Learning Discrete World Models for Classical Planning Problems

Forest Agostinelli
Dept. of Computer Science and Engineering
University of South Carolina
Columbia, SC 29205
foresta@cse.sc.edu

Misagh Soltani
Dept. of Computer Science and Engineering
University of South Carolina
Columbia, SC 29205
msoltani@email.sc.edu

Abstract

For many sequential decision making domains, planning is often necessary to solve problems. However, for domains such as those encountered in robotics, the transition function, also known as the world model, is often unknown and coding such a model by hand is often impractical. While planning could be done with a world model trained from observed transitions, such approaches are limited by errors accumulating when the model is applied across many timesteps as well as the inability to re-identify states. Furthermore, even given an accurate world model, domain-independent planning methods may not be able to reliably solve problems while domain-specific information required to construct informative heuristics may not be readily available. While methods exist that can learn domain-specific heuristic functions in a largely domain-independent fashion, such as DeepCubeA, these methods assume a given world model and may also assume that the goal is predetermined. To solve these problems, we introduce DeepCubeAI, a domain-independent algorithm that learns a world model that represents states in a discrete latent space, learns a heuristic function that generalizes over start and goal states using this learned model, and combines the learned model and learned heuristic function with 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 and Sokoban, 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 and generalizes across goal states.

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. These models can then be integrated with PDDL and off-the-shelf heuristic functions to take advantage of domain-independent planning algorithms. However, there are three major hindrances to these approaches: 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; 3) even if an accurate model is obtained, domain-independent approaches may not be sufficient to reliably solve problems and methods to learn domain-specific heuristic functions may not generalize over goal states.

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 [29], Sokoban [34], and robot manipulation tasks [15]. 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. [41]. 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.

Given an accurate model, one may then plan with this model to solve problems. If one can learn a model that explicitly represents the planning domain, such as in PDDL format, then domain-independent planning algorithms, such as the Fast Downward planning system [23], can be used. However, this may not always be possible in practice [7, 8], resulting in the need to construct a black-box model. Since the black-box model cannot be used with planners that assume formal representations, such as PDDL, search algorithms that operate on the level of states, such as A* search [21], can be used to solve problems. However, in the absence of domain-specific information, domain-independent heuristics, such as the goal count heuristic, will need to be employed, which may not be informative, in practice. Furthermore, even if a PDDL representation can be acquired, domain-independent planners still may not be able to reliably solve problems. This has been observed to be the case for problems such as the Rubik's cube when using the Fast Downward planner [28]. Also, domain-independent methods for constructing pattern databases [14] for the Rubik's cube (i.e. those that do not contain group theory knowledge), have been shown to consume more memory and time when compared to DNN heuristic functions trained with reinforcement learning, such as DeepCubeA [1]. However, DeepCubeA is trained for a predetermined goal state, and, thus, cannot generalize to new goal states without re-training.

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) [37], namely a deep Q-network (DQN) [27], will be learned using Q-learning [43, 40]. 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 [5] 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 [4], a variant of A* search [21] for DQNs, to solve problems. Since this method builds on the DeepCubeA algorithm [1], 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 [34].

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 [39]. 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) [41]. 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.

The work conducted by [9] 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 environments with an underlying combinatorial structure in which they verify the superior performance of PPGS in comparison to model-free methods, such as PPO [38]. 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 [20] uses a Recurrent State-Space Model (RSSM) [18] 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 PDDL format [7, 8]. 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 [28]. Furthermore, since learning a PDDL model from data is not always feasible, in practice, domain-independent approaches can still be used with a black-box model and a domain-independent heuristic, such as the goal-count heuristic.

3 Preliminaries

3.1 Classical Planning

Classical planning is a sub-field of automated planning [16] that focuses on deterministic, fully-observable domains. A classical planning domain, D, can be represented as a deterministic un-discounted Markov decision process (MDP) [32], 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 [31] 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. Given a world model, many problems in robotics, such as object manipulation, can be posed as classical planning problems. These problems can benefit greatly from planning techniques due to the fact that, in many cases, the underlying symbolic problem can be easily solved with planning techniques. For example, block stacking, a well studied problem in planning, is still difficult to solve using model-free RL techniques [41].

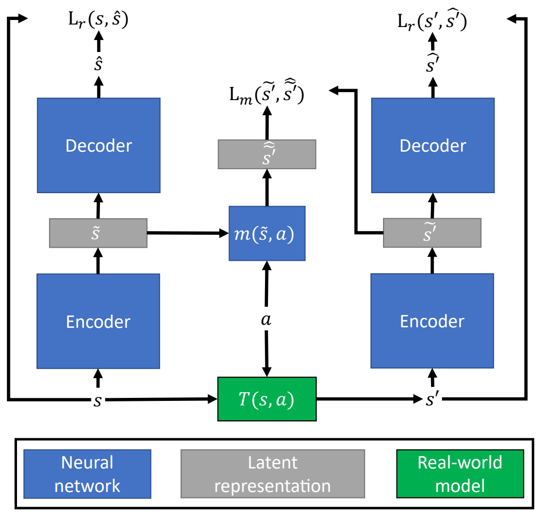


Figure 1: Overview for training the autoencoder and discrete world model.

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. 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 any 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 [11] 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 reconstruction error will be used 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 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 in Equation 1 and Equation 2 are in conflict with one another because the model only cares about ensuring the latent state is predictable, regardless of its relevance to the true state and the autoencoder only cares about the relevance of the latent state to the true state. Therefore, we use a weight ω to first weight the reconstruction loss higher than the model loss 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 latent state. A start state is then obtained by 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 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 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, that are smaller than the size of the state space and not as a table, bootstrapping from itself will lead to problems due to the non-stationary target. To address this, following previous work [27], a target network with parameters ϕ^- is maintained and periodically updated to be ϕ during training. The loss function used is shown in Equation 4, where s_g is the goal state.

$$L(\phi) = (G(s,a,s') + \min_{a'} q_{\phi^-}(s',a',s_g) - q_\phi(s,a,s_g))^2 \tag{4}$$

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 any 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 5, where τ is the temperature.

$$\frac{e^{(-q_\phi(s,a,s_g)/\tau)}}{\sum_{a'=1}^{|\mathcal{A}|} e^{(-q_\phi(s,a',s_g)/\tau)}} \tag{5}$$

Unlike approximate value iteration [33, 12], which needs to apply all actions to a given state in order to compute an update, Q-learning only needs to apply the one action to compute an update. This saves a significant amount of time when the model is a computationally expensive DNN. When learning heuristic functions, Q-learning has been shown to lead to faster training times with a slight decrease in accuracy when compared to approximate value iteration [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 $\min_{a'} q_\phi(s,a',s_g)$, like approximate value iteration, A* search requires that the model be used $|\mathcal{A}|$ times per iteration. Therefore, we instead use Q* search [4], 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 [31] version of Q* search as DeepCubeA did with A* search.

5 Experiments

We test our approach on the Rubik's cube and Sokoban. 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. Examples of states are shown in Figure 2.

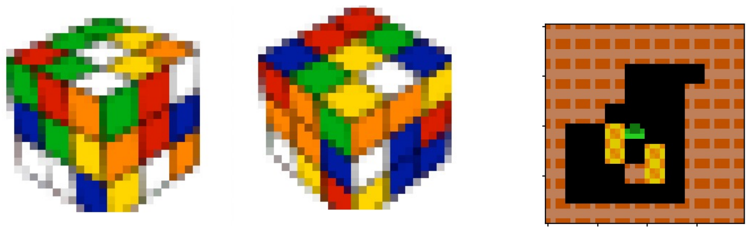

(a) Rubik's Cube (b) Sokoban

Figure 2: Visualization of states.

In our experiments, we generate an offline dataset of 300,000 examples by observing transitions across 10,000 episodes where, in each episode, the agent takes 30 random actions[1]. 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.[6]. 90% of the data is used for training the model and 10% is used for validation. During training and search, two latent states are considered equal if 100% of the bits in the latent state are equal.

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 sigmoid 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 model is as 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) [17] are utilized in all layers, except for the last layer, which uses a sigmoid 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 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 sigmoid activation function in the last layer. The model uses a one-hot representation that is extended into a tensor with a length and width the size of the latent representation (10 by 10) and a third dimension the size of the number of actions (4). This is then concatenated with the latent state.

All models are trained with gradient descent with the ADAM optimizer [26] 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. 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, a batch size of 10,000, and for 1 million iterations. Actions are selected according to Equation 5 with τ set to 3.0. To better explore the state space during

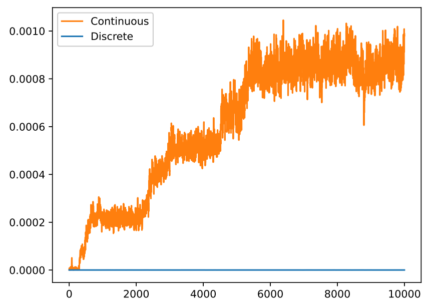
[1]Future work could use intrinsic motivation [10] to encourage the exploration of diverse states.

learning, new states are also generated by behaving greedily with respect to the DQNs policy for up to 30 steps. 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.

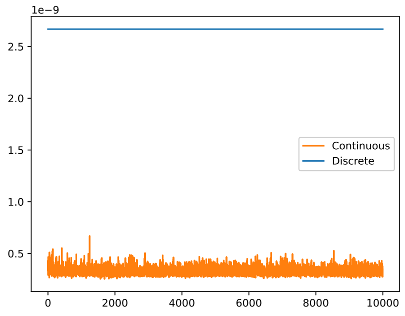
The batch size and weight on the path cost for Q* is 10,000 and 0.6, respectively, for the Rubik's cube. These are the same parameters used by DeepCubeA. For Sokoban, we found it necessary to use a batch size of 100 and weight of 0.1, while DeepCubeA used 1 and 0.8. To specify the goal state, an image 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 Section 6. We compare DeepCubeAI to DeepCubeA, as well as a domain-specific PDB that leverages group-theory knowledge [35, 36].

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 3. The figure shows that, while the continuous model does not accumulate errors for Sokoban, it accumulates a significant amount of error for the Rubik's cube. Figure 4 shows an example for the Rubik's cube where the continuous model makes significant errors but the discrete model does not. Figure 5 shows an example for Sokoban where both the continuous and discrete models do not make significant errors. It could be that Sokoban is easier to reconstruct across many timesteps because the boxes quickly get pushed up against walls and, therefore, become immovable. After this, only the location of the agent changes between transitions.



(a) Rubik's Cube



(b) Sokoban

Figure 3: Mean squared reconstruction error (MSE) as a function of timestep. For the Rubik's cube, the continuous model accumulates error over time. However, for Sokoban, while the continuous model starts with a lower MSE, neither model accumulates error.

5.2 Problem Solving Performance

We test DeepCubeAI on 1,000 test instances for the Rubik's cube and 1,000 test instances for Sokoban obtained from the DeepCubeA repository [2]. The Rubik's cube test instances were obtained by scrambling each cube between 1,000 and 10,000 times and Sokoban test instances come from Guez et al.[6]. DeepCubeAI solved 100% of all test instances. We note that DeepCubeAI was not told of the goal states during training and it is quite possible that, during training it never saw any of the goal states given to it at test time. 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. In this case, DeepCubeAI solved 99.9% (only missing 1) of all test cases and had similar performance to when the canonical goal state was given.

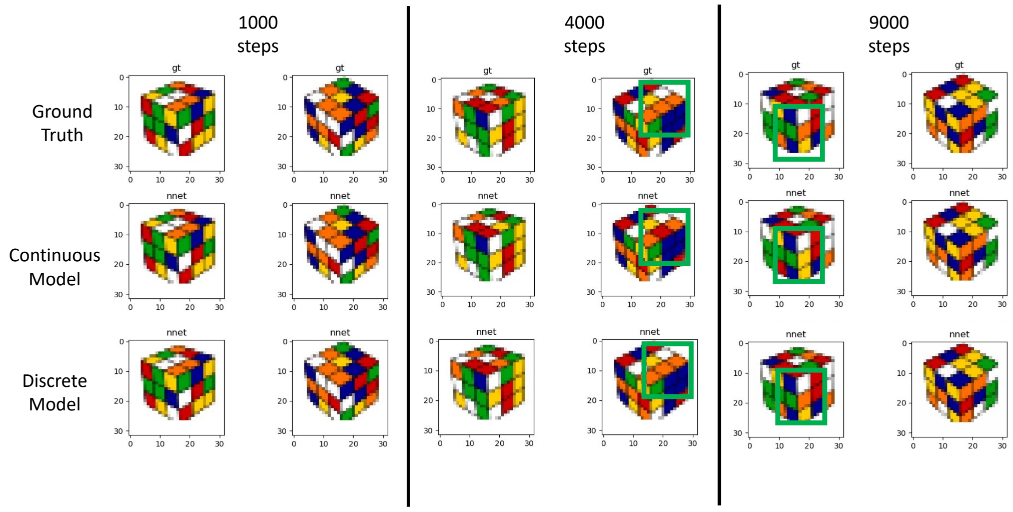


Figure 4: A visualization of the reconstructions for models with continuous and discrete latent states at different timesteps for the Rubik's cube. The discrete model accurately represents the ground truth while the continuous model makes errors. Errors made by the continuous model are highlighted in the green box.

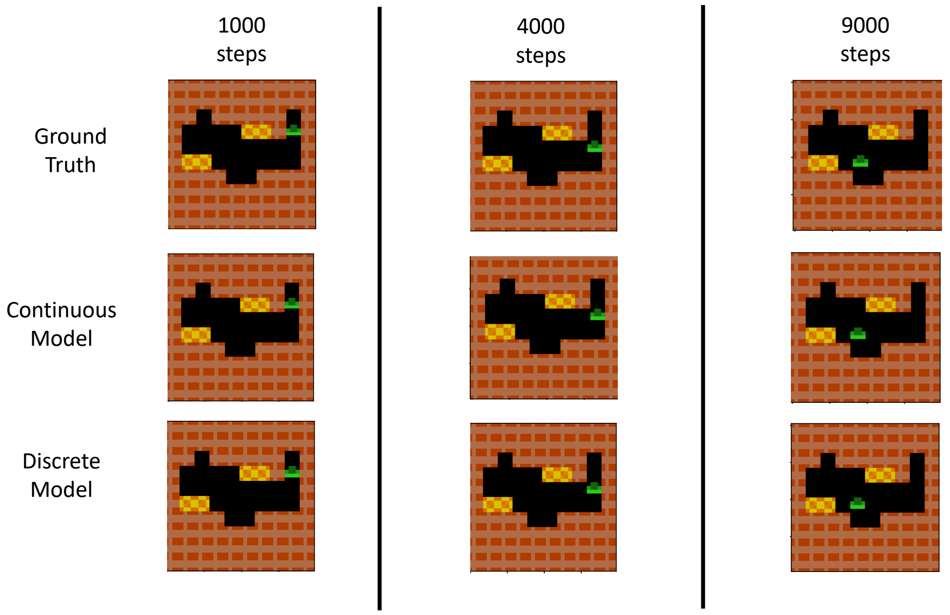


Figure 5: A visualization of the reconstructions for models with continuous and discrete latent states at different timesteps for Sokoban. Both the discrete and continuous models accurately reconstruct the ground truth image after thousands of timesteps.

A detailed comparison of DeepCubeAI to DeepCubeA and PDBs is shown in Table 1. The results show that DeepCubeAI has a longer path cost than DeepCubeA. This could be because DeepCubeAI is learning a more complex heuristic function that generalizes over goals, while DeepCubeA is trained for a predetermined 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 expands fewer nodes and takes less time when finding solutions. This is most likely due to the speedup provided by Q* search. Unlike DeepCubeA and PDBs, DeepCubeAI is not given a model and can readily generalize to other goals.

Domain	Solver	Len	Opt	Nodes	Secs	Nodes/Sec	Solved
RC	PDBs$^+$[35]	20.67	100.0%	2.05E+06	2.20	1.79E+06	100%
	DeepCubeA [1]	21.50	60.3%	6.62E+06	24.22	2.90E+05	100%
	DeepCubeAI	22.85	19.5%	2.00E+05	6.21	3.22E+04	100%
RC$_{rev}$	DeepCubeAI	22.81	21.92%	2.00E+05	6.30	3.18+04	99.9%
Sokoban	LevinTS[30]	39.80	-	6.60E+03	-	-	100%
	LevinTS[30] (*)	39.50	-	5.03E+03	-	-	100%
	LAMA[30]	51.60	-	3.15E+03	-	-	100%
	DeepCubeA [1]	32.88	-	1.05E+03	2.35	5.60E+01	100%
	DeepCubeAI	33.12	-	3.30E+03	2.62	1.38E+03	100%

Table 1: Comparison of DeepCubeAI with 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. PDBs$^+$ refers to domain-specific PDBs for the Rubik's cube that leverage knowledge of group theory [35, 36].

6 Future Work

6.1 Correcting Model Errors

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, during search, an error of greater than 0.5 was made by the model, meaning rounding was unable to correct it. Future work could address these rare mistakes by training an additional DNN to correct slightly corrupted latent states. This way, if the model makes an error of a few bits, the correction DNN can be used to recover from the error.

6.2 Robust Goal Specification

Similar to research in model-based reinforcement learning [42], 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, one may know only high-level information about a goal without knowing the low-level details. In these scenarios, 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 [3]. This approach can be extended to learned models and allow one to specify goals without having to generate any goal images.

6.3 Extensions to Stochastic and Partially Observable Environments

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 to lack of knowledge of the environment dynamics (such as for chaotic systems) and partially observable due to limited sensing ability. Research has been done on learning models in stochastic environments by training DNNs to sample possible next states [25, 19]. Sequence models, such as recurrent neural networks [24] have been used to learn to embed belief states [22, 13] 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 both the Rubik's cube and Sokoban, results show that DeepCubeAI solves over 99% of test cases and effectively generalizes across goal states.

References

[1] F. Agostinelli, S. McAleer, A. Shmakov, and P. Baldi. Solving the Rubik's cube with deep reinforcement learning and search. *Nature Machine Intelligence*, 1(8):356–363, 2019.

[2] F. Agostinelli, S. McAleer, A. Shmakov, and P. Baldi. DeepcubeA. `https://github.com/forestagostinelli/DeepCubeA`, 2020.

[3] F. Agostinelli, R. Panta, and V. Khandelwal. Specifying goals to deep neural networks with answer set programming. In *ICAPS 2023 Workshop on Human-Aware Explainable Planning*, 2023.

[4] F. Agostinelli, A. Shmakov, S. McAleer, R. Fox, and P. Baldi. A* search without expansions: Learning heuristic functions with deep q-networks. *arXiv preprint arXiv:2102.04518*, 2021.

[5] M. Andrychowicz, F. Wolski, A. Ray, J. Schneider, R. Fong, P. Welinder, B. McGrew, J. Tobin, O. P. Abbeel, and W. Zaremba. Hindsight experience replay. In *Advances in Neural Information Processing Systems*, pages 5048–5058, 2017.

[6] K. G. R. K. S. R. T. W. D. R. A. S. L. O. T. E. G. W. D. S. T. L. V. V. Arthur Guez, Mehdi Mirza. An investigation of model-free planning: boxoban levels. https://github.com/deepmind/boxoban-levels/, 2018.

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

[8] M. Asai, H. Kajino, A. Fukunaga, and C. Muise. Classical planning in deep latent space. *Journal of Artificial Intelligence Research*, 74:1599–1686, 2022.

[9] M. Bagatella, M. Olšák, M. Rolínek, and G. Martius. Planning from pixels in environments with combinatorially hard search spaces. *Advances in Neural Information Processing Systems*, 34:24707–24718, 2021.

[10] A. G. Barto, S. Singh, N. Chentanez, et al. Intrinsically motivated learning of hierarchical collections of skills. In *Proceedings of the 3rd International Conference on Development and Learning*, volume 112, page 19. Citeseer, 2004.

[11] Y. Bengio, N. Léonard, and A. Courville. Estimating or propagating gradients through stochastic neurons for conditional computation. *arXiv preprint arXiv:1308.3432*, 2013.

[12] D. P. Bertsekas and J. N. Tsitsiklis. *Neuro-dynamic programming*. Athena Scientific, 1996.

[13] A. R. Cassandra, L. P. Kaelbling, and M. L. Littman. Acting optimally in partially observable stochastic domains. In *AAAI*, volume 94, pages 1023–1028, 1994.

[14] J. C. Culberson and J. Schaeffer. Pattern databases. *Computational Intelligence*, 14(3):318–334, 1998.

[15] C. Finn, I. Goodfellow, and S. Levine. Unsupervised learning for physical interaction through video prediction. *Advances in neural information processing systems*, 29, 2016.

[16] M. Ghallab, D. Nau, and P. Traverso. *Automated Planning: theory and practice*. Elsevier, 2004.

[17] X. Glorot, A. Bordes, and Y. Bengio. Deep sparse rectifier neural networks. In *Proceedings of the fourteenth international conference on artificial intelligence and statistics*, pages 315–323, 2011.

[18] D. Hafner, T. Lillicrap, I. Fischer, R. Villegas, D. Ha, H. Lee, and J. Davidson. Learning latent dynamics for planning from pixels. In *International conference on machine learning*, pages 2555–2565. PMLR, 2019.

[19] D. Hafner, T. Lillicrap, M. Norouzi, and J. Ba. Mastering atari with discrete world models. 2021.

[20] D. Hafner, J. Pasukonis, J. Ba, and T. Lillicrap. Mastering diverse domains through world models. *arXiv preprint arXiv:2301.04104*, 2023.

[21] P. E. Hart, N. J. Nilsson, and B. Raphael. A formal basis for the heuristic determination of minimum cost paths. *IEEE transactions on Systems Science and Cybernetics*, 4(2):100–107, 1968.

[22] M. Hausknecht and P. Stone. Deep recurrent q-learning for partially observable mdps. In *2015 AAAI fall symposium series*, 2015.

[23] M. Helmert. The fast downward planning system. *Journal of Artificial Intelligence Research*, 26:191–246, 2006.

[24] S. Hochreiter and J. Schmidhuber. Long short-term memory. *Neural computation*, 9(8):1735–1780, 1997.

[25] L. Kaiser, M. Babaeizadeh, P. Milos, B. Osinski, R. H. Campbell, K. Czechowski, D. Erhan, C. Finn, P. Kozakowski, S. Levine, et al. Model-based reinforcement learning for Atari. *International Conference on Learning Representations*.

[26] D. P. Kingma and J. Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

[27] V. Mnih, K. Kavukcuoglu, D. Silver, A. A. Rusu, J. Veness, M. G. Bellemare, A. Graves, M. Riedmiller, A. K. Fidjeland, G. Ostrovski, et al. Human-level control through deep reinforcement learning. *Nature*, 518(7540):529–533, 2015.

[28] B. Muppasani, V. Pallagani, B. Srivastava, and F. Agostinelli. On solving the rubik's cube with domain-independent planners using standard representations. *arXiv preprint arXiv:2307.13552*, 2023.

[29] J. Oh, X. Guo, H. Lee, R. L. Lewis, and S. Singh. Action-conditional video prediction using deep networks in atari games. *Advances in neural information processing systems*, 28, 2015.

[30] L. Orseau, L. Lelis, T. Lattimore, and T. Weber. Single-agent policy tree search with guarantees. In *Advances in Neural Information Processing Systems*, pages 3201–3211, 2018.

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

[32] M. L. Puterman. *Markov decision processes: discrete stochastic dynamic programming*. John Wiley & Sons, 2014.

[33] M. L. Puterman and M. C. Shin. Modified policy iteration algorithms for discounted markov decision problems. *Management Science*, 24(11):1127–1137, 1978.

[34] S. Racanière, T. Weber, D. Reichert, L. Buesing, A. Guez, D. J. Rezende, A. P. Badia, O. Vinyals, N. Heess, Y. Li, et al. Imagination-augmented agents for deep reinforcement learning. In *Advances in neural information processing systems*, pages 5690–5701, 2017.

[35] T. Rokicki. cube20. `https://github.com/rokicki/cube20src`, 2016.

[36] T. Rokicki, H. Kociemba, M. Davidson, and J. Dethridge. The diameter of the Rubik's cube group is twenty. *SIAM Review*, 56(4):645–670, 2014.

[37] J. Schmidhuber. Deep learning in neural networks: An overview. *Neural networks*, 61:85–117, 2015.

[38] J. Schulman, F. Wolski, P. Dhariwal, A. Radford, and O. Klimov. Proximal policy optimization algorithms. *arXiv preprint arXiv:1707.06347*, 2017.

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

[40] R. S. Sutton and A. G. Barto. *Reinforcement learning: An introduction*. MIT press, 2018.

[41] S. Tian, S. Nair, F. Ebert, S. Dasari, B. Eysenbach, C. Finn, and S. Levine. Model-based visual planning with self-supervised functional distances. *International Conference on Learning Representations*, 2021.

[42] S. Tian, S. Nair, F. Ebert, S. Dasari, B. Eysenbach, C. Finn, and S. Levine. Model-based visual planning with self-supervised functional distances. In *International Conference on Learning Representations*, 2021.

[43] C. J. Watkins and P. Dayan. Q-learning. *Machine learning*, 8(3-4):279–292, 1992.