Learning Discrete World Models for Heuristic Search

Forest Agostinelli
foresta@cse.sc.edu
Dept. of Computer Science and Engineering
University of South Carolina

Misagh Soltani
msoltani@email.sc.edu
Dept. of Computer Science and Engineering
University of South Carolina

Abstract

For many sequential decision making problems, planning is often necessary to find solutions. However, for domains such as those encountered in robotics, the transition function, also known as the world model, is often unknown. While model-based reinforcement learning methods learn world models that can then be used for planning, such approaches are limited by errors that accumulate when the model is applied across many timesteps as well as the inability to re-identify states during planning. To solve these problems, we introduce DeepCubeAI, an algorithm that learns a world model that represents states in a discrete latent space, uses reinforcement learning to learn a heuristic function that generalizes over start and goal states using this learned model, and combines the learned model and learned heuristic function with heuristic search to solve problems. Since the latent space is discrete, we can prevent the accumulation of small errors by rounding and we can re-identify states by simply comparing two binary vectors. In our experiments on a pixel representation of the Rubik's cube, Sokoban, IceSlider, and DigitJump, we find that DeepCubeAI is able to apply the model for thousands of steps without accumulating any error. Furthermore, DeepCubeAI solves over 99% of test instances in all domains, generalizes across goal states, and significantly outperforms a greedy policy that does not plan with the learned world model.

1 Introduction

Planning requires a state-transition function, also known as a world model, that can accurately map states and actions to next states. While it is often convenient to manually construct a world model for environments with symbolic representations, this approach becomes impractical for environments with sub-symbolic representations, such as pixels. On the other hand, using machine learning techniques to learn a model from observed transitions offers the promise of a domain-independent approach to model construction. Reinforcement learning can then be used with these learned models to learn a policy or heuristic function without needing to collect any additional real-world data. Furthermore, at test time, the learned model can be used with search to improve performance. However, there are two major hindrances to this approach: 1) many learned models suffer from model degradation, thus rendering them ineffective for long-horizon planning; 2) many learned models do not have the ability to re-identify states during search, resulting in loops in the search-tree and, thus, inefficient planning.

Model degradation happens when small errors in the model's prediction accumulate over timesteps, resulting in decreasingly reliable predictions over long horizons. Model degradation has been observed in domains such as the Atari Learning Environment (Oh et al., 2015), Sokoban (Racanière et al., 2017), and robot manipulation tasks (Finn et al., 2016). Since this limits the usage of learned models to short horizon tasks, if such a learned model is used to learn a heuristic function, the agent will be limited to exploring states close to states observed in the real-world, which can lead to poor

generalization. While this can be remedied by more real-world exploration, real-world exploration is often many times more time-consuming than using a learned model that simulates the real-world. When planning with a model that degrades, only states that are relatively close to the starting state will be able to be considered. This can result in poor plans and the need for frequent re-planning (Tian et al., 2021). State re-identification is the ability to know when two latent embeddings represent the same state. This is crucial to planning because, without state re-identification, the same state will be visited multiple times during the search process. In the worst case, this leads to an exponential increase in computation time and memory as the depth of the search tree increases.

To address these problems, we will learn a mapping from states to a discrete latent space and learn a model that captures state transitions in this discrete latent space. This will allow us to combat model degradation because errors that are less than 0.5 can be readily fixed by rounding. This will allow the model to be used across thousands of timesteps without accumulating any errors. Furthermore, this discrete representation makes state re-identification a simple comparison between two binary vectors. Once the model is learned, a heuristic function represented by a deep neural network (DNN) (Schmidhuber, 2015), namely a deep Q-network (DQN) (Mnih et al., 2015), will be learned using Q-learning (Watkins & Dayan, 1992; Sutton & Barto, 2018). Since the goals that will be specified at test time are not assumed to be known beforehand, the heuristic function will be trained with a method inspired by hindsight experience replay (Andrychowicz et al., 2017) to allow it to generalize over goals. This results in a domain-independent algorithm for training domain-specific heuristic functions that generalize across problem instances. This heuristic function will then be used with Q* search (Agostinelli et al., 2024b), a variant of A* search (Hart et al., 1968) for DQNs, to solve problems. Since this method builds on the DeepCubeA algorithm (Agostinelli et al., 2019), which combines deep reinforcement learning and search to solve classical planning problems, we will call our method DeepCubeA-Imagination (DeepCubeAI), where imagination is in reference to the ability to use a learned model to "imagine" future scenarios (Racanière et al., 2017).

2 Related Work

Model-based reinforcement learning (RL) methods seek to leverage learned models to reduce the amount of real-world training data needed to learn a policy or value function as well as to do policy improvement at test time. One of the earliest instances of this is the Dyna architecture (Sutton, 1991). The Dyna architecture approach, which is similar to many approaches today, is to use observed transitions to train a model that can be subsequently used for learning and planning. Although strong results were demonstrated in the tabular setting, reliable results in large state spaces that cannot be represented by tables were not obtained and remain elusive to this day. An example of a modern model-based RL approach is Model-Based RL with Offline Learned Distance (MBOLD) (Tian et al., 2021). MBOLD presents an approach for using offline data to train a model to predict the pixels of the next state. It uses this offline data and model to train a heuristic function to estimate the cost-to-go. However, the model operates in a continuous latent space and accumulates error. Therefore, it is limited in how training data for the heuristic function is generated, cannot plan until the goal is reached, and does not re-identify states.

Work conducted by Bagatella et al. (2021) introduces a method named Planning from Pixels through Graph Search (PPGS), which learns to represent the states in a continuous latent space. State re-identification is done by comparing the distance between two vectors and setting a threshold for re-identification. By leveraging state re-identification, they create a latent graph and deploy graph search algorithms to solve classical planning problems. This architecture incorporates an encoder, a forward model, and an inverse model, the latter of which is employed to ensure the latent states contain relevant information that the model will need to use. Subsequently, they introduce two new environments, IceSlider and DigitJump, with an underlying combinatorial structure in which they verify the superior performance of PPGS in comparison to model-free methods, such as PPO (Schulman et al., 2017). However, the learned model accumulates errors and requires re-planning when the predicted latent states do not match what is observed. Furthermore, PPGS does not learn

a heuristic function, so it relies on breadth-first search to solve problems, which will not scale to more complex problems.

DreamerV3 (Hafner et al., 2023) uses a Recurrent State-Space Model (RSSM) (Hafner et al., 2019) to model states in a discrete latent space. They use this model and actor-critic methods to train a policy function. DreamerV3 is able to collect diamonds in Minecraft from scratch without human data. However, DreamerV3 only uses the learned model for training and not for planning at test time; as a result, it has not shown the ability to plan until a goal is reached or to re-identify states.

Instead of learning black-box models that operates in a latent space that is not readily understood by humans, research has been done on learning models that can be explicitly represented in Planning Domain Definition Language (PDDL) format (Asai & Fukunaga, 2018; Asai et al., 2022). Given such a representation, domain-independent planners can be employed to solve problems. However, these domain-independent planners may often fail when solving problems such as the Rubik's cube (Muppasani et al., 2023; Agostinelli et al., 2024a). Furthermore, since learning a PDDL model from data is not always feasible, a learned black-box model and domain-independent heuristics that work with black-box models, such as the goal-count heuristic, may have to be used.

3 Preliminaries

In this work, we are designing an algorithm capable of learning a discrete world model in deterministic, fully-observable domains. A domain, D, can be represented as a deterministic un-discounted Markov decision process (MDP) (Puterman, 2014), which is a tuple $< \mathcal{S}, \mathcal{A}, T, G >$, where $\mathcal{S}$ is the set of all states, $\mathcal{A}$ is the set of all actions, and T is the state-transition function that maps states and actions to next states, and G, the transition cost function that maps states, actions, and next states to a transition cost. It can also be represented as a weighted directed graph (Pohl, 1970) whose nodes represent states, edges represent transition between states, and edge weights represent transition costs. Goals correspond to a set of states that are considered goal states. Given a start state, the objective is to then find a sequence of actions that transforms the start state into a goal state while attempting to minimize the path cost, which is the sum of transition costs. The state-transition function and the transition cost function comprise the world model. When the transition costs are uniform, then learning a model is simplified to just learning the state transition function.

4 Methods

4.1 Learning a Discrete World Model

We seek to learn a model, m, that represents the state-transition function, T, in some latent space. In this setting, we assume that all transition costs are one. Similar to Tian et al. (2021), we will learn a model from offline data collected from random exploration. This dataset will contain a set of tuples, (s, a, s'), of states, actions, and next states. An enconder will be trained to project a given state to a state in a latent space. The encoder will use a logistic function at its output layer which will be rounded to be either 0 or 1. A straight-through gradient estimator (Bengio et al., 2013) will be used during gradient descent to account for the fact that the derivative of a rounding function with respect to its input is zero. A decoder will then map the latent space back to the state space. A mean squared error will be used as the reconstruction error to encourage the output of the decoder to be as close to the input of the encoder as possible. This ensures that the encoding captures what is present in the state. The reconstruction error is shown

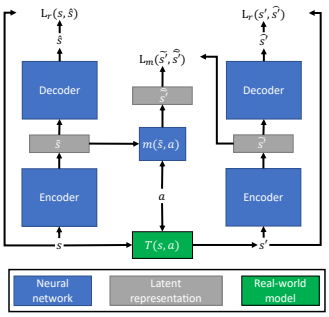


Figure 1: Overview for training the autoencoder and discrete world model.

in Equation 1 where N is the batch size, $\hat{s}$ is the output of the decoder, and θ are the parameters of the autoencoder and model.

$$L_r(\theta) = \frac{1}{N}\sum_{i=1}^{N}\frac{1}{2}||s_i - \hat{s}_i||_2^2 + \frac{1}{2}||s'_i - \hat{s}'_i||_2^2 \tag{1}$$

Simultaneously, a model will be trained to map latent states and actions to next latent states. A loss will be used to encourage the output of the model and the output of the encoder to be as close to each other as possible. In our experiments, we found that the best way to train the model together with the autoencoder was to encourage the output of the model to match the output of the encoder while simultaneously encouraging the output of the encoder to match the output of the model. However, we only round the output of the model when encouraging the output of the encoder to match the output of the model and the output of the encoder is always rounded. This is shown below in Equation 2, where $r()$ is the rounding function that uses a straight-through estimator during gradient descent and $.detach()$ removes the tensor from the computation graph.

$$L_m(\theta) = \frac{1}{N}\sum_{i=1}^{N}(\frac{1}{2}\|r(\tilde{s'_i}) - r(\hat{\tilde{s}}'_i).detach()\|_2^2 + \frac{1}{2}\|r(\tilde{s'_i}).detach() - \hat{\tilde{s}}'_i\|_2^2) \tag{2}$$

In our experiments, we observed that first training the autoencoder, then the model, resulted in an imperfect model, meaning that it was not able to predict the next latent state with 100% accuracy. Therefore, we saw the need to train the autoencoder and model together to ensure that the parameters of the autoencoder are encouraged to learn a representation that the model can also learn. However, the loss functions in Equation 1 and Equation 2 can conflict with one another because L_r does not consider the ability of the model to predict the latent state and L_m does not consider the reconstruction error. Therefore, we use a weight ω to first weight the L_r loss higher than L_m and gradually adjust ω to be 0.5 to weight them equally. The loss is shown in Equation 3.

$$L(\theta) = (1-\omega)L_r(\theta) + \omega L_m(\theta) \tag{3}$$

The training process is summarized in Figure 1. After training, every time the model is applied, a rounding operation is applied to its output to correct errors and prevent error accumulation.

4.2 Learning a Heuristic Function

Given a trained model and offline data, training data consisting of pairs of start and goal states can then be generated to train a heuristic function that generalizes over both start and goal states. For each training example, a real-world state is sampled from the offline data. The encoder is then used to obtain the corresponding latent state. A start state is then obtained by starting from the sampled latent state and using the model to randomly take t_s steps in the latent space, where t_s is uniform randomly distributed between 0 and T_s. A goal state is then obtained by starting from the start state and taking t_g steps, where t_g is randomly distributed between 0 and T_g. From this data, a DQN is trained with reinforcement learning to map start states and goal states to the cost-to-go of every action.

A DQN is a neural network that maps states to a vector of size $|\mathcal{A}|$, where each element at index a represents the expected cost-to-go when starting in a given state and taking action a, denoted as $Q(s,a)$. In the un-discounted deterministic setting, the estimate of $Q(s,a)$ is iteratively updated to be $G(s,a,s') + \min_{a'} Q(s',a')$. However, since Q is represented as a DQN with parameters ϕ, q_ϕ, bootstrapping from itself will lead to problems due to the non-stationary target. To address this, following previous work (Mnih et al., 2015), a target network with parameters ϕ^- is maintained

and periodically updated to be ϕ during training. The loss function used is $L(\phi) = (G(s, a, s') + \min_{a'} q_{\phi^-}(s', a', s_g) - q_\phi(s, a, s_g))^2$, where s_g is the goal state.

To select an action to update for Q-learning, we prioritize more promising actions over less promising actions because, in many environments, the majority of actions in a given state are not on a shortest path, resulting in bias. Therefore, we select actions according to a Boltzmann distribution where each action a is selected with probability according to Equation 4, where τ is the temperature.

$$\frac{e^{(-q_\phi(s,a,s_g)/\tau)}}{\sum_{a'=1}^{|\mathcal{A}|} e^{(-q_\phi(s,a',s_g)/\tau)}} \tag{4}$$

4.3 Planning with a Learned Model and Learned Heuristic Function

Given a learned model and heuristic function, planning can be done in the form of state-space search. While the DQN can be used with A* search by setting the heuristic function, $h(s, s_g)$, to $\min_{a'} q_\phi(s, a', s_g)$, A* search requires that the model be used $|\mathcal{A}|$ times per iteration. Given that the model is a computationally expensive DNN, we would like to minimize the number of times we use it. Therefore, we instead use Q* search (Agostinelli et al., 2024b), a modification of A* search that takes advantage of the fact that Q* search can compute the heuristic values for all next states with a single pass through a DQN. In practice, Q* search has been shown to perform similar to A* search while being orders of magnitude faster and more memory efficient. To take advantage of GPU parallelism and speed up search, we also use a batched and weighted (Pohl, 1970) version of Q* search as DeepCubeA did with A* search.

5 Experiments

We test our approach on the Rubik's cube, Sokoban, IceSlider, and DigitJump. For the Rubik's cube, states are represented by two 32 by 32 RGB images, where each image shows three faces of the Rubik's cube. For Sokoban, states are represented by one 40 by 40 RGB image showing the agent, walls, and boxes. IceSlider and DigitJump (introduced by Bagatella et al. (2021)), are represented as one 64 by 64 RGB image representing a two dimensional 8 by 8 grid. In IceSlider, the agent must slide across the ice, where only a rock or environment boundary stops its movement, to reach a given cell. In our work, we indicate the goal cell using the agent as an indicator instead of a purple square, as used in previous work. In DigitJump, the agent starts from the top left corner, and the goal is to reach the bottom right corner. The number of cells an agent will jump in a given cell is denoted by the MNIST (LeCun et al., 1998) digit in the given cell, where digits range from 1 to 6. Examples of states are shown in Figure 3.

In our experiments, we generate an offline dataset by observing transitions across episodes where, in each episode, the agent takes random actions[1]. For the Rubik's cube and Sokoban, we generate 300,000 transitions across 10,000 episodes, with 30 random actions taken in each episode. For IceSlider and DigitJump, we generate offline data in a similar manner to that of Bagatella et al. (2021). Specifically, we generate 400,000 transitions across 20,000 episodes, with 20 random actions taken in each episode. For the Rubik's cube, starting states for each episode are obtained by randomly scrambling the goal state between 100 and 200 times. For Sokoban, starting states for each episode are randomly sampled from training examples provided by Guez et al. (2018). For IceSlider and DigitJump, starting states for each episode are obtained from the same 1,000 levels used by Bagatella et al. (2021). For the Rubik's cube and Sokoban, 90% of the generated data is used for training the model and 10% is used for validation. For IceSlider and DigitJump, validation data is generated by repeating the procedure with 20 random actions across 5,000 episodes, using another set of 1,000 distinct levels, resulting in a total of 100,000 transitions. During training and search, two latent states are considered equal if 100% of the bits in the latent state are equal.

[1]Future work could use intrinsic motivation (Barto et al., 2004) to encourage the exploration of diverse states.

For the Rubik's cube, the autoencoder architecture is a fully connected neural network where both the encoder and decoder have one hidden layer and an encoding dimension of size 400. The encoder uses a logistic activation function while the decoder uses a linear activation function. Though the RGB values are bounded between 0 and 1, we found that a linear layer in the last layer of the decoder performed better. The discrete world model is a fully connected neural network with four layers of size 500, 500, 500, and 400. It uses batch normalization in all layers, excluding the last layer. Additionally, rectified linear units (ReLU) (Glorot et al., 2011) are utilized in all layers, except for the last layer, which uses a logistic activation function. The model uses a one-hot representation for the action which is concatenated with the latent state.

For Sokoban, the autoencoder architecture uses a convolutional encoder and decoder, both with two layers with 16 channels, a kernel size of 2, a stride of 2, and batch normalization in the first layer. The decoder uses an additional convolutional layer with a kernel size of 1 and a linear activation function. The discrete world model is a convolutional neural network with three layers with channel sizes of 32, 32, and 16, all with kernel sizes of 3, strides of 1, batch normalization in the first two layers, rectified linear units in the first two layers, and a logistic activation function in the last layer. The model represents the actions with a one-hot representation that is extended into a tensor with a length and width the size of the latent representation and number of channels that equals the number of actions. This is then concatenated with the latent state along the channel dimension.

For IceSlider, the autoencoder architecture is similar to that of Sokoban, utilizing a two-layer convolutional encoder with 32 channels in the first layer and 3 channels in the last layer. The decoder utilizes transposed convolutional layers with 32 channels. The convolutional layers have kernel sizes of 4 and 2, and strides of 4 and 2, respectively. Batch normalization and activation functions are the same as Sokoban. The decoder also includes an additional layer with a linear activation function. In the discrete world model, we follow Sokoban's concatenation process. Initially, a convolutional layer with a kernel size of 1 and a stride of 1, processes the input. This is then given to four residual blocks, maintaining the same number of channels as the input. The output from the last residual block is passed to a convolutional layer with a kernel size of 1 and a stride of 1. Finally, an additional layer with a kernel size of 1 serves as the output layer. Batch normalization and rectified linear units are applied in all layers except for the first layer, which uses layer normalization, and the last layer, which uses a logistic activation function without normalization. DigitJump shares the same architectural layers as IceSlider, with the encoder having an output of 12 channels, and the residual blocks utilizing 47 channels.

All models are trained with gradient descent with the ADAM optimizer (Kingma & Ba, 2014) with a learning rate of 0.001, a decay rate of 0.9999993, and a batch size of 100. ω is initialized to 0.0001 and is gradually shifted to 0.5 by iteration 120,000. The neural network is then trained until iteration 180,000 and the learning rate is reduced by a factor of 10 every 20,000 iterations.

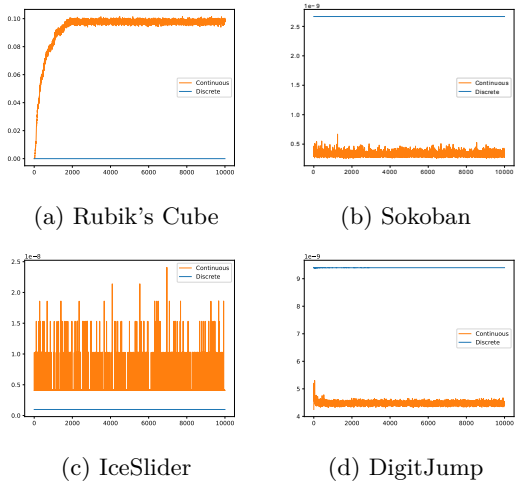
Q-learning is then used to train the heuristic function. To generate start and goal pairs, both T_s and T_g are set to be 30 for the Rubik's cube and Sokoban, and 20 for IceSlider and DigitJump. The heuristic function is trained with Q-learning with the ADAM optimizer, with a learning rate of 0.001, a decay rate of 0.9999993, and a batch size of 10,000 for 1 million iterations. Actions are selected according to Equation 4 with τ set to 3.0. To better explore the state space during learning, new states are also generated by behaving greedily with respect to the DQN for up to 30 steps for the Rubik's cube and Sokoban, and for up to 20 steps for IceSlider and DigitJump. The DQN is tested with a greedy policy every 5,000 iterations and the target network's parameters are updated if the greedy policy has improved.

The respective batch size and weight on the path cost for Q* is 10,000 and 0.6 for the Rubik's cube, 100 and 0.8 for Sokoban, and 1 and 0.7 for IceSlider and DigitJump. To specify a goal state, an image of the goal is given, encoded to the latent space, and given to the heuristic function. For Sokoban, goals are specified by an image of where the boxes should be with the agent randomly selected to be placed next to a box. We will discuss more robust goal specification in the Future

Work Section. We compare DeepCubeAI to DeepCubeA, as well as a domain-specific PDB that leverages group-theory knowledge (Rokicki, 2016; Rokicki et al., 2014).

5.1 Model Performance

To determine how well the model performs, we test it on 10 sequences of 10,000 steps where actions are selected randomly. We obtain ground-truth images for each step as well as take steps in the latent space and obtain reconstructions from the decoder. Furthermore, we compare to a continuous model that has the same architecture and training procedure as the discrete model, but without the discretization. Results from this comparison are shown in Figure 2. The figure shows that, while the continuous model does not accumulate errors for Sokoban, IceSlider and DigitJump, it accumulates a significant amount of error for the Rubik's cube. Figure 3a shows an example for the Rubik's cube where the continuous model makes significant errors but the discrete model does not. Figures 3b, 3c, and 3d show examples for Sokoban, IceSlider and DigitJump, where both the continuous and discrete models do not make significant errors. This may be attributed to theses environments being easier to reconstruct across many timesteps. In Sokoban, for instance, the boxes quickly get pushed up against walls and, therefore, become immovable thereafter, with only the location of the agent changing between transitions. Similarly, in IceSlider and DigitJump, actions only affect the agent's position and there are states for which the agent cannot move, such as in Figure 3d.



(a) Rubik's Cube (b) Sokoban

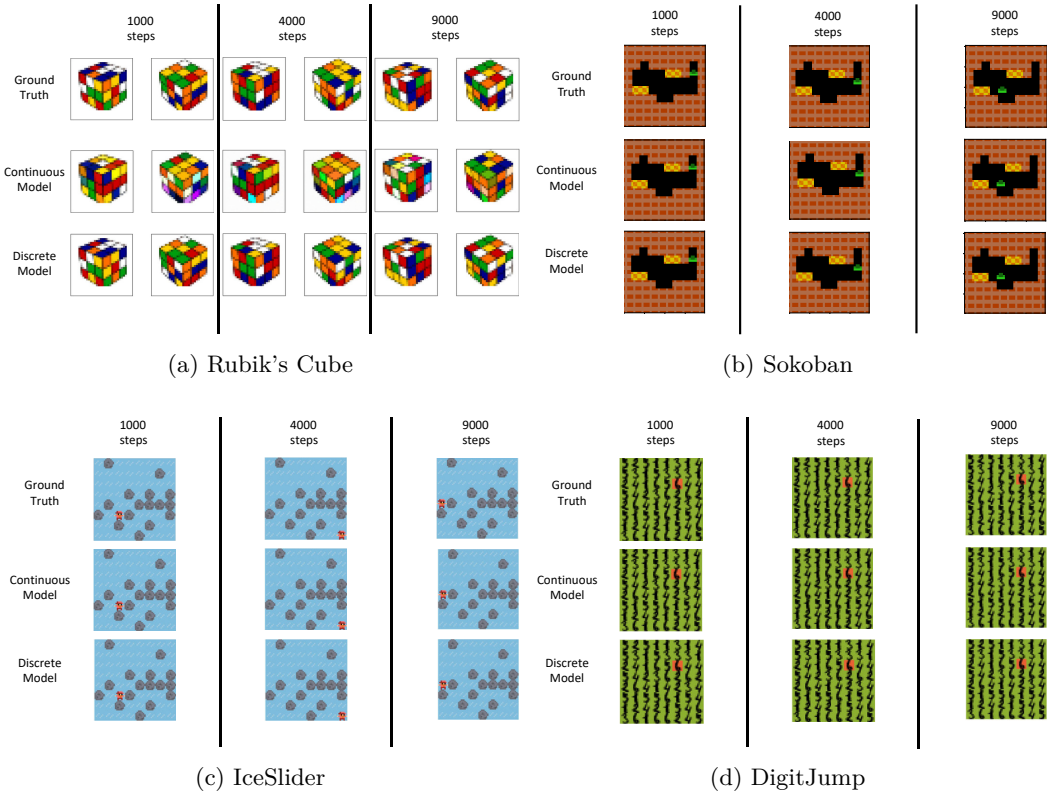
(c) IceSlider (d) DigitJump

Figure 2: Mean squared reconstruction error as a function of timestep. For the Rubik's cube, the continuous model accumulates error over time while, for Sokoban, IceSlider, and DigitJump, neither model accumulates error.

5.2 Problem Solving Performance

We evaluate DeepCubeAI on 1,000 test instances for the Rubik's cube and Sokoban obtained from the DeepCubeA repository (Agostinelli et al., 2020), and on the 100 test instances for each of IceSlider and DigitJump used to evaluate PPGS (Bagatella et al., 2021). To determine the importance of planning when solving these test instances, we also use them to evaluate a greedy policy obtained by behaving greedily with respect to the trained DQN for 100 steps. To test DeepCubeAI's ability to generalize to new goal states, we include a test set where the start and goal state are reversed for the Rubik's cube. As a result, each test instance has a different goal state. We note that DeepCubeAI was not told of the test goal states during training.

A detailed comparison of DeepCubeAI to DeepCubeA, PDBs, and the greedy policy is shown in Table 1. The results show that DeepCubeAI solved 100% of all test instances for the Rubik's cube with the canonical goal state as well as for Sokoban, IceSlider, and DigitJump. For the reversed start and goal states, DeepCubeAI solved 99.9% (only missing 1) of all test cases and had similar performance to the canonical goal test instances. We note that DeepCubeA and PDBs cannot be readily applied to this test set because they are specific to the canonical goal state. To apply DeepCubeA to different goal states, we would have to train a new DNN for each of the 1,000 goal states. The greedy policy does not solve any problem instances for the Rubik's cube and solves 41.9%, 46.0%, and 90.0% of problem instances for Sokoban, IceSlider, and DigitJump, respectively. This shows that planning with a learned world model is crucial to solving these problems. DeepCubeAI has a longer path cost than DeepCubeA. This could be because DeepCubeAI is learning a more complex heuristic function



(a) Rubik's Cube (b) Sokoban

(c) IceSlider (d) DigitJump

Figure 3: A visualization of the reconstructions for models with continuous and discrete latent states at different timesteps. For the Rubik's cube, the discrete model accurately represents the ground truth while the continuous model makes errors. For Sokoban, IceSlider, and DigitJump both the discrete and continuous models accurately reconstruct the ground truth image after thousands of timesteps.

that generalizes over goals, while DeepCubeA is trained for a pre-determined goal. However, for the Rubik's cube, despite processing fewer nodes a second due to the fact the learned model is more computationally expensive than a hand-coded model, DeepCubeAI generates fewer nodes and takes less time when finding solutions. This may be partially due to the speedup provided by Q* search. However, for Sokoban, we found that a batch size of 100 for DeepCubeAI was necessary when performing Q* search, while DeepCubeA used a batch size of 1 for A* search, so the number of nodes generated for DeepCubeAI is still larger than DeepCubeA.

6 Future Work

In the one case where DeepCubeAI was not able to find a path, we saw that it was not able to correctly identify the latent goal state. This could be that an error of greater than 0.5 was made by the model during search, meaning rounding was unable to correct it. Future work could address these rare mistakes by training a DNN to correct slightly corrupted latent states.

Similar to research in model-based reinforcement learning (Tian et al., 2021), we specify goals with a goal image. While this may be feasible for some environments, this becomes impractical in environments where goal images are difficult to generate. Furthermore, if one only knows high-level information about a goal without knowing the low-level details, a goal image will be impossible to generate. To solve this, research has been done to use formal logic to specify goals, where a goal can be a set of states (Agostinelli et al., 2024a). This approach can be extended to learned models and allow one to specify goals without having to generate any goal images.

Domain	Solver	Len	Opt	Nodes	Secs	Nodes/Sec	Solved
RC	PDBs$^+$	20.67	**100.0%**	2.05E+06	2.20	1.79E+06	100%
	DeepCubeA	21.50	60.3%	6.62E+06	24.22	2.90E+05	100%
	Greedy	-	0%	-	-	-	0%
	DeepCubeAI	22.85	19.5%	2.00E+05	6.21	3.22E+04	**100%**
RC_{rev}	Greedy	-	0%	-	-	-	0%
	DeepCubeAI	22.81	**21.92%**	2.00E+05	6.30	3.18+04	**99.9%**
Sokoban	LevinTS	39.80	-	6.60E+03	-	-	100%
	LevinTS (*)	39.50	-	5.03E+03	-	-	100%
	LAMA	51.60	-	3.15E+03	-	-	100%
	DeepCubeA	32.88	-	1.05E+03	2.35	5.60E+01	100%
	Greedy	29.55	-	-	1.68	-	41.9%
	DeepCubeAI	33.12	-	3.30E+03	2.62	1.38E+03	**100%**
IceSlider	PPGS	-	-	-	-	-	97.0%
	Greedy	9.83	84.78%	-	0.03	-	46.0%
	DeepCubeAI	9.85	**100%**	31.84	0.09	3.50E+02	**100%**
DigitJump	PPGS	-	-	-	-	-	99.0%
	Greedy	5.72	88.89%	-	0.04	-	90.0%
	DeepCubeAI	5.83	**96.0%**	8.97	0.06	1.40E+02	**100%**

Table 1: Comparison of DeepCubeAI (ours) with a greedy policy (ours), DeepCubeA, and PDBs along the dimension of solution length, percentage of optimal solutions, number of nodes generated, time taken to solve the problem (in seconds), number of nodes generated per second, and percentage solved. RC is the Rubik's cube and RC_{rev} is the Rubik's cube with the start and goal states reversed. Note that DeepCubeA cannot be applied to RC_{rev} since it is only trained on the canonical goal state. PDBs$^+$ refers to domain-specific PDBs for the Rubik's cube that leverage knowledge of group theory (Rokicki, 2016; Rokicki et al., 2014), DeepCubeA refers to work by Agostinelli et al. (2019), LevinTS and LAMA refer to work by Orseau et al. (2018), PPGS refers to work by Bagatella et al. (2021).

For certain robotic manipulation tasks, given enough sensors and enough experience in the environment, the domain can be thought of as deterministic and fully-observable. However, many tasks in robotics are stochastic due inherit characteristics of the domain or lack of knowledge of the environment dynamics and partially observable due to limited sensing. Research has been done on learning models in stochastic environments by training DNNs to sample possible next states (Kaiser et al., 2020; Hafner et al., 2021). Sequence models, such as recurrent neural networks (Hochreiter & Schmidhuber, 1997), have been used to learn to embed belief states (Hausknecht & Stone, 2015; Cassandra et al., 1994) on which we can plan. The benefits of discrete models could extend to these domains, as well, allowing for the model to be applied over long horizons to improve exploration for training and to obtain more lookahead during search.

7 Conclusion

We introduce DeepCubeAI, a domain-independent method for learning a model that operates on discrete latent states. This learned model is then used to learn a heuristic function that generalizes over problem instances. The learned model and learned heuristic function are then combined with search to solve problems. In the case of the Rubik's cube, results show that having a discrete model is crucial to preventing error accumulation. In the case of all the Rubik's cube, Sokoban, IceSlider, and DigitJump, results show that DeepCubeAI solves over 99% of test cases and effectively generalizes across goal states.

References

Forest Agostinelli, Stephen McAleer, Alexander Shmakov, and Pierre Baldi. Solving the Rubik's cube with deep reinforcement learning and search. *Nature Machine Intelligence*, 1(8):356–363, 2019.

Forest Agostinelli, Stephen McAleer, Alexander Shmakov, and Pierre Baldi. DeepcubeA. https://github.com/forestagostinelli/DeepCubeA, 2020.

Forest Agostinelli, Rojina Panta, and Vedant Khandelwal. Specifying goals to deep neural networks with answer set programming. In *Proceedings of the International Conference on Automated Planning and Scheduling*, volume 34, pp. 2–10, 2024a.

Forest Agostinelli, Shahaf S Shperberg, Alexander Shmakov, Stephen McAleer, Roy Fox, and Pierre Baldi. Q* search: Heuristic search with deep q-networks. In *ICAPS Workshop on Bridging the Gap between AI Planning and Reinforcement Learning*, 2024b.

Marcin Andrychowicz, Filip Wolski, Alex Ray, Jonas Schneider, Rachel Fong, Peter Welinder, Bob McGrew, Josh Tobin, OpenAI Pieter Abbeel, and Wojciech Zaremba. Hindsight experience replay. In *Advances in Neural Information Processing Systems*, pp. 5048–5058, 2017.

Masataro Asai and Alex Fukunaga. Classical planning in deep latent space: Bridging the subsymbolic-symbolic boundary. In *Proceedings of the aaai conference on artificial intelligence*, volume 32, 2018.

Masataro Asai, Hiroshi Kajino, Alex Fukunaga, and Christian Muise. Classical planning in deep latent space. *Journal of Artificial Intelligence Research*, 74:1599–1686, 2022.

Marco Bagatella, Miroslav Olšák, Michal Rolínek, and Georg Martius. Planning from pixels in environments with combinatorially hard search spaces. *Advances in Neural Information Processing Systems*, 34:24707–24718, 2021.

Andrew G Barto, Satinder Singh, Nuttapong Chentanez, et al. Intrinsically motivated learning of hierarchical collections of skills. In *Proceedings of the 3rd International Conference on Development and Learning*, volume 112, pp. 19. Citeseer, 2004.

Yoshua Bengio, Nicholas Léonard, and Aaron Courville. Estimating or propagating gradients through stochastic neurons for conditional computation. *arXiv preprint arXiv:1308.3432*, 2013.

Anthony R Cassandra, Leslie Pack Kaelbling, and Michael L Littman. Acting optimally in partially observable stochastic domains. In *AAAI*, volume 94, pp. 1023–1028, 1994.

Chelsea Finn, Ian Goodfellow, and Sergey Levine. Unsupervised learning for physical interaction through video prediction. *Advances in neural information processing systems*, 29, 2016.

Xavier Glorot, Antoine Bordes, and Yoshua Bengio. Deep sparse rectifier neural networks. In *Proceedings of the fourteenth international conference on artificial intelligence and statistics*, pp. 315–323, 2011.

Arthur Guez, Mehdi Mirza, Karol Gregor, Rishabh Kabra, Sebastien Racaniere, Theophane Weber, David Raposo, Adam Santoro, Laurent Orseau, Tom Eccles, Greg Wayne, David Silver, Timothy Lillicrap, and Victor Valdes. An investigation of model-free planning: boxoban levels. https://github.com/deepmind/boxoban-levels/, 2018.

Danijar Hafner, Timothy Lillicrap, Ian Fischer, Ruben Villegas, David Ha, Honglak Lee, and James Davidson. Learning latent dynamics for planning from pixels. In *International conference on machine learning*, pp. 2555–2565. PMLR, 2019.

Danijar Hafner, Timothy Lillicrap, Mohammad Norouzi, and Jimmy Ba. Mastering atari with discrete world models. 2021.

Danijar Hafner, Jurgis Pasukonis, Jimmy Ba, and Timothy Lillicrap. Mastering diverse domains through world models. *arXiv preprint arXiv:2301.04104*, 2023.

Peter E Hart, Nils J Nilsson, and Bertram Raphael. A formal basis for the heuristic determination of minimum cost paths. *IEEE transactions on Systems Science and Cybernetics*, 4(2):100–107, 1968.

Matthew Hausknecht and Peter Stone. Deep recurrent q-learning for partially observable mdps. In *2015 AAAI fall symposium series*, 2015.

Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural computation*, 9(8): 1735–1780, 1997.

Lukasz Kaiser, Mohammad Babaeizadeh, Piotr Milos, Blazej Osinski, Roy H Campbell, Konrad Czechowski, Dumitru Erhan, Chelsea Finn, Piotr Kozakowski, Sergey Levine, et al. Model-based reinforcement learning for Atari. *International Conference on Learning Representations*, 2020.

Diederik P Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

Yann LeCun, Léon Bottou, Yoshua Bengio, and Patrick Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324, 1998.

Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A Rusu, Joel Veness, Marc G Bellemare, Alex Graves, Martin Riedmiller, Andreas K Fidjeland, Georg Ostrovski, et al. Human-level control through deep reinforcement learning. *Nature*, 518(7540):529–533, 2015.

Bharath Muppasani, Vishal Pallagani, Biplav Srivastava, and Forest Agostinelli. On solving the rubik's cube with domain-independent planners using standard representations. *arXiv preprint arXiv:2307.13552*, 2023.

Junhyuk Oh, Xiaoxiao Guo, Honglak Lee, Richard L Lewis, and Satinder Singh. Action-conditional video prediction using deep networks in atari games. *Advances in neural information processing systems*, 28, 2015.

Laurent Orseau, Levi Lelis, Tor Lattimore, and Théophane Weber. Single-agent policy tree search with guarantees. In *Advances in Neural Information Processing Systems*, pp. 3201–3211, 2018.

Ira Pohl. Heuristic search viewed as path finding in a graph. *Artificial intelligence*, 1(3-4):193–204, 1970.

Martin L Puterman. *Markov decision processes: discrete stochastic dynamic programming*. John Wiley & Sons, 2014.

Sébastien Racanière, Théophane Weber, David Reichert, Lars Buesing, Arthur Guez, Danilo Jimenez Rezende, Adria Puigdomenech Badia, Oriol Vinyals, Nicolas Heess, Yujia Li, et al. Imagination-augmented agents for deep reinforcement learning. In *Advances in neural information processing systems*, pp. 5690–5701, 2017.

Tomas Rokicki. cube20. https://github.com/rokicki/cube20src, 2016.

Tomas Rokicki, Herbert Kociemba, Morley Davidson, and John Dethridge. The diameter of the Rubik's cube group is twenty. *SIAM Review*, 56(4):645–670, 2014.

Jürgen Schmidhuber. Deep learning in neural networks: An overview. *Neural networks*, 61:85–117, 2015.

John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. Proximal policy optimization algorithms. *arXiv preprint arXiv:1707.06347*, 2017.

Richard S Sutton. Dyna, an integrated architecture for learning, planning, and reacting. *ACM Sigart Bulletin*, 2(4):160–163, 1991.

Richard S Sutton and Andrew G Barto. *Reinforcement learning: An introduction*. MIT press, 2018.

Stephen Tian, Suraj Nair, Frederik Ebert, Sudeep Dasari, Benjamin Eysenbach, Chelsea Finn, and Sergey Levine. Model-based visual planning with self-supervised functional distances. In *International Conference on Learning Representations*, 2021.

Christopher JCH Watkins and Peter Dayan. Q-learning. *Machine learning*, 8(3-4):279–292, 1992.