Department of Computer Science
Selma Lagerlöfs Vej 300
9220 Aalborg East, Denmark
www.cs.aau.dk

AALBORG
UNIVERSITY

STUDENT REPORT

Title

Curiosity-driven Planning with Reinforcement Learning

Project Type

Master's Thesis

Project Period

Spring 2023

Project Group

cs-23-mi-10-04

Author

Kim Nguyen

Supervisors

Christian Schilling
Kim G. Larsen

Number of pages: 12
Date of completion: June 9, 2023

Summary

Reinforcement Learning (RL) approaches are most often clueless of what to do in environments of sparse extrinsic rewards. Nonetheless, humans and animals are able to learn under similar conditions. This is primarily due to curiosity, which is fundamental in how we, humans, and animals learn. In this work, we investigate the potential of how curiosity can enhance a model-based RL agent's learning in visual environments of only sparse extrinsic rewards. We introduce a novel model-based curiosity-driven RL agent that uses a Monte Carlo Tree Search (MCTS) algorithm.

As we are concerned with visual inputs, we train a Convolutional Neural Network Variational Autoencoder (CNN-VAE) to learn a mapping of the high-dimensional visual inputs into abstract low-dimensional latent representations, where the latent representations still preserve enough information to be reconstructed – to an adequate degree – back to their initial high-dimensional representation. Our proposed agent is given these compact latent representations as input.

For generating curiosity, we use the concept of Random Network Distillation (RND) to derive an intrinsic reward that decays with familiarity. With RND, we use the modelling errors of a predictor network as measures of novelty, whose aim is to predict the output of a fixed and randomly initialised target network that is given the current latent state as input. This approach ensures that the agent is not curious about stochastic elements of the environment.

The MCTS algorithm relies on a learned policy network and value network to sample and evaluate actions. This effectively simplifies the MCTS procedure, as it entirely removes the need of doing Monte Carlo rollouts. We encourage curiosity-driven planning in the MCTS by using the intrinsic RND rewards as episodic exploration bonuses, such that the agent is incentivised to explore novel states, that has not yet been visited in the current episode, and discouraged to revisit familiar states, that it has already been to in the current episode.

We demonstrate that giving an MCTS agent curiosity enhances its learning, as the curiosity MCTS agent is able to solve a visual environment of sparse rewards, called Frozen Lake – a seemingly simple 8×8 grid environment with RGB pixel inputs and sparse rewards – that is, surprisingly, otherwise unsolvable by a vanilla MCTS agent and a model-free vanilla Proximal Policy Optimisation (PPO) agent.

Finally, we examine the prospect of building a visual and temporal abstract world model, that can be used as a MCTS simulation environment, such that our curiosity MCTS agent does not need direct access to the environment's dynamics. We build a predictive world model for an environment of complex dynamics using a CNN-VAE and a Mixture Density Network Recurrent Neural Network (MDN-RNN). We, however, find the world model inapplicable as an MCTS simulation environment in its current state, due to either a too simple experience gathering method, a limited modelling capacity, or both.

Nomenclature

BCE	Binary Cross Entropy
CNN	Convolutional Neural Network
ELU	Exponential Linear Unit
FC	Fully Connected
FNN	Feedforward Neural Network
GMM	Gaussian Mixture Model
GRU	Gated Recurrent Unit
KL	Kullback-Leibler
MCTS	Monte Carlo Tree Search
MDN	Mixture Density Network
MR	Montezuma's Revenge
MSE	Mean Squared Error
PPO	Proximal Policy Optimisation
PUCT	Probabilistic Upper Confidence Threshold
ReLU	Rectified Linear Unit
RGB	Red, Green and Blue
RL	Reinforcement Learning
RND	Random Network Distillation
RNN	Recurrent Neural Network
VAE	Variational Autoencoder

Mathematical Notation

x	Scalar (integer or real)
$\mathbf{x}$	Vector
$\mathbf{X}$	Matrix
$\mathcal{X}$	Set

Curiosity-driven Planning with Reinforcement Learning

Kim Nguyen

Abstract

Reinforcement Learning (RL) approaches are most often clueless of what to do in environments of only sparse extrinsic rewards. Nonetheless, humans and animals are able to learn under similar conditions due to curiosity, as it gives us an intrinsic drive to explore what is novel. In this work, we investigate the potential of how curiosity can enhance a model-based RL agent's learning and encourage exploration in a visual environment of only sparse extrinsic rewards. We introduce a novel model-based curiosity-driven RL agent, that learns and uses a compact latent representation of the visual environment as input, employs the concept of Random Network Distillation (RND) to generate episodic intrinsic rewards and encourage curiosity-driven planning in a Monte Carlo Tree Search (MCTS). We demonstrate that curiosity enhances the learning of a model-based agent, as our proposed agent is able to solve a visual environment of sparse rewards, that is otherwise unsolvable by a model-free and model-based agent without curiosity. Finally, we examine the prospect of building a world model that can be used as an MCTS simulation environment.

Keywords: Model-based Reinforcement Learning, Monte Carlo Tree Search, Exploration, Curiosity Learning, Random Network Distillation, World Models

1. Introduction

The reward function plays an integral role in the success of any form of Reinforcement Learning (RL) application. However, designing an appropriate reward function is not a straightforward matter. RL tends only to learn something of use with dense reward functions. However, designing these in practice are most often time-consuming and a cumbersome matter of trial-and-error, as the RL agents are in the habit of discovering ways to get rewarded in undesired ways. On the other hand, if the RL agents are left with the opposite – a sparse reward function – they are most often clueless of what to do [1].

To look for ways to solve this issue, we can take inspiration from how humans and animals learn. Despite having only sparse extrinsic rewards available to us, we are able to learn due to an intrinsic motivation – curiosity – that drives us to explore what is novel. Curiosity is fundamental in how we, humans, and animals learn. We can generally define curiosity as an intrinsic motivation to seek out novel stimuli, which diminishes as we become more familiar with it [2]. In light of this, to achieve such a behaviour, a notion of what is novel is needed, which then can be used to derive an intrinsic reward that decays with familiarity. To this extent, there have been several proposed ways of doing so.

Some works, such as Bellemare et al. [3] use a count-based exploration approach, where visit counts are used to encourage exploration of novel states. Other works, such as Schimidhuber [4] and Stadie et al. [5], involve using prediction models, and use the prediction modelling errors as measures of novelty to which intrinsic rewards can be derived from. A key challenge, however, in using prediction errors as intrinsic rewards, is that high prediction errors can also be a result of stochasticity in the environment. A local source of entropy in the environment, such as a TV with white noise would be an irresistible attraction to the agent, as the agent would simply get rewarded for endlessly watching the noisy TV.

Pathak et al. [6] propose to solve this by first training an inverse dynamics model, that predicts the action a_t given the current state s_t and the next state s_{t+1}. In doing so, the inverse dynamics model learns a latent space $\mathcal{Z}$ that only contains environmental features that are of relevance to the agent, i.e. things that are influenced by the agent's actions, or things that the agent cannot control, but can be influenced by. Subsequently, the authors train a forward latent dynamics model, that predicts the next latent state $\mathbf{z}_{t+1}$ given the current state $\mathbf{z}$ and action a_t. As a result, the agent is not curious about things that are inconsequential to it.

Burda et al. [7] propose a simpler approach to address the noisy TV problem. Their approach, called Random Network Distillation (RND), involve using two neural networks of similar anatomy: A fixed randomly initialised target network F, and a predictor network $\hat{F}$, whose aim is to predict the output of the target network given the current state as input. As a result, the answer to the prediction problem is now one of a deterministic nature. Furthermore, as both networks are of identical composition, the prediction problem is also ensured to be within the modelling capacity of the predictor network.

One drawback of both mentioned solutions, however, is that they are not robust to stochastic noise generated by agent itself. E.g. if the agent had a TV remote and its actions would switch the TV channels, it would waist all its time perpetually sapping between channels [8]. To resolve this problem, Pathak et al. [9] propose training an ensemble of forward dynamics models, instead of training only a single forward dynamics model, where the ensemble of models each have different initialisation and are trained on different randomly sampled subsets of data. As such,

Department of Computer Science, Aalborg University, Denmark

their predictions on novel states will differ from one another. In other words, they will disagree with each other. This disagreement amongst the ensemble of models is then used as the intrinsic reward. As more data is gathered and trained on, the ensembles' predictions will converge, their disagreement will decrease, and eventually the ensemble will come to agreement with each other. This intrinsic reward makes the agent robust to stochastic noise from both the environment and the agent itself.

A limitation of the works mentioned so far is that their exploration bonus is inter-episodic. Meaning, once the novelty of a state wears off, the agent is never encouraged to visit it again, despite the fact that the state might lead to undiscovered treasures. Badia et al. [10] propose to solve this by having an intrinsic reward that has both an episodic and inter-episodic exploration bonus, such that the agent is encouraged to revisit familiar states – whose potential might not be fully explored – across different episodes, but not in the same episode. For generating the episodic exploration bonus, they take inspiration from Pathak et al. [6], and first train an inverse dynamics model in order to learn a latent space that only contains relevant aspects of the environment. Subsequently, they store each visited state in their latent state representation in an episodic memory buffer. They then derive an episodic exploration bonus by measuring the similarity between the current state and the contents of the memory buffer. As the life-long inter-episodic exploration bonus, they use the RND reward of Burda et al. [7]. By having these two intrinsic rewards, they incentivise the agent to never give up in exploring.

Currently, most works – including those mentioned so far – use curiosity learning to enhance the learning of model-free RL approaches, and have done so with great success. On the other hand, the potential of curiosity learning with model-based RL approaches remains rather uncharted grounds. With model-based RL, the agent has a model of the environment, which it can use to predict how the environment will respond to its actions. This gives the agent planning capabilities, as the agent is able to simulate the outcome of different courses of action and use this to act accordingly [1]. Model-based RL approaches have played a central role in recent remarkable breakthroughs in the field of artificial intelligence. Two striking examples are: DeepMind's AlphaStar by Vinyals et al. [11] that in 2019 reached a rank above 99.8% of human players in the real-time strategy game StarCraft II, and DeepMind's MuZero by Schrittwieser et al. [12] that in 2020 achieved state-of-the-art performance in Go, Chess, Shogi and 57 Atari games. Both these accomplishments have been achieved using a Monte Carlo Tree Search (MCTS) [13] as the planning algorithm.

In this work we introduce, curiosity MCTS, and explore the prospect of curiosity learning with model-based RL in visual environments of only sparse extrinsic rewards. We use an MCTS algorithm, inspired by some aspects of MuZero [12], that relies on a policy network and a value network to sample and evaluate actions. This simplifies the whole search procedure, as it entirely removes the need of doing Monte Carlo rollouts. Besides this, we also use the Probabilistic Upper Confidence Tree (PUCT) bound and MCTS policy of MuZero [12].

For generating curiosity we use an intrinsic RND reward. The RND reward is used due to its simple nature. We encourage curiosity-driven planning by using it as an episodic exploration bonus, that is updated after each step the MCTS agent makes in the actual environment. As we are concerned with visual inputs, we train a Convolutional Neural Network Variational Autoencoder [14] (CNN-VAE), and use its encoder to map the high-dimensional visual input $\mathbf{X}$ into abstract low-dimensional latent representations $\mathbf{z}$. These learned latent representations are then used as input to the MCTS agent, and in computing the intrinsic RND rewards. Our results show that giving an MCTS agent curiosity enhances its learning, as the curiosity MCTS agent is able to solve a modified version of Frozen Lake [15] – a seemingly simple 8×8 grid environment with RGB pixel inputs and sparse rewards, where the agent's goal is to cross a frozen lake without falling into any holes – that is, surprisingly, otherwise unsolvable by a vanilla MCTS agent and a model-free vanilla Proximal Policy Optimisation (PPO) agent [16].

2. Curiosity MCTS

The curiosity MCTS consists of three parts:

- **CNN-VAE**: First, we map the high-dimensional visual input $\mathbf{X}$ into a learned abstract and compact latent state vector $\mathbf{z}$ using a trained CNN-VAE.

- **RND-network**: Next, we use an RND network R^I that takes $\mathbf{z}$ as input and generates intrinsic rewards r^I to be used for curiosity-driven planning.

- **MCTS agent**: Finally, we use an MCTS agent, that simulates the outcome of different courses of action and uses this information to act accordingly. During the agent's search, it is guided by a policy network $\hat{\pi}$, a value network $\hat{V}$ and the RND network R^I, where the policy network and value network also take $\mathbf{z}$ as input. When the MCTS agent has finished its search, it takes an action a in the environment using its own MCTS policy π.

A design overview of the three parts is shown in Fig. 1.

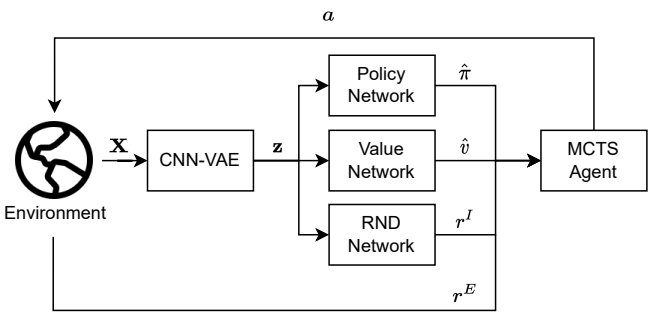

Fig. 1: Design overview of curiosity MCTS.

Ensuingly, we elaborate upon each part in further details.

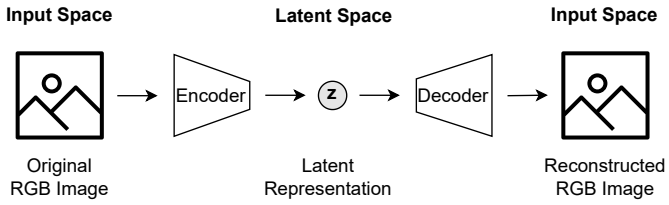

Fig. 2: Overview of the CNN-VAE.

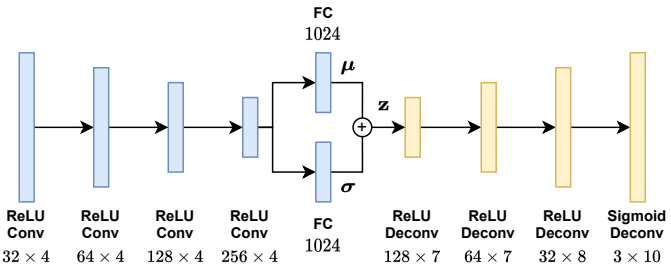

Fig. 3: The CNN-VAE architecture. Encoder layers are blue and decoder layers are yellow. Each convolutional and deconvolutional layer is given by *output channels* × *kernel size*. All convolutional and deconvolutional layers use a stride of 2. FC are Fully-Connected Layers. All layers use a ReLU activation, except the final deconvolutional layer which uses a sigmoid activation to ensure an output range of [0,1]. The vectors $\boldsymbol{\mu}$, $\boldsymbol{\sigma}$ and $\mathbf{z}$ are of size 32×1. The latent vector $\mathbf{z}$ is computed using Eq. (4).

2.1. Learning latent state representations with CNN-VAE

With a CNN-VAE we learn to map items from a high-dimensional input space $\mathcal{X}$ into a learned latent space[1] $\mathcal{Z}$, where the latent representations still preserve enough information to be reconstructed – to an adequate degree – back to their initial high-dimensional representation. A CNN-VAE is a generative model[2] that consists of two components. First, we have an encoder that maps the visual input to a probability distribution in a learned latent space. Subsequently, we can sample from this distribution and use a decoder – that does the opposite of the encoder – to reconstruct the latent representation back to its initial form. An overview of a CNN-VAE is illustrated in Fig. 2.

In this work, our CNN-VAE architecture takes inspiration from Ha and Schmidhuber [17]. We give the CNN-VAE a resized 96×96 RGB pixel image[3] with normalised values[4] in range $[0, 1]$ as input $\mathbf{X}$. The CNN-VAE passes $\mathbf{X}$ through its encoder – four convolutional layers – which maps $\mathbf{X}$ into two 32×1 low-dimensional vectors $\boldsymbol{\mu}$ and $\boldsymbol{\sigma}$ of a Gaussian distribution $\mathcal{N}$. Next, we then sample a 32×1 latent vector $\mathbf{z} \sim \mathcal{N}(\boldsymbol{\mu}, \boldsymbol{\sigma})$. Finally, we pass $\mathbf{z}$ through the decoder – four deconvolutional layers – to reconstruct the pixel image. The details of the architecture is shown in Fig. 3.

The CNN-VAE learns by minimising a loss function of two terms [14]:

$$L^{VAE} = L^R + L^{KL} \tag{1}$$

$$\text{where} \quad L^R = \sum_{i=1}^{N} \hat{x}_i \log x_i + (1 - \hat{x}_i) \log(1 - x_i) \tag{2}$$

$$L^{KL} = \frac{1}{2} \sum_{j=1}^{M} (1 + \log \sigma_j^2) - \mu_j^2 - \sigma_j^2 \tag{3}$$

Eq. (2) is the reconstruction loss between the original image $\mathbf{X}$ and its reconstructed counterpart $\hat{\mathbf{X}}$, where N is the total number of pixels in the image, and x_i along with $\hat{x}_i$ denotes the i-th pixel. Between a Mean Squared Error (MSE) loss and a Binary Cross Entropy (BCE) loss, we use the latter to quantify the reconstruction loss, at it was found to give the best results.

Eq. (3) is the Kullback-Leibler (KL) divergence. With this term we measure the similarity between the encoder generated distribution and a standard Gaussian distribution, $\mathcal{N}(0, 1)$. In doing so, this acts as a regularisation term, that enforces a Gaussian prior over the latent vectors $\mathbf{z}$, as it ensures that the encoder generated distribution is Gaussian. Here, μ_j and σ_j denote the j-th element of their respective vectors, and M is the size of $\mathbf{z}$.

Lastly, when we sample the latent vectors $\mathbf{z}$, we use the reparameterisation trick [14]:

$$\mathbf{z} = \boldsymbol{\mu} + \boldsymbol{\sigma}\boldsymbol{\epsilon} \quad \text{where} \quad \boldsymbol{\epsilon} \sim \mathcal{N}(0, 1) \tag{4}$$

Otherwise, we would not be able to backpropagate and compute the gradients.

2.2. Curiosity Learning with Random Network Distillation

With RND we use a predictor network $\hat{F}$ with parameter $\boldsymbol{\psi}$, and use its prediction errors as our measure of novelty to which we can derive intrinsic rewards from. However, with RND our predictor does not predict the next state, as such an approach is prone to the noisy TV problem, where the agent is intrinsically rewarded for seeking out local sources of entropy. Instead, we aim to predict the output of a fixed and randomly initialised target network F. This changes the nature of the prediction problem by making it one of a deterministic nature, which makes the agent not care about stochastic noise generated by the environment. Moreover, the predictor network $\hat{F}$ and target network F both have the same network architecture, which ensures that the prediction problem is within the modelling capacity of the predictor network.

The original work by Burda et al. [7] use a resized 84×84 pixel image of the current state as input, and accordingly their predictor and target networks are both CNNs. However, in our case there is no need for this, as we, instead, use the abstract and compact representations $\mathbf{z}$ – computed by the CNN-VAE – of the current state as input. As such, F and $\hat{F}$ are of a simpler nature in this work. They are, instead, Feedforward Neural Networks (FNNs) with three hidden layers of size 512, where all hidden layers use Exponential Linear Unit (ELU) activations.

We train the predictor network by minimising the expected MSE between the two networks [7]:

$$L^{MSE} = \|\hat{f}(\mathbf{z}; \boldsymbol{\psi}) - f(\mathbf{z})\|^2 \tag{5}$$

[1] A latent space is a meaningful lower-dimensional feature space, where similar items are positioned closer to each other.

[2] A model that learns a probability distribution which we can sample and generate values from.

[3] This resolution has been inspired by MuZero [12], that uses a 96×96 RGB resolution and also implements an MCTS algorithm. Other common resolutions are 64×64 RGB images as in Ha and Schmidhuber [17], or 84×84 RGB images as in Pathak et al. [7].

[4] We divide by 255.

w.r.t. the parameter $\boldsymbol{\psi}$. We use Eq. (5) as our intrinsic reward. Resultingly, when we encounter novel states that are different from the ones we have encountered and trained on before, we get a high loss and a high intrinsic reward. However, as we become more familiar with these states, their novelty diminishes, and likewise does our loss and intrinsic reward.

In this work, we use the RND reward to incentivise episodic curiosity-driven planning: The agent is encouraged to explore novel states, that has not yet been visited in the current episode, and discouraged to revisit familiar states, that it has already been to in the current episode. We achieve this by doing two things. First, we do not save the parameter updates of $\boldsymbol{\psi}$ between episodes, but instead reset $\boldsymbol{\psi}$ at the beginning of each episode. In doing so, our intrinsic RND reward becomes an episodic exploration bonus. This is different from the previous model-free works by Burda et al. [7] and Badia et al. [10], who both implement the RND reward as an inter-episodic bonus. Second, we update the RND network after each step the MCTS agent makes in the actual environment, such that the agent has an immediate notion of what is old and where it has been before[5].

Finally, we follow Burda et al. [7], and combine extrinsic and intrinsic rewards as follows:

$$R(s,a) = c^E R^E(s,a) + c^I R^I(\mathbf{z};\boldsymbol{\psi}) \tag{6}$$

where c^E and c^I are weight coefficients, that set the importance of the extrinsic reward relative to intrinsic reward.

2.3. Planning with Monte Carlo Tree Search

In RL, planning refers to using simulated experience generated by a model in order to determine the best course of action. MCTS is a tree-based decision-time planning method. Planning at decision time means that our planning only revolves around using simulated experiences to determine best action to do in the current state, and no further ahead than this. Meaning, when we have selected an action and are presented with a succeeding state, we execute our planning again, and so on. MCTS is a Monte Carlo method, where we estimate action values by simulating trajectories and average their returns. Subsequently, we then use these action value estimates to direct succeeding MCTS simulations toward even better trajectories, which in turn produces more accurate action value estimates [1]. We use an MCTS variant that takes inspiration from MuZero [12]. Each node in the search tree represents a state s and their corresponding edges $e(s,a)$ represents actions a from s. Each node stores the following statistics:

$$\left[N(s,a), R^E(s,a), \bar{Q}(s,a), P(s,a)\right]$$

where $N(s,a)$ is the visit count, $R^E(s,a)$ is the extrinsic reward given by the simulation environment, $\bar{Q}(s,a)$ is the mean action value and $P(s,a)$ is the prior action selection probability. The MCTS iteratively builds its search tree through K simulations, where each simulation begins with the current state s as root node and ends at a leaf node. Our simulation consists of the three steps shown in Fig 4.

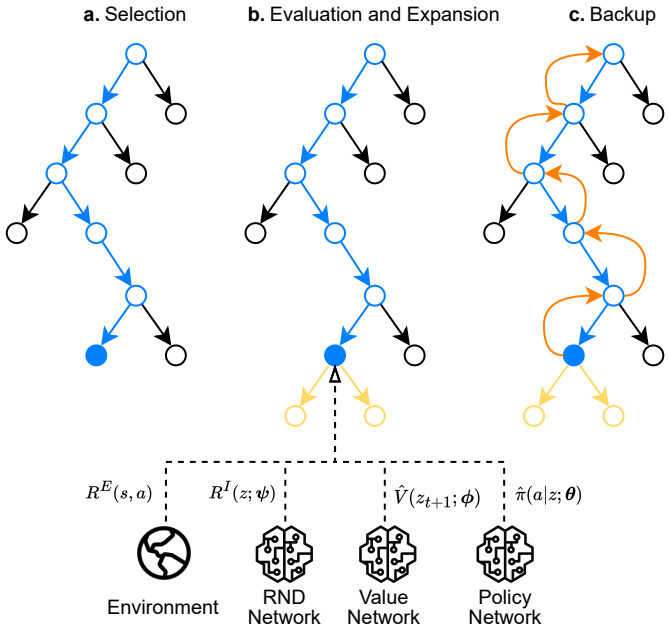

Fig. 4: The three steps of our MCTS simulation. **a.** Selection: We traverse the tree by selecting the children with the highest PUCT score, until we reach a leaf node. The tree selection traversal is shown in blue, and the selected leaf node is a filled blue node. **b.** Evaluation and Expansion: We evaluate the selected leaf node, where we get the leaf node's extrinsic reward r^E from the simulation environment, the intrinsic reward r^I from the RND network, the action selection probabilities $\hat{\pi}$ from the policy network, and the estimated expected discounted future return $\hat{V}$ from the value network. After evaluation, we expand the leaf node as shown in yellow. **c.** Backup: We update the node statistics of the search tree by traversing backwards through parent nodes. The backup is shown in orange.

2.3.1. Selection

We traverse the search tree from the root node to a leaf node using a Probabilistic Upper Confidence Threshold (PUCT) score, where we repeatedly select the highest scoring child node until we reach a leaf node. We use the PUCT score of MuZero [12]:

$$U(s,a) = \bar{Q}(s,a) + P(s,a)\frac{\sqrt{\sum_b N(s,b)}}{1+N(s,a)} \cdot \left(c_1 + \log\left(\frac{\sum_b N(s,b) + c_2 + 1}{c_2}\right)\right) \tag{7}$$

where a and b are possible actions in the current state s, and $c_1 = 1.25$ and $c_2 = 19.652$ are exploration constants that control the influence of $P(s,a)$ in relation to $\bar{Q}(s,a)$ as the action's visit count increases. In Eq. (7), the first term is an exploitation term and the second term is an exploration term. The general idea of using Eq. (7) as our search strategy is that we initially are explorative, where we prefer actions of high selection probabilities and low visit counts (high uncertainty). However, each

[5]The original model-free method by Burda et al. [7] update their RND every 128 steps. However, we argue that this makes less sense in a model-based context, as the agent would then first get a notion of what is old after 128 steps, and in the meantime unknowingly plan to explore states, that might already have been visited.

time action a is selected – its visit count increases – and our uncertainty in the estimate of $\bar{Q}(s,a)$ presumably decreases. As a result, the influence of a's exploration term decreases, and we will increasingly come to measure the merits of a solely on our estimate of $\bar{Q}(s,a)$. On the other hand, each time other sibling actions are visited, the influence of action a's exploration term increases, such that we increase the probability of exploring less favoured actions. Nonetheless, the use of the natural logarithm ensures that these influence increases gets smaller over time, but still in an unbounded manner [1]. All in all, with this search strategy we start explorative, but become more greedy as search time progresses.

2.3.2. Evaluation and Expansion

Now that we have reached a leaf node, the next step is to evaluate its value. Here, we use a value network $\hat{V}$ with parameter $\boldsymbol{\phi}$, an RND network with parameter $\boldsymbol{\psi}$, and a policy network $\hat{\pi}$ with parameter $\boldsymbol{\theta}$ to guide our MCTS, where all networks take the compact and abstract latent representation $\mathbf{z}$ – computed by the CNN-VAE – as input. The use of a policy network and value network is inspired by MuZero [12]. Commonly, a basic MCTS is implemented with four steps: a. Selection, b. Expansion, c. Rollout and d. Backup [1]. We skip the rollout step, which involves simulating a complete episode from the current leaf node until termination, in order to get a state value estimate $V(s)$. Instead, in this work, the value network provides us with these estimates. This removes the need of doing any rollouts, which effectively simplifies the MCTS procedure and reduces computation time. Our value network has two output heads and is trained to predict both the extrinsic and intrinsic expected discounted future return, $\hat{V}^E(\mathbf{z}_{t+1};\boldsymbol{\phi})$ and $\hat{V}^I(\mathbf{z}_{t+1};\boldsymbol{\phi})$. The value network is trained with an MSE loss function, where we do Monte Carlo updates and use the complete extrinsic and intrinsic discounted return, G_t^E and G_t^I, of a whole episode as our targets. We compute G_t^E and G_t^I as follows:

$$G_t^E = \sum_{k=t+1}^{T} \gamma_E^{k-t} r_k^E \tag{8}$$

$$G_t^I = \sum_{k=t+1}^{T} \gamma_I^{k-t} r_k^I \tag{9}$$

where T is the length of an episode, and γ_E and γ_I are discount factors. We follow Burda et al. [7] and set $\gamma_E = 0.999$ and $\gamma_I = 0.99$, such that our agent is far-sighted in an environment of sparse rewards.

Using the extrinsic reward r^E given by the simulation environment's reward function R^E, the intrinsic reward r^I from the RND network R^I and the future expected return from the value network $\hat{V}$, we expand upon Eq. (6) and compute the value v of a leaf node n as follows:

$$v_n = c^E\Big(r^E + \hat{V}_{t+1}^E(\mathbf{z};\boldsymbol{\phi})\Big) + c^I\Big(r^I + \hat{V}_{t+1}^I(\mathbf{z};\boldsymbol{\phi})\Big) \tag{10}$$

After evaluation, we ask the policy network for action selection preferences, and expand the leaf node by creating its offspring. We initialise each newly created child node as follows:

$$\Big[N(s,a)=0, R^E(s,a)=r^E, \bar{Q}(s,a)=0, P(s,a)\sim\hat{\pi}(a|\mathbf{z};\boldsymbol{\theta})\Big]$$

Furthermore, following MuZero [12], we add Dirichlet noise to the root node's prior action selection probabilities:

$$P(s,a) = (1-\epsilon)P(s,a) + \epsilon\eta \tag{11}$$

where $\eta \sim \mathrm{Dir}(0.03)$ and $\epsilon = 0.25$. This is done to achieve additional exploration, such that we ensure all actions from the root node are tried.

2.3.3. Backup

Next, we update the node statistics of all nodes that have been visited during the MCTS simulation. This is done by traversing backwards through the parent nodes. We update the statistics of each visited node as follows [12]:

$$\begin{aligned} N(s,a) &\leftarrow N(s,a)+1 \\ \bar{Q}(s,a) &\leftarrow \frac{N(s,a)\bar{Q}(s,a)+v_n}{N(s,a)+1} \end{aligned} \tag{12}$$

2.3.4. MCTS Policy

Finally, after K simulations, we sample an action to take in the actual environment from the MCTS policy, which is the visit count distribution of the root node [12]:

$$\pi(a|s) = \begin{cases} \dfrac{N(s,a)^{1/\tau}}{\sum_b N(s,b)^{1/\tau}}, & \tau > 0 \\ \arg\max\limits_a \Big(N(s,a)\Big), & \tau = 0 \end{cases} \tag{13}$$

where $\tau \in [0,1]$ is a greediness parameter. I.e. when $\tau = 1.0$ we are explorative, whereas as $\tau \to 0$ we become more greedy. We start by setting $\tau = 1.0$ at the beginning of our training, then when we reach 50% of all updates we set $\tau = 0.5$, and finally when we reach 75% of all updates we set $\tau = 0.25$. As a result, our MCTS policy becomes more greedy as training progresses.

The policy network plays an integral role in guiding our MCTS search, as we sample $P(s,a) \sim \hat{\pi}(a|\mathbf{z};\boldsymbol{\theta})$, where $P(s,a)$ heavily influences the PUCT-score, shown in Eq.(7), and consequently, how we traverse down the tree during selection. When the MCTS has finished its search, the end result is usually a visit count distribution π that is better than our policy network $\hat{\pi}$. Therefore, it subsequently makes sense to train the policy network to match the improved visit count distribution of the MCTS. This is done by minimising the cross entropy between the policy network's action selection distribution and the MCTSs visit count distributions. The updated policy network is then used in the next MCTS iteration, which again improves the policy network. In consequence, the MCTS variant used in this work, acts as a policy improvement loop [18].

3. Frozen Lake Experiment

Frozen Lake [15] is a simple grid environment, where the agent's goal is to cross a frozen lake from start to goal without falling into any holes. For this experiment we use the 8×8 map shown in Fig 5.

Fig. 5: The 8×8 Frozen Lake map used in this experiment. The agent starts in the top-left corner and its goal is to reach the gift in the bottom-right corner. The agent dies if it fall into a water hole.

By default, the observation space is a discrete integer space. However, we have modified the environment such that the observation space is of RGB pixel images. The action space consists of four moves: Left, down, right and up. The goal can be reached with a minimum of 14 steps. The environment is of sparse extrinsic rewards:

$$r^E = \begin{cases} +1, & \text{goal reached} \\ 0, & \text{otherwise} \end{cases} \tag{14}$$

Finally, we have kept this experiment as simple as possible and the lake is not slippery. Meaning, the agent is not at risk of slipping to unintended directions when moving.

3.1. Setup and Training

In this experiment our agent has access to a perfect simulator, as we use the actual environment itself as our MCTS simulation environment. We feed the CNN-VAE a 96×96 pixel input[6]. We implement our value network as a Recurrent Neural Network (RNN) [19] – specifically, a Gated Recurrent Unit (GRU) network [20] – as the intrinsic rewards cannot directly be inferred from the current state, but depends on the agent's history. The policy network, on the other hand, is implemented as an FNN, as the optimal actions can be inferred directly from the current state[7]. We initialise our RND network with an orthogonal weight initialisation [21] using a gain of $\sqrt{3}$, as this by coincidence was found to produce intrinsic rewards that mostly are in a desired range around $[0, 1]$ in the Frozen Lake environment. All setup details of the curiosity MCTS agent are shown in Appendix A.

We use two baselines. First, we have a model-based baseline – a vanilla MCTS agent – that uses the same setup as the curiosity MCTS agent, but without the RND network. Second, we have a model-free baseline – a vanilla PPO agent – that uses the CNN architecture of Mnih et al [22], and as input receives a temporal frame stack of the latest four 84×84 grey-scaled pixel images.

All training was done using a desktop[8] with a single GPU. The CNN-VAEs training consisted of 135 updates with 512 steps of random actions per update. Both the curiosity MCTS and vanilla MCTS agent trained for 1024 steps with 600 MCTS simulations per step (~3 hour run time). The vanilla PPO agent trained for 1 000 000 steps (~3.5h run time).

3.2. Results

Despite Frozen Lake's seemingly simple nature, the visual environment of sparse rewards is, surprisingly, too difficult to solve for both baselines. Both spend all their time aimlessly wandering the environment without even reaching the goal a single time. This goes to show how RL – both model-based and model-free – falls short in visual environments of sparse rewards, as they are clueless of what to do. The curiosity MCTS agent, on the other hand, manages to solve the environment. Its learning results are shown in Tab. 1.

Update	Goal reached	Average steps to goal
0 / 7	1 / 6	17.0
1 / 7	4 / 5	24.0
2 / 7	5 / 7	18.0
3 / 7	5 / 8	18.2
4 / 7	7 / 8	16.6
5 / 7	8 / 8	15.7
6 / 7	8 / 8	15.6
7 / 7	8 / 8	15.0

Tab. 1: Learning results of the curiosity MCTS agent.

As shown in Tab. 1, prior to any updates, the intrinsic RND reward encourages curiosity-driven planning, as the curiosity MCTS agent is already able to find the goal 1 / 6 times with a randomly initialised policy and value network. After the 5th update, the curiosity MCTS agent is able to consistently find the goal. However, it never learns to consistently do so in the least amount of possible steps of 14 during its training.

With these results, we can conclude that adding an intrinsic RND reward to an MCTS agent enhances learning and encourages curiosity-driven planning in a visual environment of sparse extrinsic rewards, as this environment is, otherwise, unsolvable without curiosity.

[6] For an example of a resized 96×96 image and its CNN-VAE reconstruction, see Appendix D.

[7] Out of curiosity we also tried implementing the policy network as an RNN in two ways. The First RNN policy network was given the latest eight states. The second RNN policy network was given the latest eight state-action pairs. We found that both RNN policy networks performed worse than the FNN policy network. Furthermore, we also tried implementing the policy network and value network as a single RNN of three output heads and a joint loss function, which also performed poorly.

[8] Desktop specifications: I7-11700F 2.5 GHz CPU, 32GB RAM and NVIDIA GeForce RTX 3060 GPU.

4. World Models

Up and until now, the curiosity MCTS agent have relied on planning using a perfect mental model of the world, as it had direct access to the environment's dynamics. This naturally restricts the general applicability of the curiosity MCTS agent, as it requires access to a perfect simulator in order to do its planning – a prerequisite that might not always feasible or practical in real world domains. By contrast, we, humans do not have this prerequisite imposed on us. We do not have a perfect mental model of the world at our disposal. Instead, we have from experience learned a visual and temporal abstract world model, which we use to predict the outcome of different courses of action, such that we can plan accordingly. According to Machado et al. [23] designing a world model that is fast and accurate, and subsequently using an imperfect world model for planning, remain two open problems. In this section, we concern ourselves with the first problem, and shortly explore building an abstract world model[9] for visual environments of sparse rewards, that then can be used as the MCTS simulation environment in future works. We take inspiration from Ha and Schmidhuber [17], who are able to train a model-free agent by only having it learn inside its own generated world model.

Ha and Schmidhuber [17] propose building a predictive world model using a CNN-VAE and a Mixture Density Network Recurrent Neural Network (MDN-RNN). As we already have the first component, the CNN-VAE, we in this section look into building the MDN-RNN. The MDN-RNN is an RNN combined with an MDN [29] as the output layer. It is a generative prediction model that is given the history H_t of the T latest latent state action pairs $(\mathbf{z}_t, a_t)$, and predicts the future. In our case, where our goal is to have a world model that can be used as an MCTS simulation environment, the MDN-RNN needs to predict two aspects of the future: The next latent state $\mathbf{z}_{t+1}$ and whether or not the agent dies in the next step d_{t+1}. With an MDN-RNN we generate the next latent state $\mathbf{z}_{t+1}$ using a Gaussian Mixture Model (GMM) – a weighted sum of multiple Gaussians – where the probability distribution P of a GMM with K components is expressed as follows [29]:

$$P(\mathbf{z}_{t+1} \mid H_t; \boldsymbol{\theta}) = \sum_{k=1}^{K} \boldsymbol{\Pi}_k(H_t; \boldsymbol{\theta}) \mathcal{N}\Big(\mathbf{z}_{t+1} \;\Big|\; \boldsymbol{\mu}_k(H_t; \boldsymbol{\theta}), \boldsymbol{\sigma}_k(H_t; \boldsymbol{\theta})\Big) \tag{15}$$

where $\boldsymbol{\theta}$ is the parameter of the MDN-RNN, $\mathcal{N}_k$ is the k-th Gaussian component with mixing weight $\boldsymbol{\Pi}_k$, mean $\boldsymbol{\mu}_k$ and standard deviation $\boldsymbol{\sigma}_k$. All of $\boldsymbol{\sigma}_k$ are positive, and all weights $\boldsymbol{\Pi}_k$ sum to one. We follow Ha and Schmidhuber [17] and set the number of GMM components, $K = 5$.

From Eq. (15) it follows that the MDN-RNN needs three output heads, such that we can produce $\boldsymbol{\Pi}_k$, $\boldsymbol{\mu}_k$ and $\boldsymbol{\sigma}_k$ for each Gaussian component, in order to fashion a GMM. The three outputs are learned using an MDN loss, which is defined as the negative logarithm of the likelihood, or in other words, the loss between the predicted distribution vs. the actual distribution [29]:

$$L^{MDN} = -\log\left[\sum_{k=1}^{K} \boldsymbol{\Pi}_k(H_t; \boldsymbol{\theta}) \mathcal{N}\Big(\mathbf{z}_{t+1} \;\Big|\; \boldsymbol{\mu}_k(H_t; \boldsymbol{\theta}), \boldsymbol{\sigma}_k(H_t; \boldsymbol{\theta})\Big)\right] \tag{16}$$

With a GMM we can generate the next latent state $\mathbf{z}_{t+1}$ in two ways:

$$\mathbf{z}_{t+1} \sim P(\mathbf{z}_{t+1} \mid H_t; \boldsymbol{\theta}) \tag{17}$$

$$\mathbf{z}_{t+1} = \sum_{k=1}^{K} \boldsymbol{\Pi}_k \boldsymbol{\mu}_k \tag{18}$$

Eq. (17) is for stochastic environments, where we sample $\mathbf{z}_{t+1}$ from P, and Eq. (18) is for deterministic environments, where we assume $\mathbf{z}_{t+1}$ to be the weighted mean of P.

Finally, the MDN-RNN also needs the ability to predict whether or not the agent dies in the next step, if it is to be used as an MCTS simulation environment. As such, our MDN-RNN has a fourth output head, that predicts the agent's death as a binary classification. Thus, our MDN-RNN has a total of four output heads, and a joint loss of two terms[10]:

$$L^{MDN-BCE} = L^{MDN} + L^{BCE} \tag{19}$$

where the first term is the MDN loss of Eq. (16), and the second term is the BCE loss between the MDN-RNNs binary death prediction vs. whether or not the agent actually dies. An overview of the MDN-RNN is shown in Fig. 6. In order to use the world

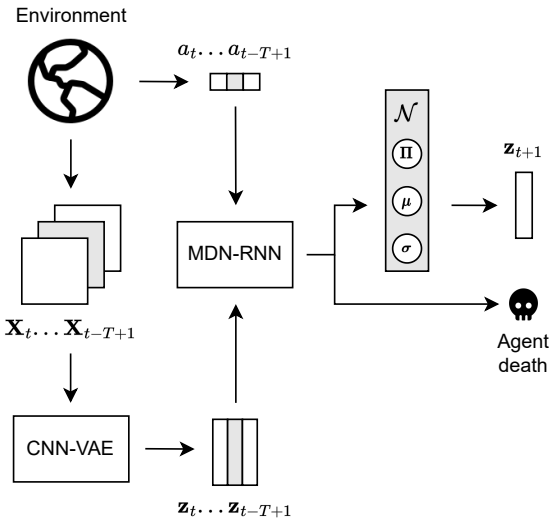

Fig. 6: Overview of the MDN-RNN.

[9] For further readings on the concept of world models, see the works by Schmidthuber [24], Kaiser et al. [25], and Hafner et al. [26] [27] [28].

[10] At first, we followed Ha and Schmidhuber [17] and only used L^{MDN} of Eq. (16) as the loss function. However, we found $L^{MDN-BCE}$ of Eq. (19) to give better results in terms of death predictions. Nonetheless, there is trade-off. The improved death predictions are at the cost of a slightly higher MDN-RNN loss than its counterpart. For further details, see Appendix C.

model as an MCTS simulation environment, we naturally need to predict further into the future, beyond $\mathbf{z}_{t+1}$. To do so, we need to compute succeeding predictions through bootstrapping, where succeeding predictions are based on preceding predictions. Inevitably, it follows that our predictions about the future will increasingly become less accurate the further ahead we look into the future, as predictions errors will accumulate. As such, the world model needs to be very accurate in its predictions, if it is be used as an MCTS simulation environment.

5. Montezuma's Revenge Experiment

Ha and Schmidhuber [17] have shown that their world model is able to emulate two fairly simple environments with good results. First, a simple driving control task in a top-down Car Racing pixel environment [30]. Second, a simple movement control task, where the agent has to move from side to side in order to dodge fireballs shot by monsters in the VizDoom pixel environment [31]. In this section we investigate if their world model concept is applicable to more complex environment dynamics. The Atari game, Montezuma's Revenge (MR)[11], is known to be a difficult environment, due to its complex dynamics and sparse rewards [23]. As such, we in this section explore the world model's capabilities in emulating the first room of MR. The layout of the first room is shown in Fig. 7.

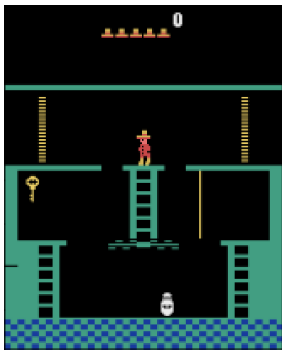

Fig. 7: The first room of MR. To exit the first room the agent has to descend the middle ladder, jump right using the rope, descend the bottom-right ladder, jump over a deadly moving skull, climb the bottom-left ladder, collect the key, then back track, after which the agent can exit by unlocking one of the two yellow doors using the collected key. In this room the agent can die by falling from any place, or when touching the moving skull that rolls back and forth. The agent has six lives, and upon losing one the agent respawns on top of the middle ladder. The game resets when all six lives are lost.

5.1. Training

All training was done using a GPU server[12]. The CNN-VAE trained for 100 updates with 1024 random steps per update (~3h run time), and the MDN-RNN trained for 300 updates with 1024 steps per update (~ 45 min. run time). To improve upon our death predictions, we augmented the training data sets by adding all sequences with death occurrences eight additional times. All setup details are shown in Appendix B.

5.2. Results

When predicting the future in the first room of MR, the world model only has to predict the future position of two entities. Namely, its own position and the moving skull's position. all other parts of the environment are static. We evaluated the world model with 10 episodes of 1024 random actions, where our world model achieved an average MDN-RNN loss of 1.28. Naturally, it is hard to tell by the number itself whether or not this is a reasonable loss. However, upon visual inspection[13], we see that such a loss is not satisfactory. The MDN-RNN is capable of predicting the future position of the moving skull. On the other hand, the MDN-RNN fails at multiple times to predict the future position of the agent. We observe that the MDN-RNN has a tendency to wrongly predict the agent to be on the middle ladder, or on the rope. We suspect that this might be due to the fact that we have only trained the MDN-RNN using random actions, which can lead it to getting stuck on either the middle ladder or rope. As such, the MDN-RNNs wrong predictions might be a case of overfitting, as a consequence of a too simple experience gathering method, that is done solely through random actions. Another explanation could be that the current network architecture has a limited modelling capacity, and the first room of MR is simply out of its reach. Besides this, the world model is also too inaccurate in its death predictions, as it only gets an F1-score of 57.10. All result details are shown in Appendix C. In light of these results, we can conclude that our world model is inapplicable as an MCTS simulation environment in its current state. It does not have the sufficient abilities to emulate the first room of MR, as it is inadequate in both future latent state predictions and death predictions, due to either an inefficient experience gathering method through random actions, a limited modelling capacity, or both.

6. Conclusion and Future Work

In this work we introduced, curiosity MCTS, a novel model-based RL-agent, that uses the concept of Random Network Distillation (RND) to generate episodic intrinsic rewards and incentivise curiosity-driven planning. We have demonstrated that curiosity enhances a model-based RL agent's learning in a visual environment of sparse rewards, that is otherwise unsolvable by both model-free and model-based agents without curiosity. Finally, we explored the prospect of building a world model, that can be used as the MCTS simulation environment, such that there is no need for a perfect simulator. We, however, found our world model to be inadequate in emulating a complex environment such as the first room of Montezuma's Revenge (MR), due to either a too simple experience gathering method using solely random actions, a limited modelling capacity, or both.

[11] For further readings on works that try to solve MR, see the works by Bellemare et al. [3], Burda et al. [7], Badia et al. [10], Mnih et al. [22], Hafner et al. [27] and Ecoffet et al. [32].

[12] CLAAUDIA GPU server [33].

[13] For visual examples, see Appendix E.

For future work, the overall goal is to develop a world model that is capable of emulating MR, and then use this as the simulation environment of the curiosity MCTS agent. The whole game of MR is only of partial observability – as the game consists of multiple rooms – and each room requires a complicated set of actions to get rewards due to obstacles, timing-based traps and moving adversaries.

To improve upon the modelling capacity of our current world model there are multiple directions to explore. First, we can take inspiration from Gemici et al. [34], and augment our world model with an external memory system, as this has shown to substantially improve performance on problems with sparse and long-term temporal dependencies. Another interesting world model architecture to explore is the recurrent state space model of Hafner et al. [26], along with its loss function, called latent overshooting, that enables multi-step predictions. Finally, the prospect of using transformers [35] instead of RNNs could also be interesting to explore, as transformers have achieved state-of-the performance in different application such as natural language processing [36], biology [37] and computer vision [38].

To improve upon our experience gathering method and also deal with the partial observability of MR, we propose an iterative training procedure of the curiosity MCTS agent, where we initially train our world model using experience gathered from random actions, and subsequently, iteratively refine our world model using experience gathered by the MCTS agent.

Finally, in more complex environments of sparse rewards, we need to ensure a high level of exploration. Otherwise, if the agent becomes greedy and extrinsically reward-driven too fast, it might converge prematurely to a suboptimal policy and never learn to act optimally. To prevent this, we can take inspiration from Badia et al. [10], and have an intrinsic reward that is both episodic and inter-episodic, such that we ensure adequate exploration both within and across episodes. In doing so, our agent is encouraged to revisit states – that might still hold undiscovered treasures, e.g. a locked door – across different episodes, and at the same time, slowly grow familiar with states that have been visited many times across many episodes. Here, we propose to use two separate RND networks: One for episodic intrinsic rewards and one for life-long inter-episodic intrinsic rewards.

7. Code Availability

- Curiosity MCTS repository:
  `https://github.com/kim-ngu/curiosity-MCTS`
- PPO implementation modified from repository by Huang et al. [39]: `https://github.com/vwxyzjn/ppo-implementation-details`
- RND implementation inspired by:
  `https://github.com/jcwleo/random-network-distillation-pytorch`
- MCTS implementation inspired by:
  `https://github.com/ciamic/MCTS`
  `https://github.com/dylandjian/SuperGo`

References

[1] R. S. Sutton, A. G. Barto, Reinforcement Learning: An Introduction, 2nd Edition, The MIT Press, 2018.

[2] C. Kidd, B. Hayden, The psychology and neuroscience of curiosity, Neuron 88 (2015) 449–460. doi:10.1016/j.neuron.2015.09.010.

[3] M. G. Bellemare, S. Srinivasan, G. Ostrovski, T. Schaul, D. Saxton, R. Munos, Unifying count-based exploration and intrinsic motivation (2016). arXiv:1606.01868.

[4] J. Schmidhuber, A possibility for implementing curiosity and boredom in model-building neural controllers, in: Proceedings of the First International Conference on Simulation of Adaptive Behavior on From Animals to Animats, MIT Press, Cambridge, MA, USA, 1991, p. 222–227.

[5] B. C. Stadie, S. Levine, P. Abbeel, Incentivizing exploration in reinforcement learning with deep predictive models (2015). arXiv:1507.00814.

[6] D. Pathak, P. Agrawal, A. A. Efros, T. Darrell, Curiosity-driven exploration by self-supervised prediction (2017). arXiv:1705.05363.

[7] Y. Burda, H. Edwards, A. Storkey, O. Klimov, Exploration by random network distillation (2018). arXiv:1810.12894.

[8] Y. Burda, H. Edwards, D. Pathak, A. Storkey, T. Darrell, A. A. Efros, Large-scale study of curiosity-driven learning (2018). arXiv:1808.04355.

[9] D. Pathak, D. Gandhi, A. Gupta, Self-supervised exploration via disagreement (2019). arXiv:1906.04161.

[10] A. P. Badia, P. Sprechmann, A. Vitvitskyi, D. Guo, B. Piot, S. Kapturowski, O. Tieleman, M. Arjovsky, A. Pritzel, A. Bolt, C. Blundell, Never give up: Learning directed exploration strategies (2020). arXiv:2002.06038.

[11] O. Vinyals, I. Babuschkin, W. M. Czarnecki, M. Mathieu, A. Dudzik, J. Chung, D. H. Choi, R. Powell, T. Ewalds, P. Georgiev, et al., Grandmaster level in starcraft ii using multi-agent reinforcement learning, Nature 575 (7782) (2019) 350–354. doi:10.1038/s41586-019-1724-z.

[12] J. Schrittwieser, I. Antonoglou, T. Hubert, K. Simonyan, L. Sifre, S. Schmitt, A. Guez, E. Lockhart, D. Hassabis, T. Graepel, et al., Mastering atari, go, chess and shogi by planning with a learned model, Nature 588 (7839) (2020) 604–609. doi:10.1038/s41586-020-03051-4.

[13] R. Coulom, Efficient selectivity and backup operators in monte-carlo tree search, in: H. J. van den Herik, P. Ciancarini, H. H. L. M. J. Donkers (Eds.), Computers and Games, Springer Berlin Heidelberg, Berlin, Heidelberg, 2007, pp. 72–83.

[14] D. P. Kingma, M. Welling, Auto-encoding variational bayes (2022). arXiv:1312.6114.

[15] F. Foundation, Frozen lake, access date: 11/05/2023 (2022).
URL https://gymnasium.farama.org/environments/toy_text/frozen_lake/

[16] J. Schulman, F. Wolski, P. Dhariwal, A. Radford, O. Klimov, Proximal policy optimization algorithms (2017). doi:10.48550/ARXIV.1707.06347.

[17] D. Ha, J. Schmidhuber, Recurrent world models facilitate policy evolution, in: Advances in Neural Information Processing Systems 31, Curran Associates, Inc., 2018, pp. 2451–2463, https://worldmodels.github.io.
URL https://papers.nips.cc/paper/7512-recurrent-world-models-facilitate-policy-evolution

[18] D. Silver, J. Schrittwieser, K. Simonyan, I. Antonoglou, A. Huang, A. Guez, T. Hubert, L. Baker, M. Lai, A. Bolton, Y. Chen, T. Lillicrap, F. Hui, L. Sifre, G. v. d. Driessche, T. Graepel, D. Hassabis, Mastering the game of Go without human knowledge, Nature 550 (7676) (2017) 354–359. doi:10.1038/nature24270.

[19] D. E. Rumelhart, G. E. Hinton, R. J. Williams, Learning representations by back-propagating errors, Nature 323 (1986) 533–536. doi:https://doi.org/10.1038/323533a0.

[20] K. Cho, B. van Merrienboer, D. Bahdanau, Y. Bengio, On the properties of neural machine translation: Encoder-decoder approaches (2014). doi:10.48550/ARXIV.1409.1259.

[21] A. M. Saxe, J. L. McClelland, S. Ganguli, Exact solutions to the nonlinear dynamics of learning in deep linear neural networks (2014). arXiv:1312.6120.

[22] V. Mnih, K. Kavukcuoglu, D. Silver, A. A. Rusu, J. Veness, M. G. Bellemare, A. Graves, M. Riedmiller, A. K. Fidjeland, G. Ostrovski, S. Petersen, C. Beattie, A. Sadik, I. Antonoglou, H. King, D. Kumaran, D. Wierstra, S. Legg, D. Hassabis, Human-level control through deep re-

inforcement learning, Nature 518 (7540) (2015) 529–533. doi:10.1038/nature14236.
[23] M. C. Machado, M. G. Bellemare, E. Talvitie, J. Veness, M. Hausknecht, M. Bowling, Revisiting the arcade learning environment: Evaluation protocols and open problems for general agents (2017). arXiv:1709.06009.
[24] J. Schmidhuber, On learning to think: Algorithmic information theory for novel combinations of reinforcement learning controllers and recurrent neural world models (2015). arXiv:1511.09249.
[25] L. Kaiser, M. Babaeizadeh, P. Milos, B. Osinski, R. H. Campbell, K. Czechowski, D. Erhan, C. Finn, P. Kozakowski, S. Levine, A. Mohiuddin, R. Sepassi, G. Tucker, H. Michalewski, Model-based reinforcement learning for atari (2020). arXiv:1903.00374.
[26] D. Hafner, T. Lillicrap, I. Fischer, R. Villegas, D. Ha, H. Lee, J. Davidson, Learning latent dynamics for planning from pixels (2019). arXiv:1811.04551.
[27] D. Hafner, T. Lillicrap, M. Norouzi, J. Ba, Mastering atari with discrete world models (2022). arXiv:2010.02193.
[28] D. Hafner, J. Pasukonis, J. Ba, T. Lillicrap, Mastering diverse domains through world models (2023). arXiv:2301.04104.
[29] C. Bishop, Mixture density networks, Workingpaper, Aston University (1994).
[30] O. Klimov, Car racing, access date: 31/05/2023 (2023).
URL https://gymnasium.farama.org/environments/box2d/car_racing/
[31] M. Wydmuch, M. Kempka, W. Jaśkowski, ViZDoom Competitions: Playing Doom from Pixels, IEEE Transactions on Games 11 (3) (2019) 248–259, the 2022 IEEE Transactions on Games Outstanding Paper Award. doi:10.1109/TG.2018.2877047.
[32] A. Ecoffet, J. Huizinga, J. Lehman, K. O. Stanley, J. Clune, First return, then explore, Nature 590 (7847) (2021) 580–586. doi:10.1038/s41586-020-03157-9.
URL https://doi.org/10.1038%2Fs41586-020-03157-9
[33] CLAAUDIA, Claaudia overview, access date: 01/06/2023 (2023).
URL https://aicloud-docs.claaudia.aau.dk/overview/
[34] M. Gemici, C.-C. Hung, A. Santoro, G. Wayne, S. Mohamed, D. J. Rezende, D. Amos, T. Lillicrap, Generative temporal models with memory (2017). arXiv:1702.04649.
[35] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, L. Kaiser, I. Polosukhin, Attention is all you need (2017). doi:10.48550/ARXIV.1706.03762.
[36] T. B. Brown, B. Mann, N. Ryder, M. Subbiah, J. Kaplan, P. Dhariwal, A. Neelakantan, P. Shyam, G. Sastry, A. Askell, S. Agarwal, A. Herbert-Voss, G. Krueger, T. Henighan, R. Child, A. Ramesh, D. M. Ziegler, J. Wu, C. Winter, C. Hesse, M. Chen, E. Sigler, M. Litwin, S. Gray, B. Chess, J. Clark, C. Berner, S. McCandlish, A. Radford, I. Sutskever, D. Amodei, Language models are few-shot learners (2020). arXiv:2005.14165.
[37] J. M. Jumper, R. Evans, A. Pritzel, T. Green, M. Figurnov, O. Ronneberger, K. Tunyasuvunakool, R. Bates, A. Zídek, A. Potapenko, A. Bridgland, C. Meyer, S. A. A. Kohl, A. Ballard, A. Cowie, B. Romera-Paredes, S. Nikolov, R. Jain, J. Adler, T. Back, S. Petersen, D. A. Reiman, E. Clancy, M. Zielinski, M. Steinegger, M. Pacholska, T. Berghammer, S. Bodenstein, D. Silver, O. Vinyals, A. W. Senior, K. Kavukcuoglu, P. Kohli, D. Hassabis, Highly accurate protein structure prediction with alphafold, Nature 596 (2021) 583 – 589. doi:https://doi.org/10.1038/s41586-021-03819-2.
[38] A. Dosovitskiy, L. Beyer, A. Kolesnikov, D. Weissenborn, X. Zhai, T. Unterthiner, M. Dehghani, M. Minderer, G. Heigold, S. Gelly, J. Uszkoreit, N. Houlsby, An image is worth 16x16 words: Transformers for image recognition at scale (2021). arXiv:2010.11929.
[39] S. Huang, R. F. J. Dossa, A. Raffin, A. Kanervisto, W. Wang, The 37 implementation details of proximal policy optimization (2022).
URL https://iclr-blog-track.github.io/2022/03/25/ppo-implementation-details/
[40] D. P. Kingma, J. Ba, Adam: A method for stochastic optimization (2014). doi:10.48550/ARXIV.1412.6980.
[41] R. Pascanu, T. Mikolov, Y. Bengio, On the difficulty of training recurrent neural networks (2013). arXiv:1211.5063.

Appendix A. Curiosity MCTS Setup

CNN-VAE	**Setting**
Network architecture	See Fig. 3
Optimiser	Adam [40]
Learning rate	$1.0 \cdot 10^{-3}$
Number of updates	135
Steps per update	512
Number of epochs	20
Batch size	32
Loss function	BCE loss + KL Divergence, see Eq. (1)
RND Network	**Setting**
Network architecture	FNN: 3 h-layers w. size 512, ELU acts.
Optimiser	Adam [40]
Learning rate	$1.0 \cdot 10^{-4}$
Number of updates	1
Steps per update	1
Number of epochs	8
Batch size	1
Extrinsic discount factor γ_E	0.999
Intrinsic discount factor γ_I	0.99
Extrinsic weight coef. c^E	3.0
Intrinsic weight coef. c^I	1.0
Loss function	MSE, see Eq. (5)
Value Network	**Setting**
Network architecture	GRU: 1 h-layer w. size 256
Sequence length	8
Optimiser	Adam [40]
Learning rate	$1.0 \cdot 10^{-5}$
Number of updates	8
Steps per update	128
Number of epochs	10
Batch size	16
Gradient value clipping	0.5
Loss function	MSE
Policy Network	**Setting**
Network architecture	FNN: 4 h-layers w. size 512, ELU acts.
Optimiser	Adam [40]
Learning rate	$1.0 \cdot 10^{-5}$
Number of updates	8
Steps per update	128
Number of epochs	10
Batch size	16
Loss function	Cross Entropy
MCTS	**Setting**
Simulations per step	600
Dirichlet α	0.03
Dirichlet ϵ	0.25

Tab. A.2: Curiosity MCTS setup

Appendix B. World Model Setup

CNN-VAE	Setting
Network architecture	See Fig. 3
Optimiser	Adam [40]
Learning rate	$1.0 \cdot 10^{-3}$
Number of updates	100
Steps per update	1024
Number of epochs	10
Batch size	32
Loss function	BCE loss + KL Divergence, see Eq. (1)
MDN-RNN	**Setting**
Network architecture	GRU: 1 h-layer w. size 512
Sequence length	32
Optimiser	Adam [40]
Learning rate	$1.0 \cdot 10^{-3}$
Number of updates	300
Steps per update	1024
Number of epochs	20
Batch size	32
Gradient value clipping	0.5
Number of GMM components	5
Loss function	MDN loss + BCE loss, see Eq. (19)

Tab. B.3: World Model setup

Instability Issues

When building the MDN-RNN, we discovered that it is rather unstable and prone to producing NaN values for two reasons: An exploding gradient problem [41] with the RNN, and numerical instability issues with the MDN. To address the exploding gradient problem, we added gradient value clipping, such that the gradients are clipped when they exceed the range [-0.5, 0.5]. To combat the numerical instability issue with the MDN we have modified how the MDN loss is calculated. We added a stability parameter, $\epsilon = 10^{-5}$, at log operations, $\log(x + \epsilon)$, to prevent taking the log of zero, we added a ReLU function prior to log operations, to prevent taking the log of a negative number, and finally, we used the log-sum-exp trick to prevent numerical overflows when using the exponential function, e^x, with large values of x. For the exact implementation details see the code provided in section 7.

Alternative MDN-RNN Architecture

Besides the MDN-RNN architecture presented in section 4, we also tried an alternative MDN-RNN architecture: RNN + 2 FNNs. Here, the RNN receives the the history H_t as input, and produced the hidden state $\mathbf{h}_t$ as output. We then separate the task of predicting the next latent state $\mathbf{z}_{t+1}$ and death predictions d_{t+1} into two FNNs, $P_1(\mathbf{z}_{t+1}|\mathbf{h}_t)$ and $P_2(d_{t+1}|\mathbf{h}_t)$. This variant was found to be less successful, as illustrated in the results in Appendix C.

Appendix C. World Model Results

Avg. loss	TP	FP	PPV	TPR	FPR	TNR	F1
1.28	177/279	164/713	51.91	63.44%	23.03%	76.97%	57.10

Tab. C.4: Prediction results of world model with joint loss, eq. (19).

Avg. loss	TP	FP	PPV	TPR	FPR	TNR	F1
1.16	134/279	144/713	48.20	48.03%	20.22%	79.78%	48.11

Tab. C.5: Prediction results of world model with only MDN loss, eq. (16).

Avg. loss	TP	FP	PPV	TPR	FPR	TNR	F1
1.59	62/279	74/713	45.59	22.22%	10.39%	89.61%	29.88

Tab. C.6: Prediction results of world model with an alternative MDN-RNN architecture: RNN + 2 FNNs.

All values are averages computed from 10 episodes of 1024 random steps. For death predictions our threshold is set to 0.5. Meaning, values ≥ 0.5, are classified as deaths, and values below are not. The first column is the average MDN-RNN loss using Eq. (19). The second column is True Positives (TP). The third column is False Positives (FP). The fourth column is the Positive Predictive Value (PPV), which is the proportion of positive classification that were correct. The fifth column is the True Positive Rate (TPR), which is the probability of correctly classifying a true positive. The sixth column is the False Positive Rate (FPR), which is the probability of wrongly classifying a true negative. The seventh column is the True Negative Rate (TNR), which the probability of correctly classifying a true negative. The eighth column is the F1-score, which is a measure that combines PPV and TPR into a single metric, with 100 as the highest score and 0 as the lowest score.

Appendix D. Frozen Lake Visual Example

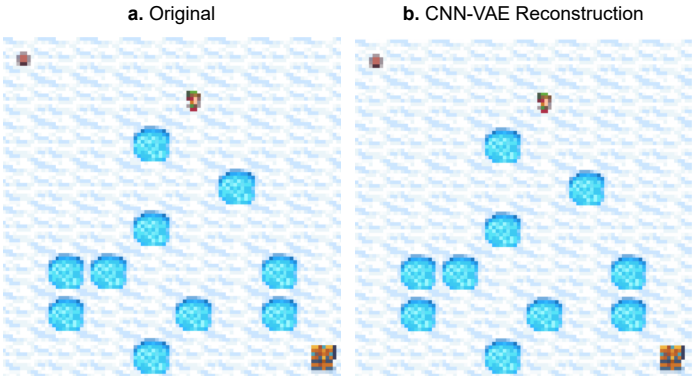

Fig. D.8: Frozen Lake CNN-VAE reconstruction example of a resized 96×96 pixel image.

Appendix E. Montezuma's Revenge Visual Examples

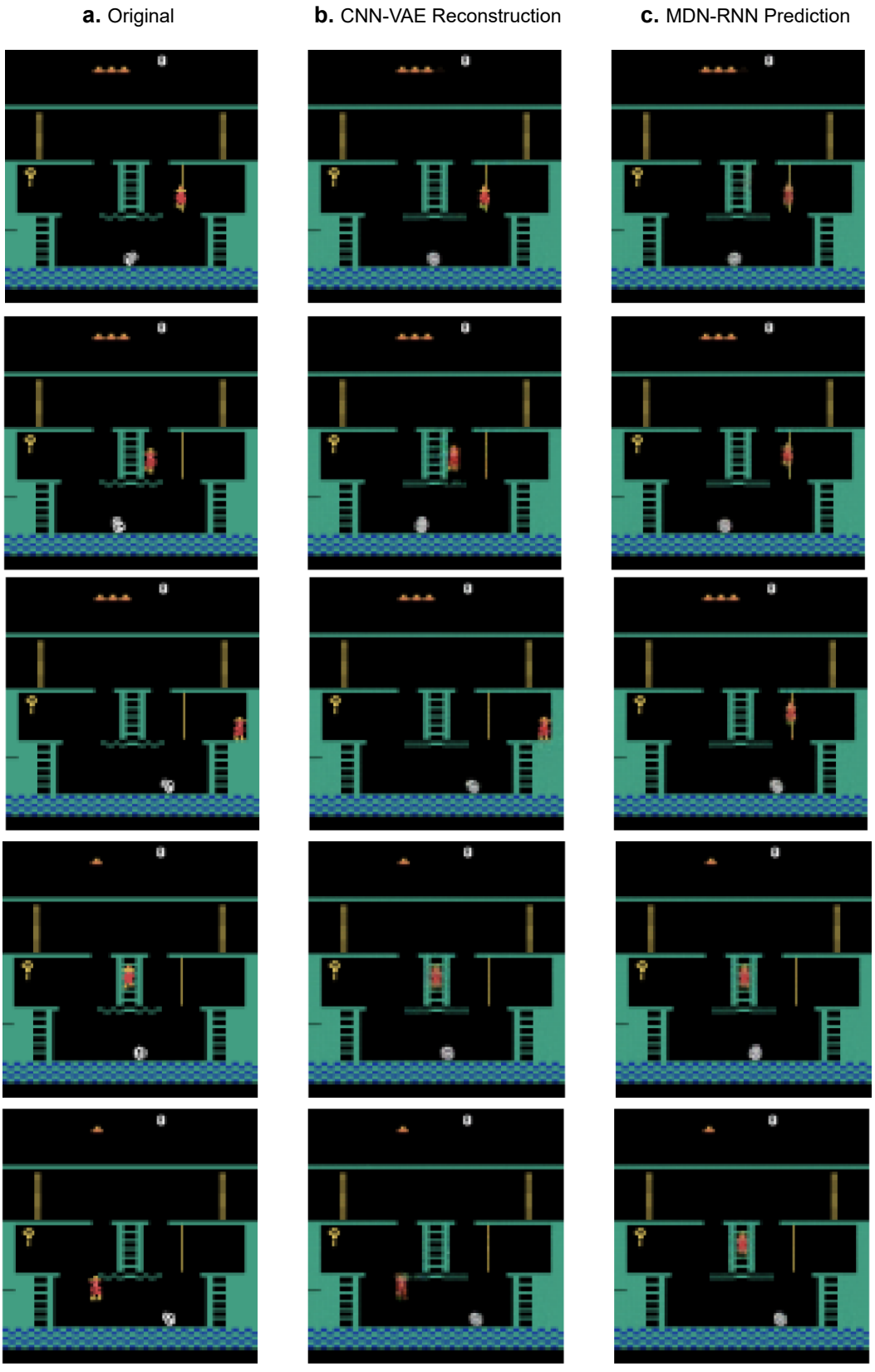

Fig. E.9: Five MR examples. All are 96×96 pixel images. Due to MR's relatively deterministic nature, the MDN-RNN predictions are generated using Eq. (18).