting the elite be the starting point in the next round of mutation. We use RMHC to find a sequence of actions that maximize the expected reward as predicted by the world model. The action sequence length, also known as the horizon, determines how far into the future the agent plans. Finally, shift buffering is used to avoid repeating the entire search process from scratch at every time step [4]. In short, after each planning step, we pop the first action of the action sequence and add a new random action to the end of the action sequence. This modified plan is then the starting point for the next planning step.

4 Experiments

We test our approach on the continuous control `CarRacing-v0` domain [14], built with the Box2D physics engine. At every trial, the agent's driving abilities are evaluated on a randomly generated track (where randomness affects the number of track tiles, its layout, and car starting position). Reaching a high score requires the agent to plan how to make each turn with continuous actions, which makes it a suitable test domain for our evolutionary latent planning approach. The environment yields a reward of -0.1 each time step and a reward of $+1000/N$ for each visited track tile where N is the total number of tiles in the track. While it is not necessarily difficult to drive slowly around a track, reaching a high reward is difficult for many current RL methods [6].

Since the environment gives observations as high dimensional pixel images, these are resized to 64×64 pixels before being used as observations in our world model. Pixels are stored as three floating-point values between 0 and 1 that represent each of the RGB channels. The dimension of our latent space is 64 since this yielded better reconstructions than using 32 as in [6]. Actions contain three numeric components that represent the degree of steering, acceleration, and braking.

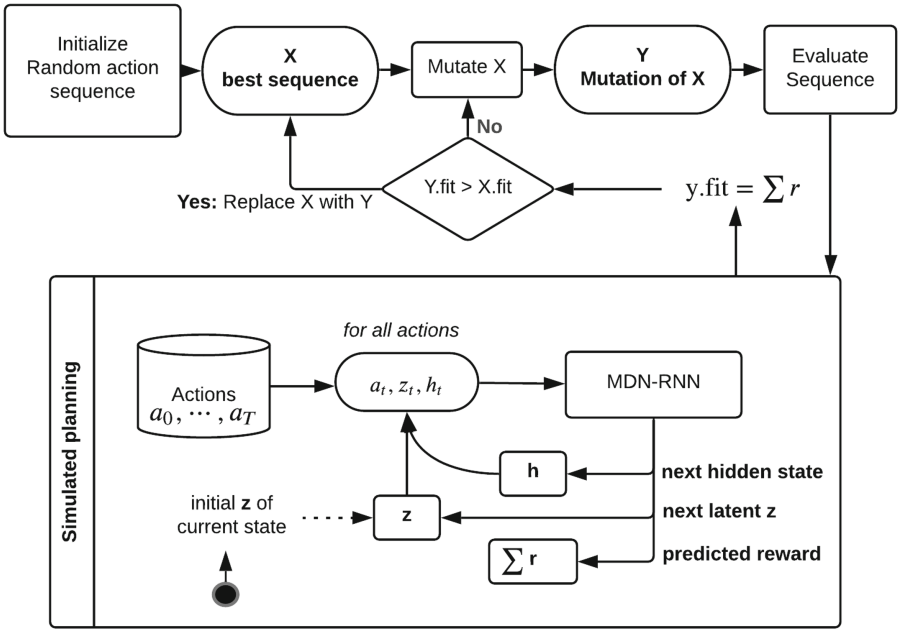

Fig. 4. Planning details. RMHC initializes a random sequence of actions sampled from the environment and mutates it repeatedly across generations. Each plan is evaluated in latent space using the simulated environment where the fitness metric is the total undiscounted expected reward associated with executing the planning trajectory in latent space.

Capturing Rollouts. The MDN-RNN and VAE models are trained in a supervised manner, and rely on access to a representative dataset of environment rollouts for training and testing. Each sample is a rollout in the environment and consists of a sequence of (state, action, reward, terminal) tuples. States, rewards, and terminals are produced by the environment when given an action. Initially, we use a random policy in the environment and record states, rewards, actions, and terminals in T steps. We use $T = 500$ in the non-iterative procedure and $T = 250$ in the iterative procedure. We found that using $T = 250$ was sufficient while speeding up the iterative training procedure.

Non-iterative Training Procedure. The non-iterative training procedure follows the same approach as presented in the original world model work [6]. To train the VAE and MDN-RNN, we first collect a dataset of 10,000 rollouts using a random policy to explore the environment where we record the random action a_t executed and the generated observations. The dataset is used to train the VAE so it can learn an abstract and compressed representation of the environment. The VAE is trained for 50 epochs with a learning rate of $1e-4$ using the *Adam* optimizer.

The MDN-RNN is trained on the 10,000 rollouts where each frame s_t is preprocessed by the VAE into a latent vector z_t at each time step t. The latent vectors and actions a_t are given to the MDN-RNN for it to learn to model the next latent vector $p(z_{t+1}|a_t, z_t, h_t)$ as a mixture of Gaussians and the reward r and the terminal d. The MDN-RNN consists of 512 hidden units and a mixture of 5 Gaussians. We train the MDN-RNN for 60 epochs with a learning rate of $1e-3$ using the *Adam* optimizer. In summary, the full non-iterative training procedure is shown below:

1. Collect 10,000 rollouts with a random policy
2. Train world model using random rollouts.
3. Evaluate the agent on 100 random tracks using the RMHC planning policy.

Iterative Training Procedure

Once the world model is trained non-iteratively, we can use it in conjunction with our planning algorithm (RMHC) to do online planning. However, while the agent may somewhat stay on the road and drive slowly at corners, its performance is limited by our world model trained with a random policy only. Consequently, the dynamics associated with well-behaved driving might be underexplored, and hence our world model may not be able to fully capture this in latent space.

To address this, we used an iterative training procedure as suggested in [6], in which we iteratively collect rollouts using our agent's planning policy and improve our world model (and thus our planning) using the new rollouts. Intuitively, we expect planning with the learned world model to yield a better policy than with a random model. The new rollouts generated during planning are stored in a replay buffer to overcome *catastrophic forgetting* by retaining both old and new rollouts, which allows the MDN-RNN to learn from both past and new experiences. We collect 500 rollouts per iteration. The iterative training procedure is as follows:

1. Train MDN-RNN and VAE non-iteratively to obtain baseline model
2. Collect rollouts using RMHC planning policy and add them to the replay buffer
3. Train the world model using rollouts in replay buffer.
4. Evaluate the agent on 100 random tracks using RMHC planning policy.
5. Go back to (2) and repeat for I iterations or until the task is complete

For both approaches, we found that training the VAE using 10k random rollouts was sufficient in representing different scenarios of the car racing environment across all our experiments. We used RMHC with a horizon of 20, and the action sequence was evolved for ten generations at every time step t with shift buffering.

5 Results

5.1 Non-iterative Training

The MDN-RNN serves as a predictive model of future latent z vector that the VAE may produce and the reward r that the environment is expected to produce. Thus the rollouts used to train the MDN-RNN may affect its predictive ability and how well it represents the real environment during online planning. For this reason, we trained two MDN-RNNs by using the non-iterative training procedure to obtain a random model and an expert model. The random model is trained on 10,000 random rollouts and acts as our baseline model for all iteratively-trained models. The expert model trains on 5,000 random rollouts and 5,000 expert rollouts. The expert rollouts are collected with the pre-trained agent in *World Models* [6]. The random rollouts allow the MDN-RNN to learn the consequences of bad-driving behavior, and the expert rollouts allow it to learn the positive reward signal associated with expert-driving. The expert model is a reference model used for comparison that helps determine how well an agent may perform when the MDN-RNN is trained on a well-representative dataset.

Using the random model, the agent did learn to drive unsteadily around the track and sometimes plan around sharp corners. However, the agent only managed to achieve a mean score of 356.20 ± 176.69 with the highest score of 804. In contrast, using the expert model, the agent managed to obtain a mean score of 765.17 ± 102.18 with the highest score of 900. The expert rollouts improved the MDN-RNN's ability to capture the dynamics of the environment, which significantly improved the agent's performance (Fig. 5).

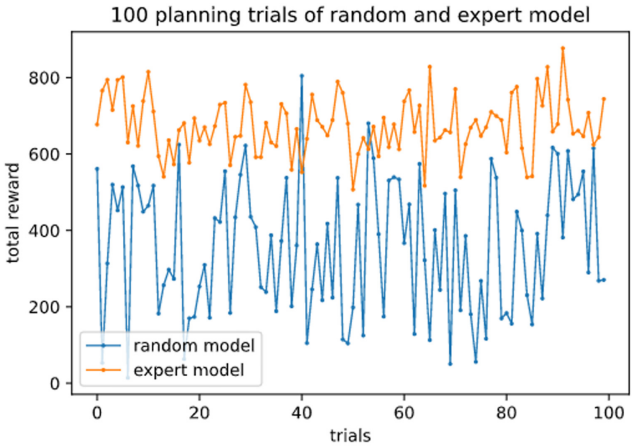

Fig. 5. Total rewards in 100 trials with MDN-RNN trained on 10,000 rollouts using random policy vs. an MDN-RNN trained on 5000 random and 500 expert rollouts. The latter yields a much higher total reward due to the dataset containing a rollouts that exhibit both random and well-behaved driving.

5.2 Iterative Training

Since we cannot rely on access to pre-trained expert rollouts, we have implemented an iterative training procedure that allows the agent to improve its performance over time by learning from its own experiences. Namely, we generate rollouts by online planning with the RMHC evolutionary policy search method, which iteratively improve our world model. We investigate if the random baseline model can improve by using a small number of only 500 rollouts and a sequence length of 250 experiences trained over ten epochs and five iterations. Figure 6 shows the mean total rewards after each of the five iterations.

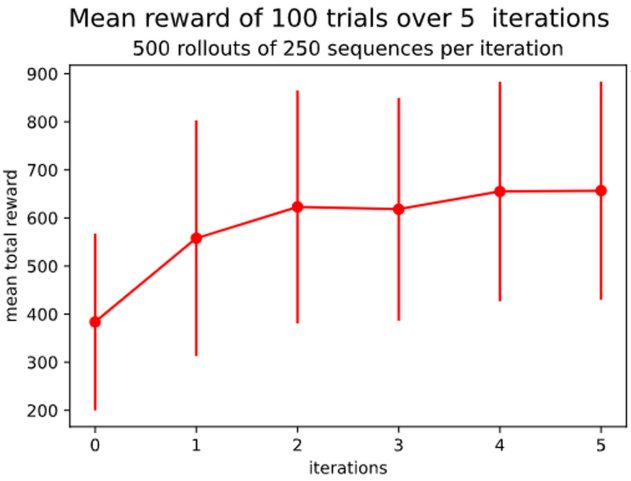

Fig. 6. Mean total rewards and standard deviations over 100 trials across five iterations. The 0th iteration represents the mean total reward before iterative training, but after training on 10,000 rollouts obtained from a random policy. Notice, using 500 rollouts (Experiment A) yields a faster training time and a better result compared to experiment B that uses 10.000 rollouts per iteration.

Already after a single training iteration, the agent managed to get a mean score of 557.87 ± 244.97 and peaked at iteration 5 with a mean score of 656.82 ± 226.67. Despite not beating the expert model, we saw improvements throughout the iterations, and the agent managed to occasionally complete the game by scoring a total reward of 900 during benchmarks. While the first iteration yielded the most significant improvement in total average reward, the following iterations still improved. The great improvement seen in the first iteration might be due to the MDN-RNN learning the dynamics of more well-behaved driving from the agent's planning policy, which ultimately mitigates the errors made by the initially random model.

Investigating Different Planning Horizons and Generations. The benchmarks from iterative training show how refining the world model can affect the agent's planning capabilities. Given the best iterative model found after five training iterations, it is interesting to see how different horizon lengths and max generations affect the agent's ability to plan with the iterative model and the RMHC policy search method.

Both planning parameters are adjusted independently and individually to see how they affect planning. However, we must keep in mind that the horizon length and the maximum number of generations are very likely to be highly correlated. Thus, one should also conduct experiments where both parameters are adjusted together. Figure 7 shows how typical parameter values used for evolutionary planning [15] affect the average total reward obtained by planning with RMHC on a model trained with five iterations across 100 trials.

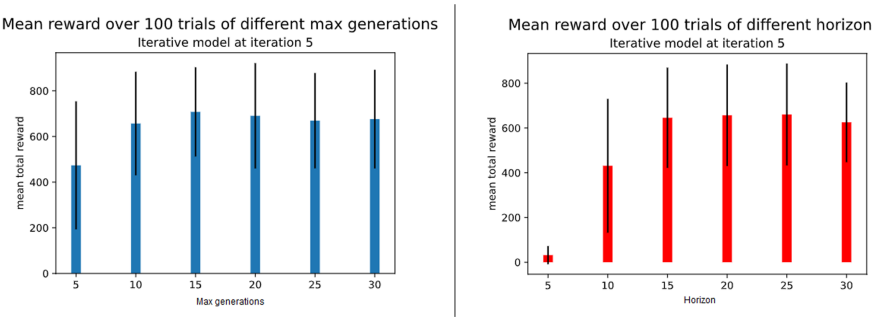

Fig. 7. Left: max planning generations vs. mean rewards with a horizon of length 20. Right: Horizon planning length vs. mean reward with a max generation of 10. While a minimum number of generations and horizon length are necessary for the agent to plan well, increasing these values further does not increase the performance of the agent.

The baseline number of generations is 10. Reducing this to 5 shows a decrease in mean total reward, achieving a score of 473.53 ± 280.18. This reduction is likely due to the agent having less planning time so it is unable to converge to a better trajectory in a local policy subspace. Increasing the number of generations to 15 increases the mean reward to 707.79 ± 195.44, which is not surprising since it gives the agent more planning time. However, increasing the number of generations further did not improve the results. Presumably, this may imply that the agent has converged to a locally optimal trajectory in the simulated environment after being evolved 15 generations.

The results when varying the horizon are shown in Fig. 7, right. The baseline horizon length is 20. Reducing this value to 5 resulted in poor planning and yielded a mean score of 31.91 ± 40.54. Seemingly, a short horizon exploits a more certain near-future but does not bring much information for long-term planning of a trajectory associated with well-behaved driving. Consequently, this kind of short-sighted agent may not act in time before driving into the grass. As we

increase the horizon from 5 to 20, the mean score increases. The agent receives more information about the car's trajectory, which allows it to plan accordingly. However, a horizon beyond 20 does not help the agent, which is likely due to the increased uncertainty caused by planning too far into the future. The further the agent plans ahead, the more uncertain the trajectory becomes, which makes it less relevant to the current situation that the agent must act upon. This is a problem in most model-based RL approaches, which may be addressed by having a separate network predict state values beyond the planning horizon [7].

In conclusion, the iterative training procedure significantly improved our random baseline model and showed improvement after only one iteration (Table 1). Additionally, increasing the maximum number of generations to 15 and a horizon of 20 used in our RMHC policy search approach improved the total average reward obtained across 100 random tracks (Fig. 7). However, increasing the parameter values more than this yields diminishing returns, and a slight decrease in total reward. This may be due to the model's inability to predict far into the future when using a high horizon or the planning trajectory having converged when using a large number of generations. Notice, we do not dynamically adjust the horizon and number of generations during iterative training but keep them fixed during all five iterations. Instead, we compare different combinations of parameters across whole runs of five iterations. To sum up, our results show it is possible to beat traditional model-free RL methods with an evolutionary online planning approach, although we are not yet able to consistently beat or match the learned expert model presented in *World Models* [6].

6 Discussion and Future Work

While the agent reaches a decent score, it does fail occasionally. It usually happens when the agent is unable to correct itself due to loss of friction during turns at sharp corners with high speed. Compared to the expert model that enacts conservative driving-behavior, the current iterative model prefers more risky driving at high speed. Possibly, the expert policy has learned to slow down at corners,

Table 1. CarRacing-v0 approaches with mean scores over 100 trials. Our approaches are shown in bold.

Methods	Mean scores
DQN [19]	343 ± 18
Non-iterative random model	$\mathbf{356 \pm 177}$
A3C (Continuous) [9]	591 ± 45
Iterative model (5 iterations, 15 gen., 20 horizon)	$\mathbf{708 \pm 195}$
Non-iterative - expert model	$\mathbf{765 \pm 102}$
World model [6]	906 ± 21
Deep neuroevolution [20]	903 ± 72

which helps maximize the reward. On the other hand, our planning agent does not seem to have explored sufficient rollouts of this kind to make the MD-RNN learn to associate higher rewards with slower driving when approaching corners.

Another issue occurs when the agent approaches the right corners. In many cases, the agent can complete right corners though there are times where the agent does not know whether to turn or not. In these scenarios, the agent usually brakes or slows down while trying to navigate the race track in a sensible direction. This phenomenon is likely due to the right turns being underrepresented in the generated tracks that are biased towards containing mainly left turns. Consequently, the MDN-RNN is unable to represent right turns in the simulated environment compared to other frequently occurring segments of the track. Arguably, both issues are resolved by running more iterative training iterations. However, it also depends on how often the issues arise in the generated rollouts. Interestingly, the issues occurred more often in the random model compared to the iterative model, which indicates that the iterative training procedure can help improve the world model.

Acknowledgments. We would like to thank Mathias Kristian Kyndlo Löwe for helping us with computational infrastructure. A special thanks go to Corentin Tallec and his team for providing the PyTorch open-source implementation of *World Models* [6]. We also thank Simon Lucas, Chris Bamford, and Alexander Dockhorn for helpful suggestions. This project was supported by a Sapere Aude: DFF-Starting Grant (9063-00046B) and by the Danish Ministry of Education and Science, Digital Pilot Hub, and Skylab Digital.

References

1. Bamford, C., Lucas, S.: Neural game engine: Accurate learning of generalizable forward models from pixels. arXiv preprint arXiv:2003.10520 (2020)
2. Blythe, J.: An overview of planning under uncertainty. In: Wooldridge, M.J., Veloso, M. (eds.) Artificial Intelligence Today. LNCS (LNAI), vol. 1600, pp. 85–110. Springer, Heidelberg (1999). https://doi.org/10.1007/3-540-48317-9_4
3. Browne, C.B., et al.: A survey of monte carlo tree search methods. IEEE Trans. Comput. Intell. AI Games **4**(1), 1–43 (2012)
4. Gaina, R.D., Devlin, S., Lucas, S.M., Perez-Liebana, D.: Rolling horizon evolutionary algorithms for general video game playing. arXiv preprint arXiv:2003.12331 (2020)
5. Gaina, R.D., Lucas, S.M., Pérez-Liébana, D.: Population seeding techniques for rolling horizon evolution in general video game playing. In: 2017 IEEE Congress on Evolutionary Computation (CEC), pp. 1956–1963. IEEE (2017)
6. Ha, D., Schmidhuber, J.: World models. arXiv preprint arXiv:1803.10122 (2018)
7. Hafner, D., Lillicrap, T., Ba, J., Norouzi, M.: Dream to control: Learning behaviors by latent imagination. arXiv preprint arXiv:1912.01603 (2019)
8. Hafner, D., et al.: Learning latent dynamics for planning from pixels. In: International Conference on Machine Learning, pp. 2555–2565. PMLR (2019)
9. Jang, S., Min, J., Lee, C.: Reinforcement car racing with A3C (2017)

10. Justesen, N., Mahlmann, T., Risi, S., Togelius, J.: Playing multiaction adversarial games: online evolutionary planning versus tree search. IEEE Trans. Games **10**(3), 281–291 (2017)
11. Justesen, N., Risi, S.: Continual online evolutionary planning for in-game build order adaptation in StarCraft. In: Proceedings of the Genetic and Evolutionary Computation Conference, pp. 187–194 (2017)
12. Kahn, G., Villaflor, A., Pong, V., Abbeel, P., Levine, S.: Uncertainty-aware reinforcement learning for collision avoidance. arXiv preprint arXiv:1702.01182 (2017)
13. Kingma, D.P., Welling, M.: Auto-encoding variational bayes. arXiv preprint arXiv:1312.6114 (2013)
14. Klimov, O.: Carracing-v0 (2016). https://gym.openai.com/envs/CarRacing-v0/
15. Lucas, S.M., et al.: Efficient evolutionary methods for game agent optimisation: Model-based is best. arXiv preprint arXiv:1901.00723 (2019)
16. Michie, D.: Game-playing and game-learning automata. In: Advances in Programming and Non-numerical Computation, pp. 183–200. Elsevier (1966)
17. Ovalle, A., Lucas, S.M.: Bootstrapped model learning and error correction for planning with uncertainty in model-based RL. arXiv preprint arXiv:2004.07155 (2020)
18. Perez, D., Samothrakis, S., Lucas, S., Rohlfshagen, P.: Rolling horizon evolution versus tree search for navigation in single-player real-time games. In: Proceedings of the 15th Annual Conference on Genetic and Evolutionary Computation, pp. 351–358 (2013)
19. Prieur, L.: Deep-q learning for Box2D racecar RL problem (2017). https://goo.gl/VpDqSw
20. Risi, S., Stanley, K.O.: Deep neuroevolution of recurrent and discrete world models. In: Proceedings of the Genetic and Evolutionary Computation Conference, pp. 456–462 (2019)
21. Schrittwieser, J., et al.: Mastering atari, go, chess and shogi by planning with a learned model. arXiv preprint arXiv:1911.08265 (2019)
22. Silver, D., et al.: Mastering the game of go with deep neural networks and tree search. Nature **529**(7587), 484–489 (2016)
23. Silver, D., et al.: Mastering chess and shogi by self-play with a general reinforcement learning algorithm. arXiv preprint arXiv:1712.01815 (2017)
24. Tong, X., Liu, W., Li, B.: Enhancing rolling horizon evolution with policy and value networks. In: 2019 IEEE Conference on Games (CoG), pp. 1–8. IEEE (2019)