Planning behavior in a recurrent neural network that plays Sokoban

Anonymous Authors

Abstract
To predict how advanced neural networks generalize to novel situations, it is essential to understand how they reason. Guez et al. (2019, “An investigation of model-free planning”) presented recurrent neural network (RNN) trained with model-free reinforcement learning whose ability to solve Sokoban puzzles improved when given extra steps at episode starts. We replicate and open-source their setup, and further investigate the planning behavior. We find that the RNN starts to benefit from thinking early on in training, and that in many cases extra thinking time reduces redundant actions that the RNN takes. Our findings suggest that the RNN learns to think during training despite the penalties, indicating emergent planning behavior. We believe the small size (1.28M parameters), straightforward input/output and interesting behavior of this model will greatly interest the mechanistic interpretability community.

1. Introduction
Humans benefit from pondering for many tasks, and large language models benefit from thinking step by step (Kojima et al., 2022). We should thus expect advanced AIs to have the capacity to reason, and to sometimes do better at tasks when thinking for a while. If we want to understand their behavior in novel situations, we need to understand how reasoning in neural networks (NNs) works.

Of particular relevance to AI alignment, sometimes internal reasoning is for the purpose of furthering a goal. Hubinger et al. (2019) term “mesa-optimizers” neural networks that have learned to explicitly, in their internal reasoning, pursue some goal, and warn that it may not be the same as the training goal (Di Langosco et al., 2022; Shah et al., 2022).

This paper presents a 1.28M parameter recurrent neural network (RNN) that clearly benefits from extra pondering time, towards a concrete goal. Following Guez et al. (2019), we train a convolutional LSTM to play Sokoban, a difficult puzzle game that is still a benchmark for planning algorithms Peters et al. (2023) and reinforcement learning (Chung et al., 2024).

We believe this NN is small enough, and its behavior is straightforward enough, that mechanistic interpretability techniques can succeed at reverse engineering it. Yet, at the same time, it performs reasoning in a relatively complex environment, in a way that benefits from extra thinking time. Thus, it is a useful first step towards understanding reasoning in complex neural networks.

Our contributions are:

- We replicate and open-source a RNN that improves performance from thinking-time (Guez et al., 2019).

Figure 1. Top: Proportion of medium-difficulty validation levels solved (out of a sample of 5120), as a function of environment transitions used in training. Each curve displays the DRC model for a particular number of forced thinking steps at the beginning of the episode, plus a ResNet baseline. Bottom: estimate of the planning effect, the pointwise maximum curve minus the minimum. We can see that thinking arises early on in training, and steadily increases as training progresses.
We closely follow the setup from Guez et al. (2019), which introduced the Deep Repeating ConvLSTM (DRC) recurrent architecture and trained it with reinforcement learning (RL) to play Sokoban. We make the trained networks available before publication.

Environment. Sokoban is a grid puzzle game with walls, floor, movable boxes and target tiles. The player’s goal is to push all boxes onto a target, while the walls make it difficult. We use the Boxoban dataset (Guez et al., 2018), consisting of $10 \times 10$ procedurally generated Sokoban levels, each with 4 boxes and targets. The edge tiles are always walls, so the playable area is $8 \times 8$. Boxoban separates levels into train, validation and test sets, with three difficulty levels: unfiltered, medium and hard. In this paper, we use unfiltered-train (900k levels), unfiltered-test (1k levels) and validation-medium (50k levels) sets.

The observations are $10 \times 10$ RGB images, normalized by dividing by 255. Each type of tile is represented by a pixel of a different color (Schrader, 2018). See Figure 5 for examples. The player can take actions (Up, Down, Left, Right). The reward is -0.1 per step, 1 for putting a box on a target, -1 for removing a box, and 10 for finishing the level. To avoid strong time-correlations during learning, episodes reset at a length between 91 and 120 time steps, chosen uniformly at random.

**DRC**($D, N$) architecture. Guez et al. (2019) introduced the Deep Repeating ConvLSTM (DRC), whose core consists of $D$ convolutional LSTM layers with 32 channels and $3 \times 3$, each applied $N$ times per time step. Our DRC($3, 3$) has 1.28M parameters. Diverging from Guez et al. (2019), we make this architecture residual, so each LSTM layer adds its output to a residual stream instead of feeding directly into the next layer. Before the LSTM core, a (conv, relu, conv) sequence encodes the observation with $4 \times 4$ filters. Unlike the original ConvLSTM (Shi et al., 2015), the input to each layer of a DRC consists of several components:

- The encoded observation is passed to each layer.
- A residual stream that include the hidden state of each ConvLSTM layer, which gets added to the residual stream and serves as an input to each layer.
- To allow spatial information to travel fast in the ConvLSTM layers, we apply pool-and-inject: we max- and mean-pool the previous step’s hidden state and linearly combine them before feeding them into the LSTM.
- To avoid convolution edge effects from disrupting the LSTM dynamics, we also feed in a $12 \times 12$ channel with zeros on the inside and ones on the boundary. Unlike the other inputs, this one is not zero-padded to keep the output the same size.

**ResNet architecture.** This is a convolutional residual neural network, also from Guez et al. (2019). It serves as a non-recurrent baseline, which cannot think for a variable number of steps but is nevertheless good at the game. The ResNet consists of 9 blocks, each with $4 \times 4$ convolutional filters. The first two blocks have 32 channels, and the others have 64. Each block consists of a convolution, then two (relu, conv) sub-blocks which each split off and are added back to the trunk. The ResNet architecture contains 3.07M parameters.

**Value and policy heads.** After the image processing, a linear layer projects the flattened spatial output into 256 channels. We then apply a ReLU, and two different affine layers: one for the actor (policy) and one for the critic (value function).

**RL training.** We train each network for 256 million environment steps using IMPALA (Espeholt et al., 2018). For each training iteration, we collect 20 transitions on 32 actors using the network parameters from the previous iteration and simultaneously take a gradient step. We use a discount rate $\gamma = 0.97$, V-trace $\lambda = 0.97$. The value and entropy losses are multiplied by 0.25 and 0.001, respectively. We use the Adam optimizer with a learning rate $4 \cdot 10^{-4}$, clipping the gradient norm to $1.5 \cdot 10^{-4}$. Our hyperparameters are mostly the same as Guez et al. (2019); see Appendix A for the full list.

**A* solver.** To obtain optimal solutions to each Sokoban puzzle, we used the A* search algorithm. The heuristic was the sum of the Manhattan distances of each box to its nearest target. Solving a single level on 1 CPU takes anywhere from a few seconds to 15 minutes.

3. **Quantitative behavior analysis**

Are these neural networks good at playing Sokoban? The ResNet (at 1B steps) is able to solve 95.8% of unfiltered test levels, while the DRC (at 250M steps) can only solve...
3.1. Does thinking time improve performance?

Does our trained DRC(3, 3) subject replicate the thinking time effect from Guez et al. (2019)? We evaluate the DRC on 5000 medium-difficulty validation levels. To give the DRC \( n \) steps to think, we repeat the initial environment observation \( n \) times while advancing the DRC hidden state, then let the DRC policy act normally. Figure 2 shows the DRC’s performance for \( n \in \{0, 2, 4, 6, 8, 12, 16, 32\} \) steps, as well as the ResNet baseline. The DRC’s solving rate improves from approximately 50% to 58% with more thinking steps, with peak performance at 8 steps. The effect is about twice as strong as the DRC(3, 3) by Guez et al. (2019, Figure 6), which improves the solving rate only by about 4.5%, though that network started at a \( \sim 78\% \) solve rate for medium validation levels (see their Figure 6).

3.2. Planning emerges early on in training

When does the DRC learn to benefit from thinking steps? Figure 1 shows the performance of the DRC on 5120 medium-validation levels (different from those of Figure 2). We can see that the DRC displays a strong planning effect by 50M environment steps, which gets stronger as training progresses. This suggests that planning behavior is reinforced by the training setup and loss.

3.3. Do difficult levels benefit more from thinking?

Intuitively, taking extra thinking time should help more with harder levels. The A* solver gives us two proxies for the difficulty of a level: the number of nodes it had to expand and the length of the optimal solution. We measured this by comparing the number of thinking steps at which a level is first solved with the proxies. The length of a level’s optimal solution seems to be a good proxy for its difficulty for the network: Figure 3 shows the average optimal length of episodes first solved at \( n \) thinking steps, and it clearly trends up.

The number of expanded nodes is not a good proxy for the difficulty the DRC has with a level, and displayed in Figure 7. We speculate that this is because the DRC has much better heuristics than the one we use for A*.
Planning in a RNN that plays Sokoban

3.4. What happens when we make the DRC think?

Figure 2 shows that, when artificially made to think, the DRC can solve many more levels. How does that happen? Does it have any effects other than solving more levels?

Thinking makes the DRC “patient.” One hypothesis is that, without the extra thinking steps, the DRC puts some boxes into targets without realizing that that makes the puzzle unsolvable. This makes sense based on the training signal, which penalizes every step it takes to solve the puzzle – it need not be “irrational” for the training process to push the DRC to be “impatient” and solve part of the puzzle on-the-go if that lets it solve puzzles fast enough.

For $n$ thinking steps, Figure 4 plots the average number of steps (time-to-box) until each of 4 boxes gets on a target. We discard unsolved levels at every $n$ so this is well-defined. The results support the hypothesis: levels that are solved at 8 thinking steps but were not solved with 0 thinking steps have their time-to-box go up for boxes 1–3 but down for box 4. In contrast, on average over all levels, times go down for all boxes.

Thinking often improves solutions for already-solved levels, but is sometimes harmful. Is an increased solution rate the only benefit of artificial thinking, or does it increase returns in other ways? Table 1 breaks down the 5000 medium-difficulty levels into types according to the behavioral effect of adding 8 steps of thinking. The breakdown shows two major benefits of thinking time: the DRC solves some levels that it did not previously solve, but also many levels that were already solved previously increase in return. The only way to improve undiscounted returns on solved levels is to take fewer steps, which matches Figure 4.

Of the levels that the DRC solved both when thinking and not thinking (58.4%), about 84.3% had a higher return, and remaining had the same or worse return.

Overall, extra thinking steps were not always good, but often they let the DRC solve more levels or improve return on already solved levels.

4. Case studies: how does extra thinking change DRC behavior?

Let us examine an example level from each of three categories from Table 1: a solved, previously unsolved level; an unsolved, previously solved level; and a level on which the DRC improved its return with extra thinking.

Thinking lets the DRC solve, Figure 5 (a). In the no-thinking condition, the DRC first pushes $A$ down then right onto position $x$. This locks the puzzle, and the DRC jitters close to $x$ for the rest of the level. In contrast, after thinking, the DRC puts the other boxes onto their targets first before coming back and dealing with box $A$.

Thinking speeds up solving, Figure 5 (b). In the no-thinking condition, the DRC starts by going right to push box $C$ onto target $c$, which does not lock the puzzle but makes it much longer. After doing that, it oscillates left and right a total of 4 times, before making its way towards position $z$ to continue solving the puzzle. After making it to $z$, it does a loop around it, before going on to target $b$ and to the rest of the level: push $A, B$ to $a, b$, then get out again through $z$, loop around the bottom and put $D$ onto $d$.

In contrast, the thinking-condition DRC starts left towards position $z$. It also does a loop around $z$ before continuing towards the right to put boxes $A, B$ onto $a, b$. Then it also exits left, loops towards the bottom, then pushes $C, D$ onto $c, d$.

Thinking slows down solving, Figure 5 (c). In the no-thinking condition, the DRC makes a beeline to position $y$, then pushes targets $A, B, C, D$ into their corresponding targets. In the thinking condition, it first makes its way to $d$ (possibly intending to push box $C$), but then realizes its mistake and goes to $y$. After wasting steps, the DRC then does the same sequence of moves as in the no-thinking condition.

4.1. Hypothesis: the DRC ‘paces’ to get thinking time

One hypothesis for why the DRC network learns to think during training is that the DRC sometimes moves around in
cycles (without touching any box) until it comes up with a plan to solve the episode. This comes up randomly during training, and is reinforced because the final return is larger. We see qualitative evidence for this ‘pacing’ behavior when reviewing replay videos, for example in the second case study from Section 4.

We measured this quantitatively. For solved levels, we compare the amount of time-steps that extra thinking steps compared to not-thinking, to a proxy for whether the DRC is naturally thinking by pacing. The proxy is the number of time-steps taken to complete the longest cycle in the sequence of states that is an episode. We plot this in Figure 6 (left), and see a faint correlation, with some lines of points going up and to the right. This is overall very weak evidence for the hypothesis, or perhaps the proxy is wrong.

In Figure 6 (right) we plot the cycle length compared to the number of planning steps for which the return peaks. We see increases in the average and in each quantile, but there are enough outliers to make this also weak.

Figure 6. Data points are levels which are solved both at 0 and 8 thinking steps. The y axis displays the longest cycle in states the DRC visits during evaluation (no thinking). Left: x axis displays the net reduction in episode steps after artificial thinking. The cycles are longer for levels that benefit from a higher reduction. We can see lines of points going up and to the right, so a subset of data correlates very well, but unless this is a natural category the effect is very small overall. Right: x axis displays the number of thinking steps at which a particular episode’s return peaks, as a proxy for how much the DRC benefitted from thinking in it. The longest cycles per episode are higher as measured by both the mean (blue line), and quantiles (boxes). However, the highest points are basically the same across all times.

We have replicated and open-sourced a recurrent network which learns to think for extra steps during training. We have described its behavior and believe understanding how it works is of great interest to the community.

Figure 5. Case study of three levels demonstrating different behaviors after 8 thinking steps. Videos available at [anonymized], though see Section 4 for a description of the DRC’s behavior.

5. Related work

World models. Several attempts to find world models in NNs with interpretability, (Li et al., 2023; Wijmans et al., 2023, for example).

Goal misgeneralization. From the alignment perspective, AIs optimizing monomaniacally for a goal have been a concern for a long time (Russell, 2019, e.g. ). In a machine learning paradigm, the goal of the training system doesn’t necessarily get optimized, and instead the NN may optimize for something else entirely: be a mesa-optimizer (Hubinger et al., 2019) - Mesa-optimizers an alignment concern (Hubinger et al., 2019; Ivanitskiy et al., 2023)

Chain-of-thought faithfulness. LLMs use chain of thought, are they faithful to it or do they think about the future in other ways? One could hope that they do all of their long-term reasoning in plain English, and thus any unintended (by humans) consequences of their actions can be easily monitored (Pfau et al., 2024; Scheurer et al., 2023)

Reasoning neural network architectures. Many papers try to make the NNs think artificially (Bansal et al., 2022; Graves, 2016).

Ethical treatment of AIs. Do they experience pleasure/pain? (Tomasik, 2015; Schwitzgebel & Garza, 2015) What does it mean for a NN to want something? We might not care about a giant lookup table, but maybe NNs that have expectations about the world are moral patients (Daswani & Leike, 2015).

6. Conclusion

We have replicated and open-sourced a recurrent network which learns to think for extra steps during training. We have described its behavior and believe understanding how it works is of great interest to the community.

References


Planning in a RNN that plays Sokoban


Tomasik, B. A dialogue on suffering subroutines. , 2015.


A. Training hyperparameters

Weight initialization. We initialize the network with the Flax (Heek et al., 2023) default: normal weights truncated at 2 standard deviations, and scaled to have standard deviation $\sqrt{\frac{1}{\text{fan in}}}$. Biases are initialized to 0. The forget gate of LSTMs has 1 added to it (Jozefowicz et al., 2015). We initialize the value and policy head weights with orthogonal vectors of norm 1.

Surprisingly, this makes the variance of these unnormalized residual networks decently close to 1.

Adam optimizer. As our batch size is fairly small, $\beta_1 = 0.9$, $\beta_2 = 0.999$. The denominator epsilon is $\epsilon = 1.5625 \cdot 10^{-10}$.

L2 regularization. In the training loss, we regularize the policy logits with L2 regularization with coefficient $1.5625 \times 10^{-5}$. We regularize the actor and critic heads’ weights with L2 at coefficient $1.5625 \times 10^{-7}$.

Why is our gradient clipping and epsilon so small? The original IMPALA implementation, as well as Huang et al. (2023), sum the per-step losses. We instead average them for more predictability across batch sizes, so we had to scale down some parameters.

Software. We base our IMPALA implementation on Cleanba (Huang et al., 2023). We implemented Sokoban in C++ using Envpool (Weng et al., 2022) for faster training, based off of gym-sokoban (Schrader, 2018).

B. Extra figures and tables

![Figure 7. Number of thinking steps at which the level is solved vs. number of nodes that A* needs to expand to solve it.](image)

Table 2. Success rate and return of DRC and ResNet on a random sample of unfiltered test set at various training environment steps.

<table>
<thead>
<tr>
<th>TRAINING ENV STEPS</th>
<th>RESNET</th>
<th>DRC</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>SUCCESS RATE</td>
<td>RETURN</td>
</tr>
<tr>
<td>50M</td>
<td>67.0</td>
<td>-0.23</td>
</tr>
<tr>
<td>250M</td>
<td>91.2</td>
<td>7.81</td>
</tr>
<tr>
<td>1B</td>
<td>96.3</td>
<td>9.52</td>
</tr>
</tbody>
</table>