An Investigation of Model-Free Planning

Arthur Guez [*1] **Mehdi Mirza** [*1] **Karol Gregor** [*1] **Rishabh Kabra** [*1] **Sébastien Racanière** [1] **Théophane Weber** [1] **David Raposo** [1] **Adam Santoro** [1] **Laurent Orseau** [1] **Tom Eccles** [1] **Greg Wayne** [1] **David Silver** [1] **Timothy Lillicrap** [1]

Abstract

The field of reinforcement learning (RL) is facing increasingly challenging domains with combinatorial complexity. For an RL agent to address these challenges, it is essential that it can plan effectively. Prior work has typically utilized an explicit model of the environment, combined with a specific planning algorithm (such as tree search). More recently, a new family of methods have been proposed that learn how to plan, by providing the structure for planning via an inductive bias in the function approximator (such as a tree structured neural network), trained end-to-end by a model-free RL algorithm. In this paper, we go even further, and demonstrate empirically that an entirely model-free approach, without special structure beyond standard neural network components such as convolutional networks and LSTMs, can learn to exhibit many of the characteristics typically associated with a model-based planner. We measure our agent's effectiveness at planning in terms of its ability to generalize across a combinatorial and irreversible state space, its data efficiency, and its ability to utilize additional thinking time. We find that our agent has many of the characteristics that one might expect to find in a planning algorithm. Furthermore, it exceeds the state-of-the-art in challenging combinatorial domains such as Sokoban and outperforms other model-free approaches that utilize strong inductive biases toward planning.

1. Introduction

One of the aspirations of artificial intelligence is a cognitive agent that can adaptively and dynamically form plans to achieve its goal. Traditionally, this role has been filled by model-based RL approaches, which first learn an explicit model of the environment's system dynamics or rules, and then apply a planning algorithm (such as tree search) to the learned model. Model-based approaches are potentially powerful but have been challenging to scale with learned models in complex and high-dimensional environments (Talvitie, 2014; Asadi et al., 2018), though there has been recent progress in that direction (Buesing et al., 2018; Ebert et al., 2018).

More recently, a variety of approaches have been proposed that learn to plan *implicitly*, solely by model-free training. These *model-free planning* agents utilize a special neural architecture that mirrors the structure of a particular planning algorithm. For example the neural network may be designed to represent search trees (Farquhar et al., 2017; Oh et al., 2017; Guez et al., 2018), forward simulations (Racanière et al., 2017; Silver et al., 2016), or dynamic programming (Tamar et al., 2016). The main idea is that, given the appropriate inductive bias for planning, the function approximator can learn to leverage these structures to learn its own planning algorithm. This kind of *algorithmic function approximation* may be more flexible than an explicit model-based approach, allowing the agent to customize the nature of planning to the specific environment.

In this paper we explore the hypothesis that planning may occur implicitly, even when the function approximator has *no special inductive bias toward planning*. Previous work (Pang & Werbos, 1998; Wang et al., 2018) has supported the idea that model-based behavior can be learned with general recurrent architectures, with planning computation amortized over multiple discrete steps (Schmidhuber, 1990), but comprehensive demonstrations of its effectiveness are still missing. Inspired by the successes of deep learning and the universality of neural representations, our main idea is simply to furnish a neural network with a high capacity and flexible representation, rather than mirror any particular planning structure. Given such flexibility, the network can in principle learn its own algorithm for approximate planning. Specifically, we utilize a family of neural networks based on a widely used function approximation architecture: the stacked convolutional LSTMs (ConvLSTM by

[*]Equal contribution [1]DeepMind, London, UK. Correspondence to: <{aguez, mmirza, rkabra, countzero}@google.com>.

Proceedings of the 36th International Conference on Machine Learning, Long Beach, California, PMLR 97, 2019. Copyright 2019 by the author(s).

Xingjian et al. (2015)).

It is perhaps surprising that a purely model-free reinforcement learning approach can be so successful in domains that would appear to necessitate explicit planning. This raises a natural question: what is planning? Can a model-free RL agent be considered to be planning, without any explicit model of the environment, and without any explicit simulation of that model?

Indeed, in many definitions (Sutton et al., 1998), planning requires some explicit deliberation using a model, typically by considering possible future situations using a forward model to choose an appropriate sequence of actions. These definitions emphasize the nature of the mechanism (the explicit look-ahead), rather than the effect it produces (the foresight). However, what would one say about a deep network that has been trained from examples in a challenging domain to emulate such a planning process with near-perfect fidelity? Should a definition of planning rule out the resulting agent as effectively planning?

Instead of tying ourselves to a definition that depends on the inner workings of an agent, in this paper we take a behaviourist approach to measuring planning as a property of the agent's interactions. In particular, we consider three key properties that an agent equipped with planning should exhibit.

First, an effective planning algorithm should be able to generalize with relative ease to different situations. The intuition here is that a simple function approximator will struggle to predict accurately across a combinatorial space of possibilities (for example the value of all chess positions), but a planning algorithm can perform a local search to dynamically compute predictions (for example by tree search). We measure this property using procedural environments (such as random gridworlds, Sokoban (Racanière et al., 2017), Boxworld (Zambaldi et al., 2018)) with a massively combinatorial space of possible layouts. We find that our model-free planning agent achieves state-of-the-art performance, and significantly outperforms more specialized model-free planning architectures. We also investigate extrapolation to a harder class of problems beyond those in the training set, and again find that our architecture performs effectively – especially with larger network sizes.

Second, a planning agent should be able to learn efficiently from relatively small amounts of data. Model-based RL is frequently motivated by the intuition that a model (for example the rules of chess) can often be learned more efficiently than direct predictions (for example the value of all chess positions). We measure this property by training our model-free planner on small data-sets, and find that our model-free planning agent still performs well and generalizes effectively to a held-out test set.

Third, an effective planning algorithm should be able to make good use of additional thinking time. Put simply, the more the algorithm thinks, the better its performance should be. This property is likely to be especially important in domains with irreversible consequences to wrong decisions (e.g. death or dead-ends). We measure this property in Sokoban by adding additional thinking time at the start of an episode, before the agent commits to a strategy, and find that our model-free planning agent solves considerably more problems.

Together, our results suggest that a model-free agent, without specific planning-inspired network structure, can learn to exhibit many of the behavioural characteristics of planning. The architecture presented in this paper serves to illustrate this point, and shows the surprising power of one simple approach. We hope our findings broaden the search for more general architectures that can tackle an even wider range of planning domains.

2. Methods

We first motivate and describe the main network architecture we use in this paper. Then we briefly explain our training setup. More details can be found in Appendix 8.

2.1. Model architectures

We desire models that can represent and learn powerful but unspecified planning procedures. Rather than encode strong inductive biases toward particular planning algorithms, we choose high-capacity neural network architectures that are capable of representing a very rich class of functions. As in many works in deep RL, we make use of convolutional neural networks (known to exploit the spatial structure inherent in visual domains) and LSTMs (known to be effective in sequential problems). Aside from these weak but common inductive biases, we keep our architecture as general and flexible as possible, and trust in standard model-free reinforcement learning algorithms to discover the capacity to plan.

2.1.1. Basic architecture

The basic element of the architecture is a ConvLSTM (Xingjian et al., 2015) – a neural network similar to an LSTM but with a 3D hidden state and convolutional operations. A recurrent network f_θ stacks together ConvLSTM modules. For a stack depth of D, the state s contains all the cell states c_d and outputs h_d of each module d: $s = (c_1, \ldots, c_D, h_1, \ldots, h_D)$. The module weights $\theta = (\theta_1, \ldots, \theta_D)$ are not shared along the stack. Given a previous state and an input tensor i, the next state is computed as $s' = f_\theta(s, i)$. The network f_θ is then repeated N times within each time-step (i.e., multiple internal ticks per

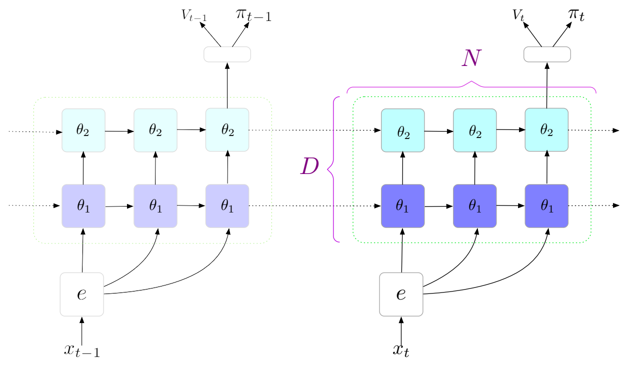


Figure 1. Illustration of the agent's network architecture. This diagram shows DRC(2,3) for two time steps. Square boxes denote ConvLSTM modules and the rectangle box represents an MLP. Boxes with the same color share parameters.

real time-step). If s_{t-1} is the state at the end of the previous time-step, we obtain the new state given the input i_t as:

$$s_t = g_\theta(s_{t-1}, i_t) = \underbrace{f_\theta(f_\theta(\ldots f_\theta(s_{t-1}, i_t), \ldots, i_t), i_t)}_{N \text{ times}} \tag{1}$$

The elements of s_t all preserve the spatial dimensions of the input i_t. The final output o_t of the recurrent network for a single time-step is h_D, the hidden state of the deepest ConvLSTM module after N ticks, obtained from s_t. We describe the ConvLSTM itself and alternative choices for memory modules in Appendix 8.

The rest of the network is rather generic. An encoder network e composed of convolutional layers processes the input observation x_t into a $H \times W \times C$ tensor i_t — given as input to the recurrent module g. The encoded input i_t is also combined with o_t through a skip-connection to produce the final network output. The network output is then flattened and an action distribution π_t and a state-value v_t are computed via a fully-connected MLP. The diagram in Fig 1 illustrates the full network.

From here on, we refer to this architecture as Deep Repeated ConvLSTM (DRC) network architecture, and sometimes followed explicitly by the value of D and N (e.g., DRC(3, 2) has depth $D = 3$ and $N = 2$ repeats).

2.1.2. Additional details

Less essential design choices in the architectures are described here. Ablation studies show that these are not crucial, but do marginally improve performance (see Appendix 11).

Encoded observation skip-connection The encoded observation i_t is provided as an input to all ConvLSTM modules in the stack.

Top-down skip connection As described above, the flow of information in the network only goes up (and right through time). To allow for more general computation we add feedback connection from the last layer at one time step to the first layer of the next step.

Pool-and-inject To allow information to propagate faster in the spatial dimensions than the size of the convolutional kernel within the ConvLSTM stack, it is useful to provide a pooled version of the module's last output h as an additional input on lateral connections. We use both max and mean pooling. Each pooling operation applies pooling spatially for each channel dimension, followed by a linear transform, and then tiles the result back into a 2D tensor. This is operation is related to the pool-and-inject method introduced by Racanière et al. (2017) and to Squeeze-and-Excitation blocks (Hu et al., 2017).

Padding The convolutional operator is translation invariant. To help it understand where the edge of the input image is, we append a feature map to the input of the convolutional operators that has ones on the boundary and zeros inside.

2.2. Reinforcement Learning

We consider domains that are formally specified as RL problems, where agents must learn via reward feedback obtained by interacting with the environment (Sutton et al., 1998). At each time-step t, the agent's network outputs a policy $\pi_t = \pi_\theta(\cdot|h_t)$ which maps the history of observations $h_t := (x_0, \ldots, x_t)$ into a probability distribution over actions, from which the action $a_t \sim \pi_t$, is sampled. In addition, it outputs $v_t = v_\theta(h_t) \in \mathbb{R}$, an estimate of the current policy value, $v^\pi(h_t) = \mathbb{E}[G_t|h_t]$, where $G_t = \sum_{t' \geq t} \gamma^{t'-t} R_{t'}$ is the return from time t, $\gamma \leq 1$ is a discount factor, and R_t the reward at time t.

Both quantities are trained in an actor-critic setup where the policy (actor) is gradually updated to improve its expected return, and the value (critic) is used as a baseline to reduce the variance of the policy update. The update to the policy parameters have the following form using the score function estimator (a la REINFORCE (Williams, 1992)): $(g_t - v_\theta(h_t))\nabla_\theta \log \pi_\theta(a_t|h_t)$.

In practice, we use truncated returns with bootstrapping for g_t and we apply importance sampling corrections if the trajectory data is off-policy. More specifically, we used a distributed framework and the IMPALA V-trace actor-critic algorithm (Espeholt et al., 2018). While we found this training regime to help for training networks with more parameters, we also ran experiments which demonstrate that the DRC architecture can be trained effectively with A3C (Mnih et al., 2016). More details on the setup can be found in Appendix 9.2.

3. Planning Domains

The RL domains we focus on are combinatorial domains for which episodes are procedurally generated. The procedural and combinatorial aspects emphasize planning and generalization since it is not possible to simply memorize an observation to action mapping. In these domains each episode is instantiated in a pseudorandom configuration, so solving an episode typically requires some form of reasoning. Most of the environments are fully-observable and have simple 2D visual features. The domains are illustrated and explained further in Appendix 6. In addition to the planning domains listed below, we also run control experiments on a set of Atari 2600 games (Bellemare et al., 2013).

Gridworld A simple navigation domain following (Tamar et al., 2016), consisting of a grid filled with obstacles. The agent, goal, and obstacles are randomly placed for each episode.

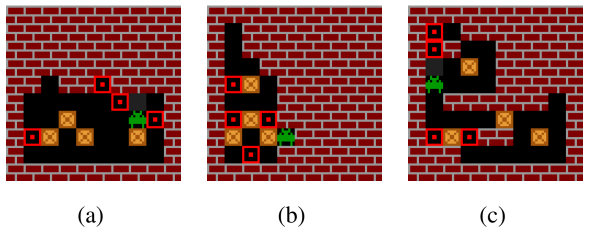


Figure 2. Examples of Sokoban levels from the (a) unfiltered, (b) medium test sets, and from the (c) hard set. Our best model is able to solve all three levels.

Sokoban A difficult puzzle domain requiring an agent to push a set of boxes onto goal locations (Botea et al., 2003; Racanière et al., 2017). Irreversible wrong moves can make the puzzle unsolvable. We describe how we generate a large number of levels (for the fixed problem size 10x10 with 4 boxes) at multiple difficulty levels in Appendix 7, and then split some into a training and test set. Briefly, problems in the first difficulty level are obtained from directly sampling a source distribution (we call that difficulty level *unfiltered*). Then the *medium* and *hard* sets are obtained by sequentially filtering that distribution based on an agent's success on each level. We are releasing these levels as datasets in the standard Sokoban format[1]. Unless otherwise specified, we ran experiments with the easier *unfiltered* set of levels.

Boxworld Introduced in (Zambaldi et al., 2018), the aim is to reach a goal target by collecting coloured keys and opening colour-matched boxes until a target is reached. The agent can see the keys (i.e., their colours) locked within boxes; thus, it must carefully plan the sequence of boxes that should be opened so that it can collect the keys that will lead to the target. Keys can only be used once, so opening an incorrect box can lead the agent down a dead-end path from which it cannot recover.

[1]https://github.com/deepmind/boxoban-levels

MiniPacman (Racanière et al., 2017). The player explores a maze that contains food while being chased by ghosts. The aim of the player is to collect all the rewarding food. There are also a few power pills which allow the player to attack ghosts (for a brief duration) and earn a large reward. See Appendix 6.2 for more details.

4. Results

Paralleling our behaviourist approach to the question of planning, we look at three areas of analysis in our results. We first examine the performance of our model and other approaches across combinatorial domains that emphasize planning over memorization (Section 4.1).[2] We also report results aimed at understanding how elements of our architecture contribute to observed performance. Second, we examine questions of data-efficiency and generalization in Section 4.2. Third, we study evidence of iterative computation in Section 4.3.

4.1. Performance & Comparisons

In general, across all domains listed in Section 3, the DRC architecture performed very well with only modest tuning of hyper-parameters (see Appendix 9). The DRC(3,3) variant was almost always the best in terms both of data efficiency (early learning) and asymptotic performance.

Gridworld: Many methods efficiently learn the Gridworld domain, especially for small grid sizes. We found that for larger grid sizes the DRC architecture learns more efficiently than a vanilla Convolutional Neural Network (CNN) architecture of similar weight and computational capacity. We also tested Value Iteration Networks (VIN) (Tamar et al., 2016), which are specially designed to deal with this kind of problem (i.e. local transitions in a fully-observable 2D state space). We found that VIN, which has many fewer parameters and a well-matched inductive bias, starts improving faster than other methods. It outperformed the CNN and even the DRC during early-stage training, but the DRC reached better final accuracy (see Table 1 and Figure 14a in the Appendix). Concurrent to our work, Lee et al. (2018) observed similar findings in various path planning settings when comparing VIN to an architecture with weaker inductive biases.

[2]Illustrative videos of trained agents and playable demo available at https://sites.google.com/view/modelfreeplanning/

[3]This percentage at 1e9 is lower than the 90% reported originally by I2A (Racanière et al., 2017). This can be explained by some training differences between this paper and the I2A paper: train/test dataset vs. procedurally generated levels, co-trained vs. pre-trained models. See appendix 13.5 for more details.

Table 1. Performance comparison in Gridworld, size 32x32, after 10M environment steps. VIN (Tamar et al., 2016) experiments are detailed in Appendix 13.1.

Model	% solved at $1e6$ steps	% solved at $1e7$ steps
DRC(3, 3)	30	**99**
VIN	**80**	97
CNN	3	90

Table 2. Comparison of test performance on (unfiltered) Sokoban levels for various methods. I2A (Racanière et al., 2017) results are re-rerun within our framework. ATreeC (Farquhar et al., 2017) experiments are detailed in Appendix 13.2. MCTSnets (Guez et al., 2018) also considered the same Sokoban domain but in an expert imitation setting (achieving 84% solved levels).

Model	% solved at $2e7$ steps	% solved at $1e9$ steps
DRC(3, 3)	**80**	**99**
ResNet	14	96
CNN	25	92
I2A (unroll=15)	21	84[3]
1D LSTM(3,3)	5	74
ATreeC	1	57
VIN	12	56

Sokoban: In Sokoban, we demonstrate state-of-the-art results versus prior work which targeted similar box-pushing puzzle domains (ATreeC (Farquhar et al., 2017), I2A (Racanière et al., 2017)) and other generic networks (LSTM (Hochreiter & Schmidhuber, 1997), ResNet (He et al., 2016), CNNs). We also test VIN on Sokoban, adapting the original approach to our state space by adding an input encoder to the model and an attention module at the output to deal with the imperfect state-action mappings. Table 2 compares the results for different architectures at the end of training. Only 1% of test levels remain unsolved by DRC(3,3) after 1e9 steps, with the second-best architecture (a large ResNet) failing four times as often.

Boxworld: On this domain several methods obtain near-perfect final performance. Still, the DRC model learned faster than published methods, achieving ≈80% success after 2e8 steps. In comparison, the best ResNet achieved ≈50% by this point. The relational method of Zambaldi et al. (2018) can learn this task well but only solved <10% of levels after 2e8 steps.

MiniPacman: Here again, we found that the DRC architecture trained faster and obtained a better score than the ResNet architectures we tried (see Figure 15a).

Atari 2600 To test the capacity of the DRC model to deal with richer sensory data, we also examined its performance on five planning-focused Atari games (Bellemare et al., 2013). We obtained state-of-the-art scores on three of five games, and competitive scores on the other two (see Appendix 10.2 and Figure 10 for details).

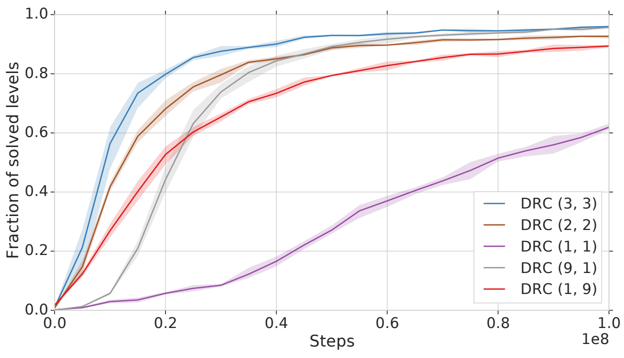
4.1.1. Influence of network architecture

We studied the influence of stacking and repeating the ConvLSTM modules in the DRC architecture, controlled by the parameters D (stack length) and N (number of repeats) as described in Section 2.1. These degrees of freedom allow our networks to compute its output using shared, iterative, computation with $N > 1$, as well as computations at different levels of representation and more capacity with $D > 1$. We found that the DRC(3,3) (i.e, $D = 3, N = 3$) worked robustly across all of the tested domain. We compared this to using the same number of modules stacked without repeats (DRC(9,1)) or only repeated without stacking (DRC(1,9)). In addition, we also look at the same smaller capacity versions $D = 2, N = 2$ and $D = 1, N = 1$ (which reduces to a standard ConvLSTM). Figure 3a shows the results on Sokoban for the different network configurations. In general, the versions with more capacity performed better. When trading-off stacking and repeating (with total of 9 modules), we observed that only repeating without stacking was not as effective (this has the same number of parameters as the DRC(1,1) version), and only stacking was slower to train in the early phase but obtained a similar final performance. We also confirmed that DRC(3,3) performed better than DRC(1,1) in Boxworld, MiniPacman, and Gridworld.

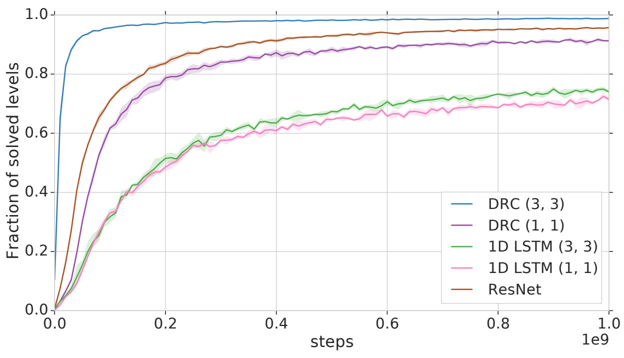
On harder Sokoban levels (Medium-difficulty dataset), we trained the DRC(3,3) and the larger capacity DRC(9,1) configurations and found that, even though DRC(9,1) was slower to learn at first, it ended up reaching a better score than DRC(3,3) (94% versus 91.5% after 1e9 steps). See Fig 9 in appendix. We tested the resulting DRC(9,1) agent on the hardest Sokoban setting (Hard-difficulty), and found that it solved 80% of levels in less than 48 minutes of evaluation time. In comparison, running a powerful tree search algorithm, Levin Tree Search (Orseau et al., 2018), with a DRC(1,1) as policy prior solves 94%, but in 10 hours of evaluation.

In principle, deep feedforward models should support iterative procedures within a single time-step and perhaps match the performance of our recurrent networks (Jastrzebski et al., 2017). In practice, deep ResNets did not perform as well as our best recurrent models (see Figure 3b), and are in any case incapable of caching implicit iterative planning steps over time steps. Finally, we note that recurrence by itself was also not enough: replacing the ConvLSTM modules with flat 1D LSTMs performed poorly (see Figure 3b).

Across experiments and domains, our results suggests that both the network capacity and the iterative aspect of a model drives the agent's performance. Moreover, in these 2D puz-



(a)



(b)

Figure 3. a) Learning curves for various configurations of DRC in Sokoban-Unfiltered. b) Comparison with other network architectures tuned for Sokoban. Results are on test-set levels.

zle domains, spatial recurrent states significantly contribute to the results.

4.2. Data Efficiency & Generalization

In combinatorial domains generalization is a central issue. Given limited exposure to configurations in an environment, how well can a model perform on unseen scenarios? In the supervised setting, large flexible networks are capable of over-fitting. Thus, one concern when using high-capacity networks is that they may over-fit to the task, for example by memorizing, rather than learning a strategy that can generalize to novel situations. Recent empirical work in SL (Supervised Learning) has shown that the generalization of large networks is not well understood (Zhang et al., 2016; Arpit et al., 2017). Generalization in RL is even less well studied, though recent work (Zhang et al., 2018a;b; Cobbe et al., 2018) has begun to explore the effect of training data diversity.

We explored two main axes in the space of generalization. We varied both the diversity of the environments as well as the size of our models. We trained the DRC architecture in various data regimes, by restricting the number of unique Sokoban levels — during the training, similar to SL, the training algorithm iterates on those limited levels many times. We either train on a Large (900k levels), Medium-size (10k) or Small (1k) set — all subsets of the Sokoban-unfiltered training set. For each dataset size, we compared a larger version of the network, DRC(3,3), to a smaller version DRC(1,1).[4] Results are shown in Figure 4.

In all cases, the larger DRC(3,3) network generalized better than its smaller counterpart, both in absolute terms and in terms of *generalization gap*. In particular, in the Medium-size regime, the generalization gap[5] is 6.5% for DRC(3,3) versus 33.5% for DRC(1, 1). Figures 5a-b compare these same trained models when tested on both the unfiltered and on the medium(-difficulty) test sets. We performed an analogous experiment in the Boxworld environment and observed remarkably similar results (see Figure 5c and Appendix Figure 12).

Looking across these domains and experiments there are two findings that are of particular note. First, unlike analogous SL experiments, reducing the number of training levels does not necessarily improve performance on the train set. Networks trained on 1k levels perform worse in terms of the fraction of levels solved. We believe this is due to the exploration problem in low-diversity regime: With more levels, the training agent faces a natural curriculum to help it progress toward harder levels. Another view of this is that larger networks can overfit the training levels, but only if they experience success on these levels at some point. While local minima for the loss in SL are not practically an issue with large networks, local minima in policy space can be problematic.

From a classic optimization perspective, a surprising finding is that the larger networks in our experiment (both Sokoban & Boxworld) suffer *less* from over-fitting in the low-data regime than their smaller counterparts (see Figure 5). However, this is in line with recent findings (Zhang et al., 2016) in SL that the generalization of a model is driven by the architecture and nature of the data, rather than simply as a results of the network capacity and size of the dataset. Indeed, we also trained the same networks in a purely supervised fashion through imitation learning of an expert policy.[6] We observed a similar result when comparing the classification accuracy of the networks on the test set, with the DRC(3,3) better able to generalize — even though both networks had similar training errors on small datasets.

Extrapolation Another facet of generality in the strategy

[4]DRC(3,3) has around 300K more parameters, and it requires around 3 times more computation

[5]We compute the generalization gap by subtracting the performance (ratio of levels solved) on the training set from performance on the test set.

[6]Data was sampled on-policy from the expert policy executed on levels from the training datasets.

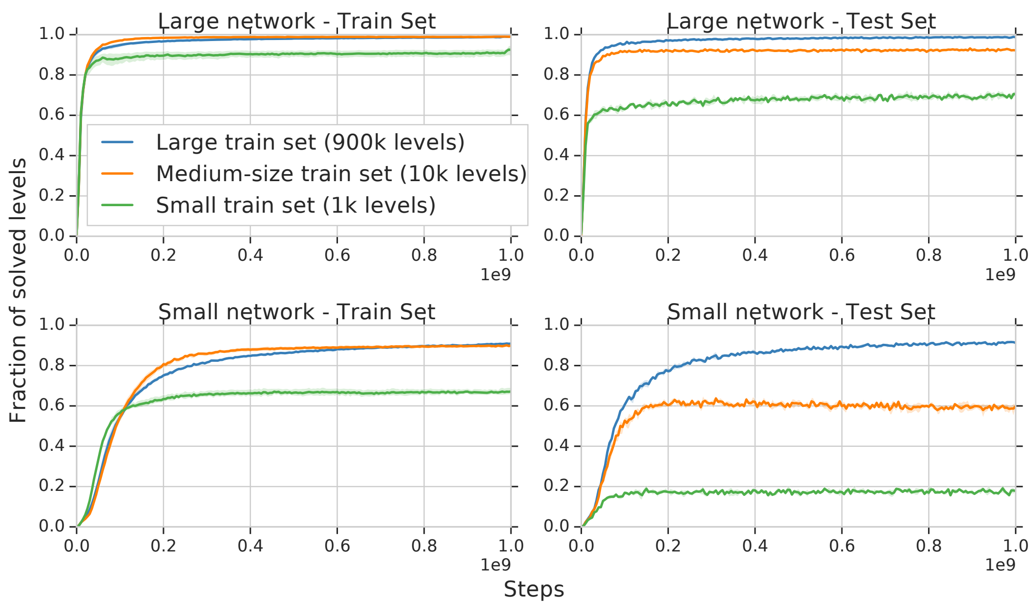


Figure 4. Comparison of DRC(3,3) (Top, Large network) and DRC(1,1) (Bottom, Small network) when trained with RL on various train set sizes (subsets of the Sokoban-unfiltered training set). Left column shows the performance on levels from the corresponding train set, right column shows the performance on the test set (the same set across these experiments).

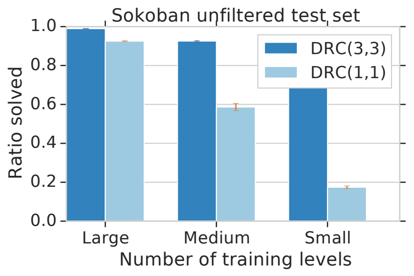
found by the DRC network is how it performs outside the training distribution. In Sokoban, we tested the DRC(3,3) and DRC(1,1) networks on levels with a larger number of boxes than those seen in the training set. Figure 13a shows that DRC was able to extrapolate with little loss in performance to up to 7 boxes (for a a fixed grid size). The performance degradation for DRC(3,3) on 7 boxes was 3.5% and 18.5% for DRC(1,1). In comparison, the results from Racanière et al. (2017) report a loss of 34% when extrapolating to 7 boxes in the same setup.

4.3. Iterative Computation

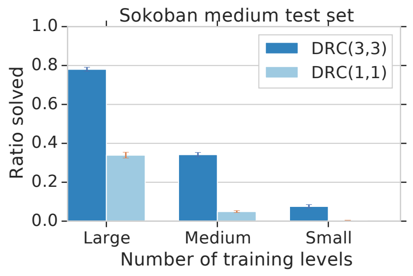
One desirable property for planning mechanisms is that their performance scales with additional computation without seeing new data. Although RNNs (and more recently ResNets) can in principle learn a function that can be iterated to obtain a result (Graves, 2016; Jastrzebski et al., 2017; Greff et al., 2016), it is not clear whether the networks trained in our RL domains learn to amortize computation over time in this way. To test this, we took trained networks in Sokoban (unfiltered) and tested post hoc their ability to improve their results with additional steps. We introduced 'no-op' actions at the start of each episode – up to 10 extra computation steps where the agent's action is fixed to have no effect on the environment. The idea behind forced no-ops is to give the network more computation on the same inputs, intuitively akin to increasing its search time. Under these testing conditions, we observed clear performance improvements on medium difficulty levels of about 5% for DRC networks (see Figure 6). We did not find such improvements for the simpler fully-connected LSTM architecture. This suggests that the DRC networks have learned a scalable strategy for the task which is computed and refined through a series of identical steps, thereby exhibiting one of the essential properties of a planning algorithm.

5. Discussion

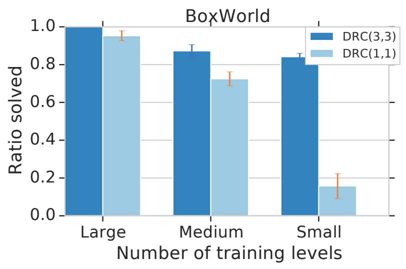
We aspire to endow agents with the capacity to plan effectively in combinatorial domains where simple memorization of strategies is not feasible. An overarching question is regarding the nature of planning itself. Can the computations necessary for planning be learned solely using model-free RL, and can this be achieved by a general-purpose neural network with weak inductive biases? Or is it necessary to have dedicated planning machinery — either explicitly encoding existing planning algorithms, or implicitly mirroring their structure? In this paper, we studied a variety of different neural architectures trained using model-free RL in procedural planning tasks with combinatorial and irreversible state spaces. Our results suggest that general-purpose, high-capacity neural networks based on recurrent convolutional structure, are particularly efficient at learning to plan. This approach yielded state-of-the-art results on several domains – outperforming all of the specialized planning architectures that we tested. Our generalization and scaling analyses, together with the procedural nature of the studied



(a) Sokoban unfiltered set



(b) Sokoban medium set



(c) Boxworld

Figure 5. Generalization results from a trained model on different training set size (Large, Medium and Small subsets of the unfiltered training dataset) in Sokoban when evaluated on (a) the unfiltered test set and (b) the medium-difficulty test set. (c) Similar generalization results for trained models in Boxworld. (These figures show a summary of results in Figure 4 and Appendix Fig. 12.)

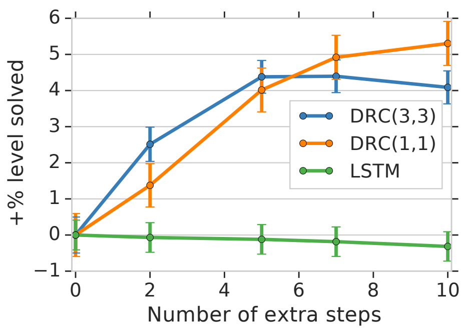


Figure 6. Forcing extra computation steps after training improves the performance of DRC on Sokoban medium-difficulty set (5 networks, each tested on the same 5000 levels). Steps are performed by overriding the policy with no-op actions at the start of an episode. The green line is the LSTM(1,1) model.

domains, suggests that these networks learn an algorithm for approximate planning that is tailored to the domain. The algorithmic function approximator appears to compute its plan dynamically, amortised over many steps, and hence additional thinking time can boost its performance.

There are, of course, many approaches to improving the efficacy of model-free algorithms. For example, DARLA, ICM, and UNREAL improve performance and transfer in RL by shaping representations using an unsupervised loss (Higgins et al., 2017; Pathak et al., 2017; Jaderberg et al., 2016). Our work hints that one of the most important approaches may be to study which inductive biases allow networks to learn effective planning-like behaviours. In principle these approaches are straightforward to combine.

Recent work in the context of supervised learning is pushing us to rethink how large neural network models generalize (Zhang et al., 2016; Arpit et al., 2017). Our results further demonstrate the mismatch between traditional views on generalisation and model size. The surprising efficacy of our planning agent, when trained on a small number of scenarios across a combinatorial state space, suggests that any new theory must take into account the algorithmic function approximation capabilities of the model rather than simplistic measures of its complexity. Ultimately, we desire even more generality and scalability from our agents, and it remains to be seen whether model-free planning will be effective in reinforcement learning environments of real-world complexity.

References

Arpit, D., Jastrzębski, S., Ballas, N., Krueger, D., Bengio, E., Kanwal, M. S., Maharaj, T., Fischer, A., Courville, A., Bengio, Y., et al. A closer look at memorization in deep networks. *arXiv preprint arXiv:1706.05394*, 2017.

Asadi, K., Misra, D., and Littman, M. L. Lipschitz continuity in model-based reinforcement learning. *arXiv preprint arXiv:1804.07193*, 2018.

Bellemare, M. G., Naddaf, Y., Veness, J., and Bowling, M. The arcade learning environment: An evaluation platform for general agents. *Journal of Artificial Intelligence Research*, 47:253–279, 2013.

Botea, A., Müller, M., and Schaeffer, J. Using abstraction for planning in sokoban. In *Computers and Games*, volume 2883, pp. 360, 2003.

Buesing, L., Weber, T., Racaniere, S., Eslami, S., Rezende, D., Reichert, D. P., Viola, F., Besse, F., Gregor, K., Hassabis, D., et al. Learning and querying fast generative models for reinforcement learning. *arXiv preprint arXiv:1802.03006*, 2018.

Cobbe, K., Klimov, O., Hesse, C., Kim, T., and Schulman, J. Quantifying generalization in reinforcement learning. *arXiv preprint arXiv:1812.02341*, 2018.

Ebert, F., Finn, C., Dasari, S., Xie, A., Lee, A., and Levine, S. Visual foresight: Model-based deep reinforcement learning for vision-based robotic control. *arXiv preprint arXiv:1812.00568*, 2018.

Espeholt, L., Soyer, H., Munos, R., Simonyan, K., Mnih, V., Ward, T., Doron, Y., Firoiu, V., Harley, T., Dunning, I., et al. Impala: Scalable distributed deep-rl with importance weighted actor-learner architectures. *arXiv preprint arXiv:1802.01561*, 2018.

Farquhar, G., Rocktäschel, T., Igl, M., and Whiteson, S. Treeqn and atreec: Differentiable tree planning for deep reinforcement learning. In *ICLR*, 2017.

Graves, A. Adaptive computation time for recurrent neural networks. *arXiv preprint arXiv:1603.08983*, 2016.

Greff, K., Srivastava, R. K., and Schmidhuber, J. Highway and residual networks learn unrolled iterative estimation. *arXiv preprint arXiv:1612.07771*, 2016.

Guez, A., Weber, T., Antonoglou, I., Simonyan, K., Vinyals, O., Wierstra, D., Munos, R., and Silver, D. Learning to search with mctsnets. *arXiv preprint arXiv:1802.04697*, 2018.

He, K., Zhang, X., Ren, S., and Sun, J. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pp. 770–778, 2016.

Higgins, I., Pal, A., Rusu, A., Matthey, L., Burgess, C., Pritzel, A., Botvinick, M., Blundell, C., and Lerchner, A. Darla: Improving zero-shot transfer in reinforcement learning. In *Proceedings of the 34th International Conference on Machine Learning-Volume 70*, pp. 1480–1490. JMLR. org, 2017.

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

Horgan, D., Quan, J., Budden, D., Barth-Maron, G., Hessel, M., Van Hasselt, H., and Silver, D. Distributed prioritized experience replay. *arXiv preprint arXiv:1803.00933*, 2018.

Hu, J., Shen, L., and Sun, G. Squeeze-and-excitation networks. *arXiv preprint arXiv:1709.01507*, 7, 2017.

Jaderberg, M., Mnih, V., Czarnecki, W. M., Schaul, T., Leibo, J. Z., Silver, D., and Kavukcuoglu, K. Reinforcement learning with unsupervised auxiliary tasks. *arXiv preprint arXiv:1611.05397*, 2016.

Jastrzebski, S., Arpit, D., Ballas, N., Verma, V., Che, T., and Bengio, Y. Residual connections encourage iterative inference. *arXiv preprint arXiv:1710.04773*, 2017.

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

Lee, L., Parisotto, E., Chaplot, D. S., Xing, E., and Salakhutdinov, R. Gated path planning networks. *arXiv preprint arXiv:1806.06408*, 2018.

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

Mnih, V., Badia, A. P., Mirza, M., Graves, A., Lillicrap, T., Harley, T., Silver, D., and Kavukcuoglu, K. Asynchronous methods for deep reinforcement learning. In *International conference on machine learning*, pp. 1928–1937, 2016.

Oh, J., Singh, S., and Lee, H. Value prediction network. In *Advances in Neural Information Processing Systems*, pp. 6118–6128, 2017.

Orseau, L., Lelis, L., Lattimore, T., and Weber, T. Single-agent policy tree search with guarantees. In *Advances in Neural Information Processing Systems*, 2018.

Pang, X. and Werbos, P. Neural network design for j function approximation in dynamic programming. *arXiv preprint adap-org/9806001*, 1998.

Pathak, D., Agrawal, P., Efros, A. A., and Darrell, T. Curiosity-driven exploration by self-supervised prediction. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition Workshops*, pp. 16–17, 2017.

Racanière, S., Weber, T., Reichert, D., Buesing, L., Guez, A., Rezende, D. J., Badia, A. P., Vinyals, O., Heess, N., Li, Y., et al. Imagination-augmented agents for deep reinforcement learning. In *Advances in Neural Information Processing Systems*, pp. 5690–5701, 2017.

Schmidhuber, J. Making the world differentiable: On using self-supervised fully recurrent neural networks for dynamic reinforcement learning and planning in non-stationary environments. 1990.

Silver, D., van Hasselt, H., Hessel, M., Schaul, T., Guez, A., Harley, T., Dulac-Arnold, G., Reichert, D., Rabinowitz, N., Barreto, A., et al. The predictron: End-to-end learning and planning. *arXiv preprint arXiv:1612.08810*, 2016.

Sutton, R. S., Barto, A. G., et al. *Reinforcement learning: An introduction*. 1998.

Talvitie, E. Model regularization for stable sample rollouts. In *Proceedings of the Thirtieth Conference on Uncertainty in Artificial Intelligence*, pp. 780–789. AUAI Press, 2014.

Tamar, A., Wu, Y., Thomas, G., Levine, S., and Abbeel, P. Value iteration networks. In *Advances in Neural Information Processing Systems*, pp. 2154–2162, 2016.

Wang, J. X., Kurth-Nelson, Z., Kumaran, D., Tirumala, D., Soyer, H., Leibo, J. Z., Hassabis, D., and Botvinick, M. Prefrontal cortex as a meta-reinforcement learning system. *Nature neuroscience*, 21(6):860, 2018.

Williams, R. J. Simple statistical gradient-following algorithms for connectionist reinforcement learning. *Machine learning*, 8(3-4):229–256, 1992.

Xingjian, S., Chen, Z., Wang, H., Yeung, D.-Y., Wong, W.-K., and Woo, W.-c. Convolutional lstm network: A machine learning approach for precipitation nowcasting. In *Advances in neural information processing systems*, pp. 802–810, 2015.

Zambaldi, V., Raposo, D., Santoro, A., Bapst, V., Li, Y., Babuschkin, I., Tuyls, K., Reichert, D., Lillicrap, T., Lockhart, E., et al. Relational deep reinforcement learning. *arXiv preprint arXiv:1806.01830*, 2018.

Zhang, A., Ballas, N., and Pineau, J. A dissection of overfitting and generalization in continuous reinforcement learning. *arXiv preprint arXiv:1806.07937*, 2018a.

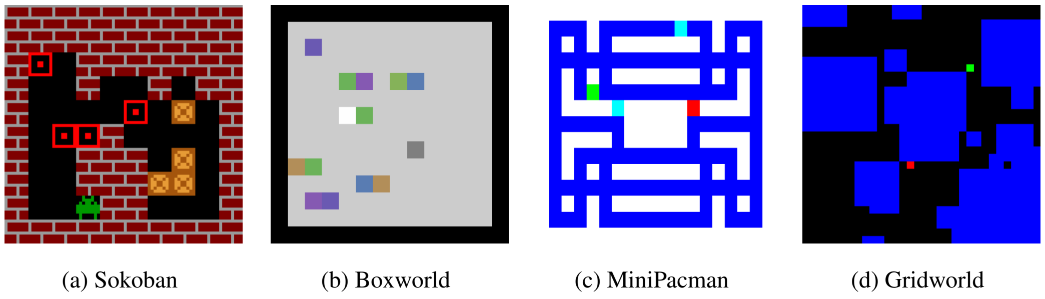
Zhang, C., Bengio, S., Hardt, M., Recht, B., and Vinyals, O. Understanding deep learning requires rethinking generalization. *arXiv preprint arXiv:1611.03530*, 2016.

Zhang, C., Vinyals, O., Munos, R., and Bengio, S. A study on overfitting in deep reinforcement learning. *arXiv preprint arXiv:1804.06893*, 2018b.

Appendix

The appendix is organized as follows: we first describe the five environments we used in this paper. We also provide details about the dataset generation process used for the Sokoban levels that we are releasing. Next, we describe the DRC architecture, various choices of memory type, and all the implementation details. We also list model parameters and RL training hyper-parameters for the DRC and other baseline models. We then present some additional experiments, including ablation studies on the DRC and an extension of our generalization analysis. Finally we compare the DRC against various baseline agents (VIN, ATreeC, and ResNet).

6. Domains



(a) Sokoban (b) Boxworld (c) MiniPacman (d) Gridworld

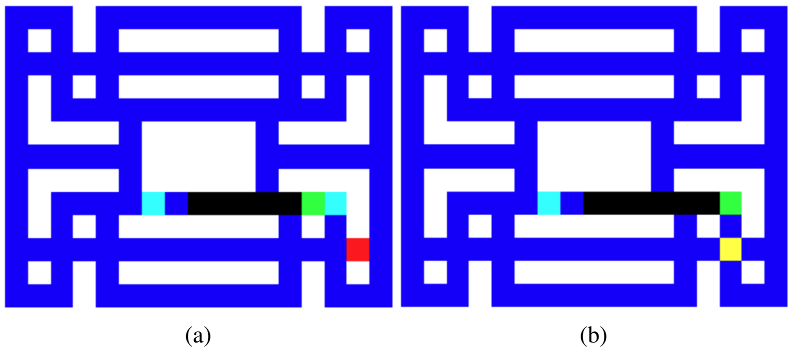
Figure 7. Sample observations for each of the planning environments at the beginning of an episode.

6.1. Sokoban

We follow the reward structure in (Racanière et al., 2017), with a reward of 10 for completing a level, 1 for getting a box on a target, and -1 for removing a box from a target, in addition to a cost per time-step of -0.01. Each episode is capped at 120 frames. We use a sprite-based representation of the state as input. Each sprite is an (8, 8, 3)-shaped RGB image.

Dataset generation is described in Appendix 7. Unless otherwise noted, we have used the levels from the unfiltered dataset in experiments.

6.2. MiniPacman



(a) (b)

Figure 8. (a) Food in dark blue offers a reward of 1. After it is eaten, the food disappears, leaving a black space. Power pills in light blue offer a reward of 2. The player is in green, and ghosts in red. An episode ends when the player is eaten by a ghost. When the player has eaten all the food, the episode continues but the level is reset with power pills and ghosts placed at random positions. (b) When the player eats a power pill, ghosts turn yellow and become edible, with a larger reward or 5 for the player. They then progressively return to red (via orange), at which point they are once again dangerous for the player.

MiniPacman is a simplified version of the popular game Pac-Man. The first, deterministic version, was introduced in (Racanière et al., 2017). Here we use a stochastic version, where at every time-step, each ghost moves with a probability of 0.95. This simulates, in a gridworld, the fact that ghosts move slower than the player. An implementation of this game is available under the name Pill Eater at `https://github.com/vasiloglou/mltrain-nips-2017/tree/master/sebastien_racaniere`, where 5 different reward structures (or *modes*) are available.

6.3. Boxworld

In all our experiments we use branch length of 3 and maximum solution length of 4. Each episode is capped at 120 frames. Unlike Sokoban, Boxworld levels are generated by the game engine. Unlike Sokoban we do not used sprite-based images; the environment state is provided as a (14, 14, 3) tensor.

6.4. Atari

We use the standard Atari setup of (Mnih et al., 2015) with up to 30 no-ops at random, episodes capped at 30 mins, an action repeat of 4, and the full set of 18 actions. We made little effort to tune the hyperparameters for Atari; only the encoder size was increased. See Appendix 10.2 for details about the experiments, and Figure 10 for the learning curves.

6.5. Gridworld

We generate 32x32-shaped levels by sampling a number of obstacles between 12 and 24. Each obstacle is a square, with side in [2, 10]. These obstacles may overlap. The player and goal are placed on random different empty squares. We use rejection sampling to ensure the player can reach the goal. Episodes terminate when reaching a goal or when stepping on an obstacle, with a reward of 1 and -1 respectively. There is a small negative reward of -0.01 at each step otherwise.

7. Dataset generation details

The unfiltered Sokoban levels are generated by the procedure described in (Racanière et al., 2017). Simple hashing is used to avoid repeats. To generate medium levels, we trained a DRC(1,1) policy whose LSTM state is reset at every time step. It achieves around 95% of level solved on the easy set. Then, we generate medium difficulty levels by rejection sampling: we sample levels using the procedure from (Racanière et al., 2017), and accept a level if the policy cannot solve it after 10 attempts. The levels are not subsampled from the easy levels dataset but from fresh generation instead, so overlap between easy and medium is minimal (though there are 120 levels in common in the easy and medium datasets). To generate hard levels, we train a DRC(3,3) policy on a mixture of easy and medium levels, until it reached a performance level of around 85%. We then select the first 3332 levels of a separate medium dataset which are not solved by that policy after 10 attempts (so there is no overlap between hard and medium sets). The datasets for all level types are available at https://github.com/deepmind/boxoban-levels. A playable demo of some challenging levels is available at https://sites.google.com/view/modelfreeplanning/.

Table 3. Best results obtained on the test set for the different difficulty levels across RL agents within 2e9 steps of training (averaged across 5 independent runs). See Figure 9 for training curves.

Unfiltered	Medium	Hard
99%	95%	80%

Table 4. Number of levels in each subset of the dataset and number of overlaps between them.

	Unfiltered train	Unfiltered test	Medium train	Medium test	hard
Unfiltered train	900,000	0	119	10	0
Unfiltered test	0	100,000	10	1	0
Medium train	119	1	450,000	0	0
Medium test	10	0	0	50,000	0
Hard	0	0	0	0	3332

8. Model

We denoted $g_\theta(s_{t-1}, i_t) = f_\theta(f_\theta(\ldots f_\theta(s_{t-1}, i_t), \ldots, i_t), i_t)$ as the computation at a full time-step of the DRC(D, N) architecture. Here $\theta = (\theta_1, \ldots, \theta_D)$ are the parameters of D stacked memory modules, and $i_t = e(x_t)$ is the agent's encoded observation.

Let $s^1_{t-1}, \ldots, s^N_{t-1}$ be the state of the D stacked memory modules at the N ticks of time-step $t-1$. Each $s^n_{t-1} = (c^n_1, \ldots, c^n_d, h^n_1, \ldots, h^n_d)$. We then have the "repeat" recurrence below:

$$\begin{aligned} s^1_{t-1} &= f_\theta(s_{t-1}, i_t) \\ s^n_{t-1} &= f_\theta(s^{n-1}_{t-1}, i_t) \text{ for } 1 < n \leq N \\ s_t &= s^N_{t-1} \end{aligned} \tag{2}$$

We can now describe the computation within a single tick, i.e., outputs c^n_d and h^n_d at $d = 1, \ldots, D$ for a fixed n. This is the "stack" recurrence:

$$c^n_d, h^n_d = \text{MemoryModule}_{\theta_d}(i_t, c^{n-1}_d, h^{n-1}_d, h^n_{d-1}) \tag{3}$$

Note that the encoded observation i_t is fed as an input not only at all ticks 1, …, N, but also at all depths 1, …, D of the memory stack. (The latter is described under "encoded observation skip-connections" in Section 2.1.2 but not depicted in Figure 1 for clarity.)

Generally each memory module is a ConvLSTM parameterized by θ_d at location (d, n) of the DRC grid. But we describe alternative choices of memory module in Appendix 8.2.

8.1. Additional details/improvements

8.1.1. Top-down skip connection

As described in the stack recurrence, we feed the hidden state of each memory module as an input h^n_{d-1} to the next module in the stack. But this input is only available for modules at depth $d > 1$. Instead of setting $h^n_0 = \mathbf{0}$, we use a top-down skip connection i.e. $h^n_0 = h^{n-1}_D$. We will explore the role of this skip-connection in Appendix 11.

8.1.2. Pool-and-inject

The pool operation aggregates hidden states across their spatial dimensions to give vectors $m^n_1, \ldots, m^n_D$. We project these through a linear layer each (weights $W_{p_1}, \ldots, W_{p_D}$), and then tile them over space to obtain summary tensors $p^n_1, \ldots, p^n_D$. These have the same shapes as the original hidden states, and can be injected as additional inputs to the memory modules at the next tick.

$$\begin{aligned} m^n_d &:= [\text{max}_{H,W}(h^n_d), \text{mean}_{H,W}(h^n_d)]^T \\ p^n_d &:= \text{Tile}_{H,W}(W_{p_d} m^n_d) \end{aligned} \tag{4}$$

Finally, p^{n-1}_d is provided as an additional input at location (d,n) of the DRC, and equation 8 becomes:

$$c^n_d, h^n_d = \text{MemoryModule}_{\theta_d}(i_t, c^{n-1}_d, h^{n-1}_d, h^n_{d-1}, p^{n-1}_d)$$

8.2. Memory modules

8.2.1. ConvLSTM

$$c_d^n, h_d^n = \text{ConvLSTM}_{\theta_d}(i_t, c_d^{n-1}, h_d^{n-1}, h_{d-1}^n)$$

For all $d > 1$:

$$f_d^n = \sigma(W_{fi} * i_t + W_{fh_1} * h_{d-1}^n + W_{fh_2} * h_d^{n-1} + b_f)$$
$$i_d^n = \sigma(W_{ii} * i_t + W_{ih_1} * h_{d-1}^n + W_{ih_2} * h_d^{n-1} + b_i)$$
$$o_d^n = \sigma(W_{oi} * i_t + W_{oh_1} * h_{d-1}^n + W_{oh_2} * h_d^{n-1} + b_o)$$
$$c_d^n = f_d^n \odot c_d^{n-1} + i_d^n \odot \tanh(W_{ci} * i_t + W_{ch_1} * h_{d-1}^n + W_{ch_2} * h_d^{n-1} + b_c)$$
$$h_d^n = o_d^n \odot \tanh(c_d^n)$$

For $d = 1$, we use the top-down skip connection h_D^{n-1} in place of h_{d-1}^n as described above.

Here $*$ denotes the convolution operator, and $\odot$ denotes point-wise multiplication. Note that $\theta_d = (W_{f\cdot}, W_{i\cdot}, W_{o\cdot}, W_{c\cdot}, b_f, b_i, b_o, b_c)_d$ parameterizes the computation of the forget gate, input gate, output gate, and new cell state. i_d^n and o_d^n should not be confused with the encoded input i_t or the final output o_t of the entire network.

8.2.2. GatedConvRNN

This is a simpler module with no cell state.

$$h_d^n = \text{GatedConvRNN}_{\theta_d}(i_t, h_d^{n-1}, h_{d-1}^n)$$

For all $d > 1$:

$$o_d^n = \sigma(W_{oi} * i_t + W_{oh_1} * h_{d-1}^n + W_{oh_2} * h_d^{n-1} + b_o)$$
$$h_d^n = o_d^n \odot \tanh(W_{hi} * i_t + W_{hh_1} * h_{d-1}^n + W_{hh_2} * h_d^{n-1} + b_h)$$

$\theta_d = (W_{o\cdot}, W_{h\cdot}, b_o, b_h)_d$ has half as many parameters as in the case of the ConvLSTM.

8.2.3. SimpleConvRNN

This is a classic RNN without gating.

$$h_d^n = \tanh(W_{hi} * i_t + W_{hh_1} * h_{d-1}^n + W_{hh_2} * h_d^{n-1} + b_h)$$

$\theta_d = (W_{h\cdot}, b_h)_d$ has half as many parameters as the GatedConvRNN, and only a quarter as the ConvLSTM.

We will explore the difference between these memory types in Appendix 11. Otherwise, we use the ConvLSTM for all experiments.

9. Hyper-Parameters

We tuned our hyper-parameters for the DRC models on Sokoban and used the same settings for all other domains. We only changed the encoder network architectures to accommodate the specific observation format of each domain.

9.1. Network parameters

All activation functions are ReLU unless otherwise specified.

Observation size

- Sokoban: (80, 80, 3)

- Boxworld: (14, 14, 3)
- MiniPacman: (15, 19, 3)
- Gridworld: (32, 32, 1)
- Atari: (210, 160, 3)

Encoder network is a multilayer CNN with following domain-specific parameters:

- Sokoban: number of channels: (32, 32), kernel size: (8, 4), strides: (4, 2)
- Boxworld: number of channels: (32, 32), kernel size: (3, 2), strides: (1, 1)
- MiniPacman: number of channels: (32, 32), kernel size: (3, 3), strides: (1, 1)
- Gridworld: number of channels: (64, 64, 32), kernel size: (3, 3, 2), strides: (1, 1, 2)
- Atari: number of channels: (16, 32, 64, 128, 64, 32), kernel size: (8, 4, 4, 3, 3, 3), strides: (4, 2, 2, 1, 1, 1)

DRC(D, N): all ConvLSTM modules use one convolution with 128 output channels, a kernel size of 3 and stride of 1. All cell states and hidden states have 32 channels.

1D LSTM(D, N): all LSTM modules have hidden size 200. On Sokoban, the (10,10,32)-sized encoded observation is flattened and compressed via a fully connected layer (with 200 units) before it is fed to the LSTM modules. On Atari, however, we flatten the (14,10,32)-sized encoded observation and feed it directly to the LSTM(D, N). This results in a much larger network for Atari.

Policy: single hidden layer MLP with 256 units.

Total number of parameters for all our models are shown in Table 5.

Table 5. Number of trainable parameters for all models we compared

Model	Sokoban	Gridworld	Boxworld	MiniPacman	Atari
DRC(3,3)	2,042,376	4,353,642	3,615,847	5,096,488	2,896,229
DRC(1,1)	1,752,456	-	3,313,639	4,782,888	2,601,317
LSTM(3,3)	2,152,992 (compressed obs)	-	-	-	13,711,877 (uncompressed obs)
LSTM(1,1)	1,088,192 (compressed obs)	-	-	-	5,159,077 (uncompressed obs)
I2A (unroll=15)	7,781,269	-	-	-	-
AtreeC	3,269,065	-	-	-	-
VIN	62,506	19,978	-	-	-
ResNet	52,584,982	-	3,814,197	2,490,902	-
CNN	2,089,222	5,159,755	-	-	-

9.2. RL training setup

We use the V-trace actor-critic algorithm described by (Espeholt et al., 2018), with 4 GPUs for each learner and 200 actors generating trajectories. We reduce variance and improve stability by using λ-returns targets ($\lambda = 0.97$) and a smaller discount factor ($\gamma = 0.97$). This marginally reduces the maximum performance observed, but increases the stability and average performance across runs, allowing better comparisons. For all experiments, we use a BPTT (Backpropagation Through Time) unroll of length 20 and a batch size of 32. We use the Adam optimizer (Kingma & Ba, 2014). The learning rate is initialized to 4e-4 and is annealed to 0 over 1.5e9 environment steps with polynomial annealing. The other Adam optimizer parameters are $\beta_1 = 0.9, \beta_2 = 0.999, \epsilon =$1e-4. The entropy and baseline loss weights are set to 0.01 and 0.5 respectively. We also apply a $\mathcal{L}^2$ norm cost with a weight of 1e-3 on the logits, and a $\mathcal{L}^2$ regularization cost with a weight of 1e-5 to the linear layers that compute the baseline value and logits.

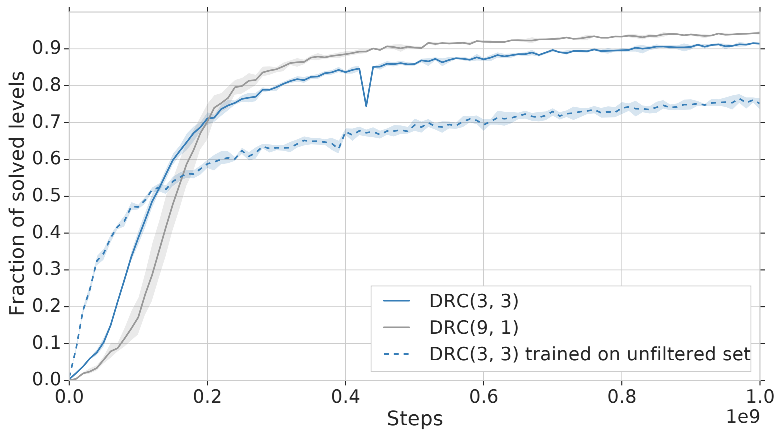


Figure 9. Learning curves in Sokoban for DRC architectures tested on the medium-difficulty test set. The dashed curve shows DRC(3,3) trained on the easier unfiltered dataset. The solid curves show DRC(3,3) and DRC(9,1) trained directly on the medium-difficulty train set.

10. Extra experiments

10.1. Sokoban

Figure 9 shows learning curve when different DRC architectures have been trained on the medium set as opposed to unfiltered set in the main text.

10.2. Atari

We train DRC(3,3) and DRC(1,1) on five Atari 2600 games: *Alien*, *Asteroid*, *Breakout*, *Ms. Pac-Man*, and *Up'n Down*. We picked these games for their planning flavor. We also train two baseline models, substituting ConvLSTMs for 1D LSTMs, in both the (1,1) and (3,3) configurations. These LSTM baselines are fed a flattened encoded observation, but the rest of the architecture is identical. We do not tune any hyperparameters for this domain and only consider out-of-the-box performance.

Figure 10 shows learning curves comparing these models on the above games. We also plot final training scores achieved by ApeX-DQN (Horgan et al., 2018) for reference. DRC(3,3) does significantly better than the LSTM baselines on two of the games (*Asteroid* and *Up'n Down*), and comparably on others. Given DRC(3,3) has a lot fewer parameters than LSTM(1,1) and LSTM(3,3), it appears to make more effective use of them. It's also worth noting that stack-and-repeat (3,3)-representations do offer an improvement on (1,1)- representations for both the DRC and the 1D LSTM architectures. On the whole, we need further study to characterize when DRC(D,N) offers an advantage over LSTM(D,N) on Atari domains.

Our scores generally improve with more training. For example, if we let the DRC(3,3) run for longer, it reaches scores above 75000 in Ms. Pacman.

11. Extra ablation studies

In Figure 11, we compare the Sokoban test-set performance of our baseline DRC(3,3) agent against various ablated versions. These results justify our choice of the ConvLSTM memory module and the architectural improvements described in Section 2.1.2.

11.1. Memory type

The ConvLSTM is the top-performing memory module among those we tested. The Gated ConvRNN module comes out very close with half as many parameters. We surmise that the gap between these two types could be larger for other domains.

The Simple ConvRNN suffers from instability and high variance as expected, asserting that some gating is essential for the DRC's performance.

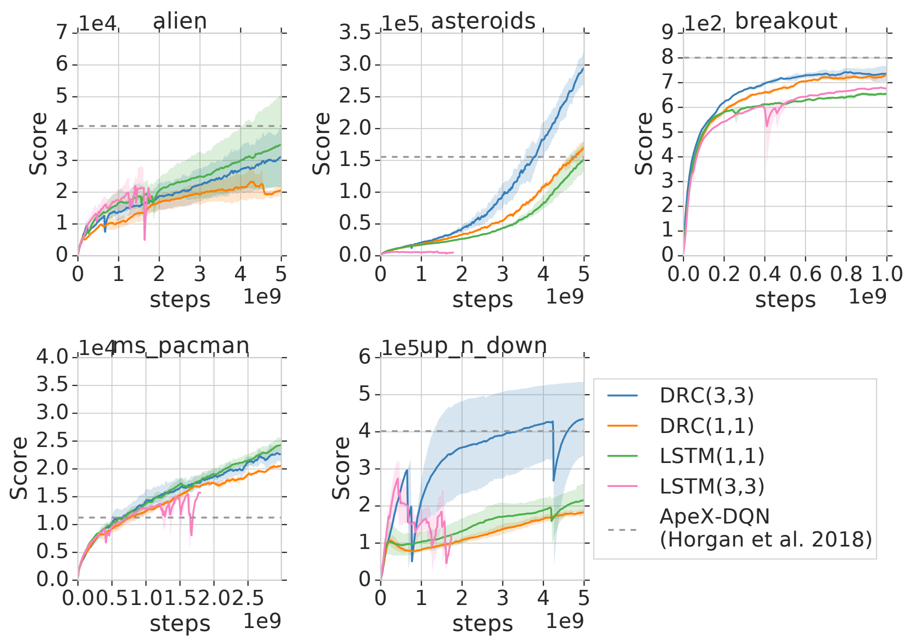


Figure 10. Learning curves comparing the DRC(3,3), DRC(1,1), LSTM(3,3), and LSTM(1,1) network configurations on five Atari 2600 games. Results are averaged over two independent runs for DRC(D,N) agents and five independent runs for LSTM(D,N) agents. We also provide ApeX-DQN (no-op regime) results from (Horgan et al., 2018) as a reference.

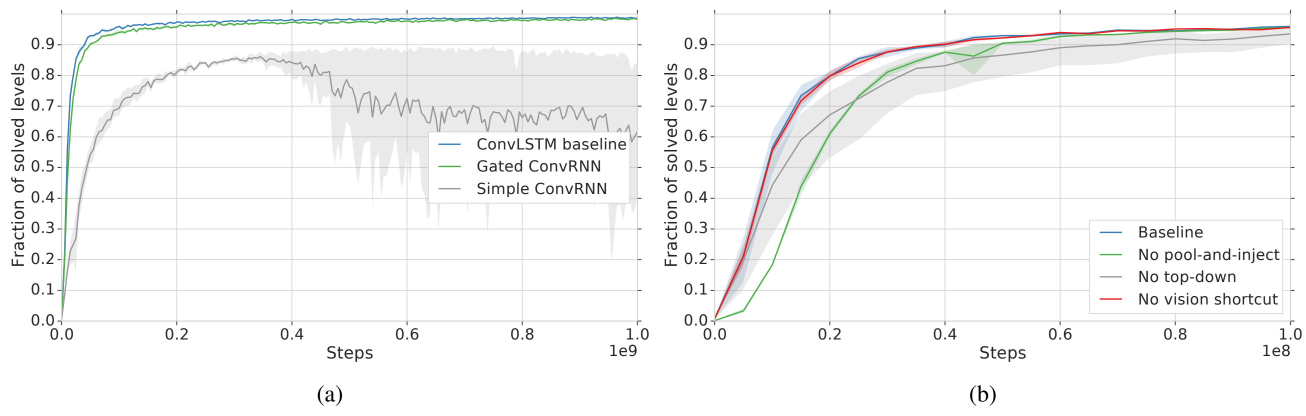


Figure 11. Ablation studies on the performance of our baseline DRC(3,3) agent on Sokoban. All curves show test-set performance. a) We replace the ConvLSTM for simpler memory modules. b) We remove our extra implementation details, namely pool-and-inject, the vision shortcut, and the top-down skip connection, from the model.

11.2. Additional improvements

Without **pool-and-inject**, the network learns significantly slower at early stages but nevertheless converges to the same performance as the baseline. The **vision shortcut** is a residual connection from the output of the encoder module to the output of the DRC. It has little influence on the performance of the baseline model. This is likely because we already feed the encoded observation to every (d,n)-location of the DRC grid (described as "encoded observation skip-connections" in Section 2.1.2 and Appendix 8). Without the **top-down skip connection**, the model exhibits larger performance variance and also slightly lower final performance. On the whole, none of these are critical to the model's performance.

12. Generalization

For generalization experiments in Boxworld we generate levels online for each level-set size using seeds. Large approximately corresponds to 22k levels, medium to 5k, and small to 800. But at each iteration it's guaranteed that same levels are generated. Figure 12 shows results of similar experiment as Figure 4 but for Boxworld domain.

Figure 13a shows the extrapolation results on Sokoban when a model that is trained with 4 boxes is tested on levels with larger number of boxes. Performance degradation for DRC(3,3) is very minimal and the models is still able to perform with performance of above 90% even on the levels with 7 boxes, whereas the performance on for DRC(1,1) drops to under 80%.

Figure 13b shows the generalization gap for Sokoban. Generalization gap is computed by the difference of performance (ratio of the levels solved) between train set and test set. The gap increase substantially more for DRC(1,1) compared to DRC(3,3) as the training set size decreases.

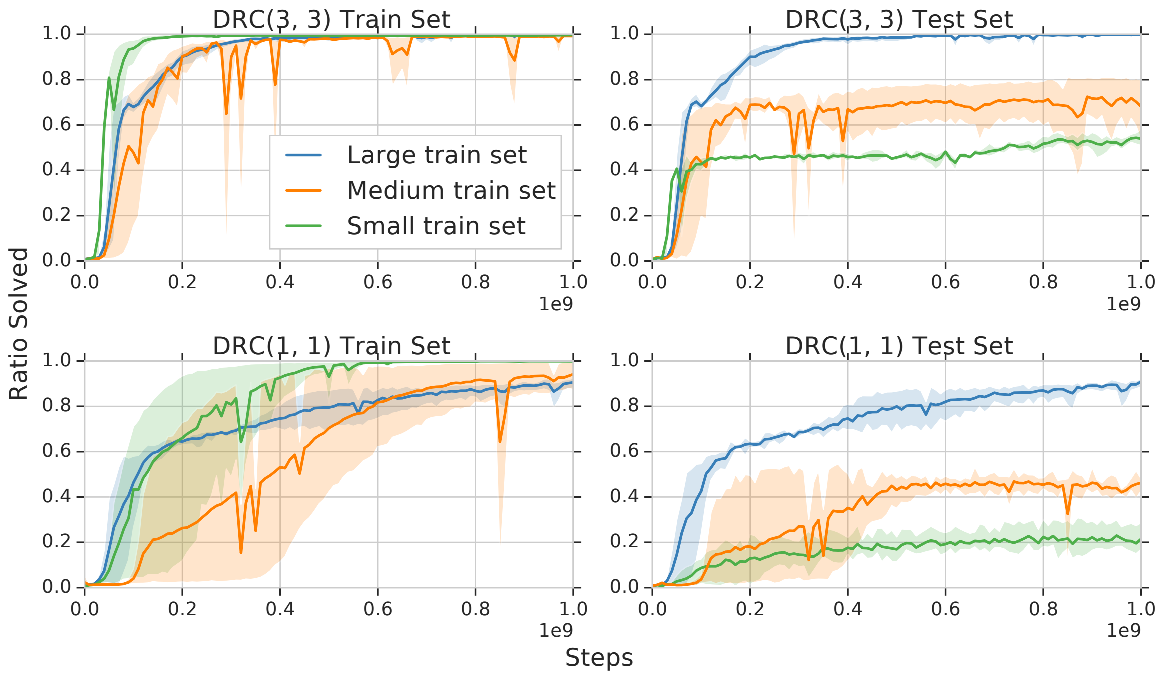


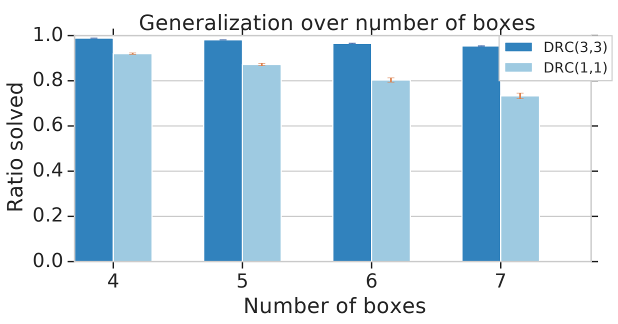
Figure 12. Generalization performance on Boxworld when model is trained on different dataset sizes

13. Baseline architectures and experiments

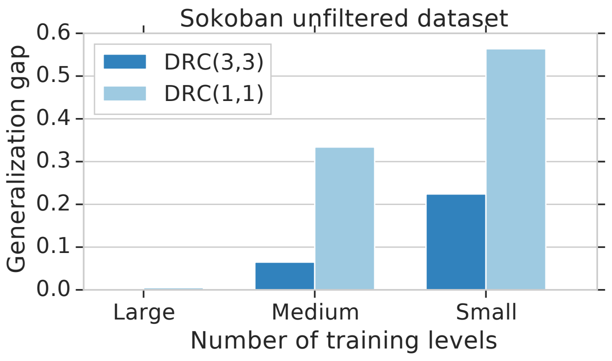
13.1. Value Iteration Networks

Our implementation of VIN closely follows (Tamar et al., 2016). This includes using knowledge of the player position in the mechanism for attention over the q-values produced by the VIN module in the case of Gridworld.

For Sokoban, which provides pixel observations, we feed an encoded frame I to the VIN module. The convolutional encoder is identical to that used by our DRC models (see Appendix 9.1). We also couldn't use the player position directly; instead



(a) Extrapolation over number of boxes.



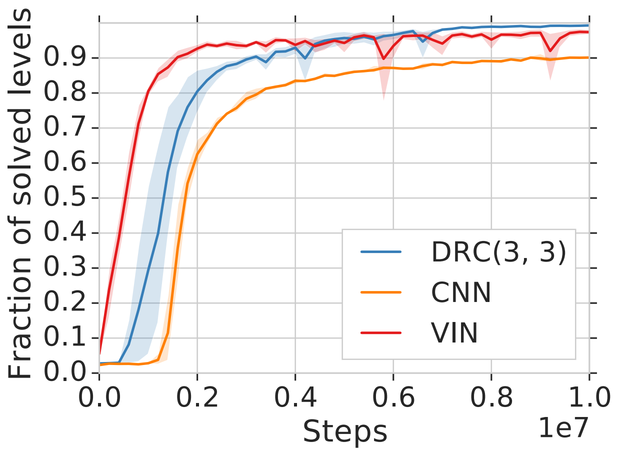
(b) Generalization gap for different train set size.

Figure 13. (a) Extrapolation to larger number of boxes than those seen in training. The model was trained on 4 boxes. (b) Generalization gap computed as the difference between performance (in % of level solved) on train set against test set. Smaller bars correspond to better generalization.

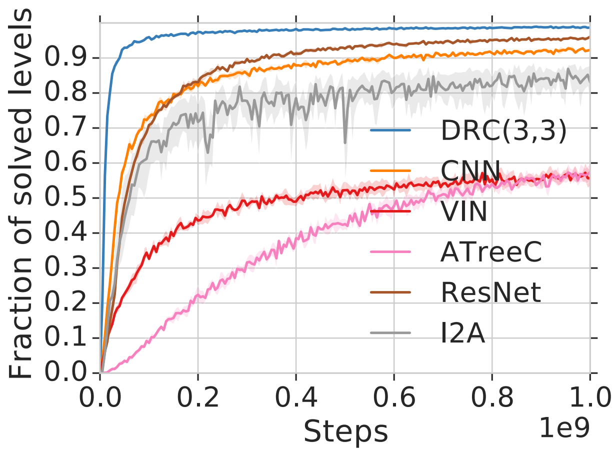
we use a learned soft attention mask $A = \sigma(W_{attention} * I)$ over the 2D state space, and sum over the attended values $A \odot \bar{Q}_a$ to obtain an attention-modulated value per abstract action a.

We introduce an additional choice of update rule in our experiments. The original algorithm sets $\bar{Q}_a{}' = W^a_{transition} * [\bar{R} : \bar{V}]$ at each value iteration. We also try the following alternative: $\bar{Q}_a{}' = \bar{R} + \gamma(W^a_{transition} * \bar{V})$ (note that ':' denotes concatenation along the feature dimension and '$*$' denotes 2D convolution). Here, γ is a discounting parameter independent of the MDP's discount factor. In both cases, $\bar{R} = W_{reward} * I$ and $\bar{V} = \max_a \bar{Q}_a$. Although these equations denote a single convolutional operation in the computation of $\bar{Q}_a$, $\bar{R}$, and the attention map A, in practice we use two convolutional layers in each case with a relu activation in between.

We performed parameter sweeps over the choice of update rules (including γ), learning rate initial value, value iteration depth, number of abstract actions, and intermediate reward channels. The annealing schedule for the learning rate was the same as described in Appendix 9.2. In total we had 64 parameter combinations for Gridworld, and 102 for Sokoban. We show the average of the top five performing runs from each sweep in Figures 14a and 14b.

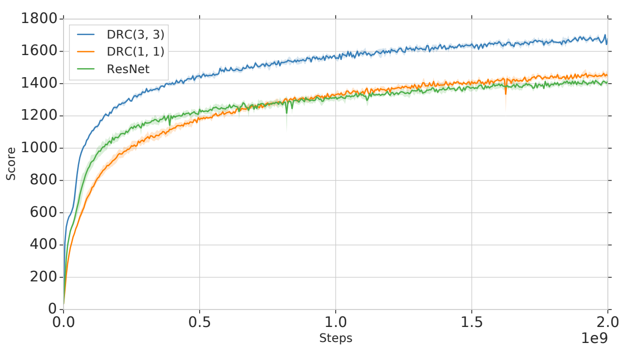


(a)

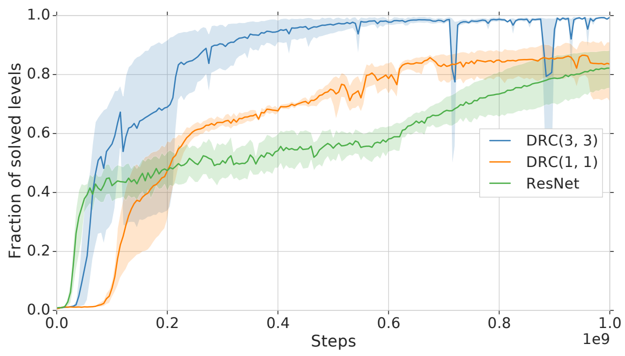


(b)

Figure 14. Performance curves comparing DRC(3,3) against baseline architectures on (a) Gridworld (32x32) and (b) Sokoban. The Sokoban curves show the test-set performance. For VIN we average the top five performing curves from the parameter sweep on each domain. For the other architectures we follow the approach described in Appendix 14, averaging not the top five curves but five independent replicas for fixed hyperparameters.



(a) MiniPacman



(b) Boxworld

Figure 15. Performance curves comparing DRC(3,3) and DRC(1,1) against the best ResNet architectures we found for (a) MiniPacman and (b) Boxworld. DRC(3,3) reaches nearly 100% on Boxworld at $1e9$ steps.

13.2. ATreeC

We implement and use the actor-critic formulation of (Farquhar et al., 2017). We feed it similarly encoded observations from Sokoban as in our DRC and VIN setups. This is flattened and passed through a fully connected layer before the tree planning and backup steps. We do not use any auxiliary losses for reward or state grounding.

We swept over the learning rate initial value, tree depth (2 or 3), embedding size (128, 512), choice of value aggregation (max or softmax), and TD-lambda (0.8, 0.9). We average the top five best-performing runs from the sweep in Figure 14b.

13.3. ResNet

We experimented with different ResNet architectures for each domain and chose one that performs best for each:

Boxworld We used a three-layer CNN as encoder with (16, 32, 64) channels, kernel shape of 2 and stride of 1. This was followed by 8 ResNet blocks. Each block consisted of two CNN layers with 64 channels, kernel shape of 3 and stride 1. The output was flattened and passed to an MLP (1 hidden layer of 256 units), before computing the policy and baseline.

MiniPacman We used the same ResNet architecture as the one for Boxworld, the only difference being that the initial CNN layer channels were (16, 32, 32) and the ResNet blocks had 32 channels.

Sokoban We did not use a separate observation encoder before the ResNet blocks. Instead, we had nine (CNN+ResNet+ResNet) sections. The leading CNN layer in each section could potentially be used for down-sampling using the stride parameter. The ResNet blocks consisted of one CNN layer each, with the same kernel size and output channels as the leading CNN layer of the section. The exact parameters for each section were as follows: (32, 32, 64, 64, 64, 64, 64, 64, 64) output channels, (8, 4, 4, 4, 4, 4, 4, 4, 4) kernel sizes, and (4, 2, 1, 1, 1, 1, 1, 1, 1) strides for the down-sampling CNN layer. Finally, the output was flattened and passed to an MLP with a single hidden layer of 256 units, which provided the policy and baseline outputs.

Figures 15a and 15b compare the ResNets above against DRC(3,3) and DRC(1,1) on MiniPacman and Boxworld.

13.4. CNN

Sokoban The CNN we used was similar to the encoder network for DRC, but with more layers. It had (32, 32, 64, 64, 64, 64, 64, 64, 64) channels, (8, 4, 4, 4, 4, 4, 4, 4, 4) kernel sizes, and strides of (4, 2, 1, 1, 1, 1, 1, 1, 1).

Gridworld The CNN had (128, 128, 128, 128, 128, 128, 128, 64) channels, constant kernel size 3, and strides of (1, 1, 2, 1, 1, 1, 1, 1).

On both domains, the output of the CNN was flattened and passed to an MLP with a single hidden layer of 256 units, which provided the policy and baseline outputs.

13.5. I2A

We use the I2A implementation available at `https://github.com/vasiloglou/mltrain-nips-2017/tree/master/sebastien_racaniere` with the following modifications:

- We replace the FrameProcessing class with a 3-layer CNN with kernel sizes $(8, 3, 3)$, strides $(8, 1, 1)$ and channels $(32, 64, 64)$; the output of the last convolution is then flattened and passed through a linear layer with 512 outputs.
- The model-free path passed to the I2A class was a FrameProcessing as describe above.
- The RolloutPolicy is a 2-layer CNN with kernel sizes $(8, 1)$, strides $(8, 1)$ and channels $(16, 16)$; the output of the last convolution is flattened and passed through an MLP with one hidden layer of size 128, and output size 5 (the number of actions in Sokoban).
- The environment model is similar to the one used in (Racanière et al., 2017). It consists of a deterministic model that given a frame and one-hot encoded action, outputs a predicted next frame and reward. The input frame is first reduced in size using a convolutional layer with kernel size 8, stride 8 and 32 channels. The action is then combined with the output of this layer using pool-and-inject (see (Racanière et al., 2017)). This is followed by 2 size preserving residual blocks with convolutions of shape 3. The output of the last residual block is then passed to two separate networks to predict frame and reward. The frame prediction head uses a convolution layer with shape 3, stride 1, followed by a de-convolution with stride 8, kernel shape 8 and 3 channels. The reward prediction head uses a convolution layer with shape 3, stride 1, followed by a linear layer with output size 3. We predict rewards binned in 3 categories (less than -1, between $[-1, 1]$ and greater than 1), reducing it to a classification problem. (Racanière et al., 2017).

The agent was trained with the V-trace actor-critic algorithm described by (Espeholt et al., 2018).

The percentage of levels solved at 1e9 in table 2 is lower than the 90% reported originally by I2A (Racanière et al., 2017). This can be explained by some differences between the present paper and the I2A paper. Firstly, the original I2A used procedurally generated levels and did not require a train/test split for the dataset. This prevented any form of over-fitting, which is happening here since the test performance of the agent is 88% at 1e9 steps. Secondly, in this paper, the environment model was not pre-trained. Instead it was trained online, using the same data that was used to train the RL loss.

14. Details about figures

Unless otherwise noted, each curve is the average of five independent runs (with identical parameters). We also show a 95% confidence interval (using a shaded area) around each averaged curve. Before independent runs are averaged, we smoothen them with a window size of 5 million steps. Steps in RL learning curves refer to the total number of environment steps seen by the actors.