Long-term and future-oriented planning with Sokoban

Pascal Perle[*]
Fakultät für Mathematik und Informatik
Universität Heidelberg
`nr231@stud.uni-heidelberg.de`
`https://github.com/pperle`

Jakob Weichselbaumer[†]
Fakultät für Mathematik und Informatik
Universität Heidelberg
`weichselbaumer@stud.uni-heidelberg.de`
`https://github.com/ip193`

September 30, 2019

1 Introduction

Long-term and future-oriented planning tasks are demanding problems in the field of reinforcement learning, for which few generally effective algorithms are known. These problems are difficult because a single action can have a significant impact on the outcome or even the solvability of a game at a later stage. Combinatorial planning games are often too large to be solved with exhaustive search approaches, as the number of decision combinations generally increases exponentially with the number of time steps.

This project, as part of the lecture "Advanced Machine Learning", reviews and analyses the application of four reinforcement learning algorithms to the puzzle game "Sokoban". The source code for this project is available at `https://github.com/ip193/AML-Sokoban`.

1.1 Sokoban

Sokoban is a puzzle/planning game, in which the player has to push boxes onto given locations. Each box can be moved to any destination; there are no predefined target fields on which a particular box should be moved. The boxes can only be pushed and not pulled by the player; it is not possible to move several boxes at the same time. Some actions are not reversible, and mistakes can make the puzzle unsolvable, e.g. when a box is pushed into a corner. These restrictions force the player to plan several moves in advance. An example of a Sokoban game can be seen in Figure 1.

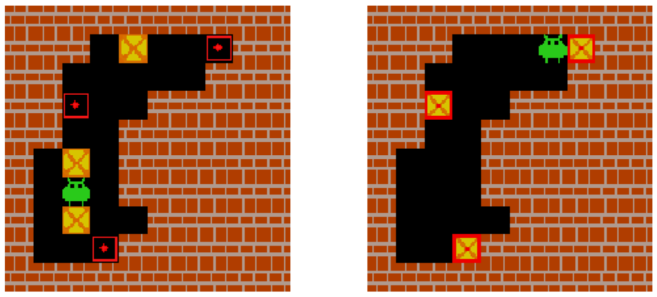

(a) start state (b) end state

Figure 1: start and end state of an exemplary Sokoban game

Sokoban puzzles are hard to solve as they have sparse rewards, big search graphs with cycles and have been shown to be PSPACE-complete [1]. Due to these characteristics, random actions have a very low probability of success. This makes solving Sokoban puzzles an interesting area of research.

[*]Chapters until Stochastic Synapse Reinforcement Learning (until p. 13), Appendix A and Appendix B
[†]Chapters on Stochastic Synapse Reinforcement Learning (from p. 13) and Appendix C

Since random game states may not be solvable (e.g. a box is stuck in a corner), game states are created by starting from the solved state and making random reverse moves (pulling boxes). For the implementation of the algorithms the Sokoban environment for OpenAI Gym [2] is used. This environment returns a reward for every action, -0.1 for every performed step, 1 when a box is pushed onto a target, -1 when a box is pushed off a target and 10 when all are boxes on the targets.

The complexity of the game increases with the number of boxes (= b) and the number of spaces that the player can move to (= s) according to Equation 1. This means that a game with 3 boxes and 21 spaces (e.g. in a game with the grid size of 7×7) has 23,940 possible states. But a game with about double the amount of spaces (e.g. in a game with the grid size of 10×10) has 447,720 possible states which is more than 18 times more states. The amount possible game states increases even further when more boxes are added to the game, see Figure 2. For example, 2,500,000 possible game states are reached for a game with 4 boxes when there are only 38 spaces, for 3 boxes when there are 64 spaces and for 2 boxes there have to be 172 spaces the player can move to.

$$s\binom{s-1}{b} \tag{1}$$

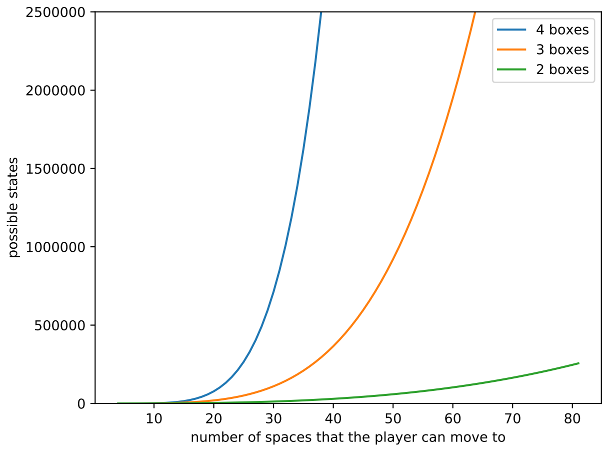


Figure 2: shows the growth of the possible game states according to $s\binom{s-1}{b}$, Equation 1

In order to compare the different approaches a test dataset of 100 games for each room type has been created, see Table 1. The generated games can be seen in Appendix A.

Table 1: Test dataset for different room types.

Room type	grid size	# boxes	comment
Sokoban-small-v0	7x7	2	
Sokoban-small-v1	7x7	3	
Sokoban-v0	10x10	3	
Sokoban-v1	10x10	4	as used in [3]
Boxoban (medium)	10x10	4	published by [4]
Boxoban (hard)	10x10	4	published by [4]

2 Approaches

Four different approaches were tested to solve the game.

2.1 Deep reinforcement learning and search

In "Solving the Rubik's cube with deep reinforcement learning and search" [3] the authors suggest to combine deep learning with path finding methods to solve a Rubik's cube and other puzzle games like Lights Out and Sokoban.

The current game state is entered into a deep neural network, which outputs the cost (necessary steps) to reach the goal. After training this so-called cost-to-go function, it is used as a heuristic to solve the puzzles with weighted A* search. This combination of the neural network and weighted A* search is named DeepCubeA by the authors.

2.1.1 data generation

The authors of [3] used a Sokoban environment with a 10×10 grid containing four boxes. The data given to the cost-to-go function is of size 400 and contains four binary vectors of size 100 each that represent the position of the agent, boxes, targets and walls.

Generating a Sokoban environment of size 10×10 with three boxes is computationally expensive; in the paper the authors used six GPUs in parallel for data generation. To speed up data generation, the given Sokoban implementation has been adapted by adding a constraint for the maximum number of steps to solve the game and return of steps to solve the game. To further speedup data generation, parts of the code have been implemented using Cython [5].

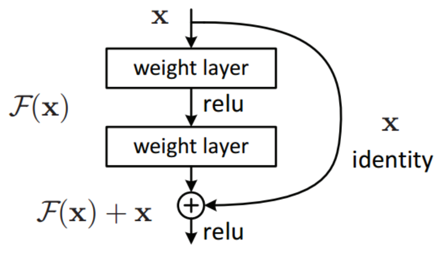
To speed up learning, a smaller grid size of 7×7 with two boxes was chosen, this speeds up data generation by 100x (see Table 2). The room type "adapted 'Sokoban-small-v0' with backtracking" uses the adapted implementation for data generation and only creates rooms that take a maximum of 8 steps to solve and also returns the steps in order to solve the game. This is needed in order to train the cost-to-go function. Data augmentation was used to generate eight times more data by mirroring images and rotating them 90 degrees.

Table 2: Comparison of time taken to generate 100 Sokoban games.

Room type	time to generate 100 rooms
'Sokoban-v0' (10×10 with three boxes)	308 s
'Sokoban-small-v0' (7×7 with two boxes)	7 s
adapted 'Sokoban-small-v0' with backtracking	3 s

2.1.2 cost-to-go function

Instead of using a lookup table that is unsuitable for puzzles with large state spaces such as Sokoban, a deep neural network is trained to minimize the mean square error of the estimated cost-to-go value. The architecture for the deep neural network can be seen in Table 3, the architecture of a residual block is shown in Figure 3.



x
weight layer
relu
weight layer
$\mathcal{F}(\mathbf{x})$
x
identity
$\mathcal{F}(\mathbf{x}) + \mathbf{x}$
relu

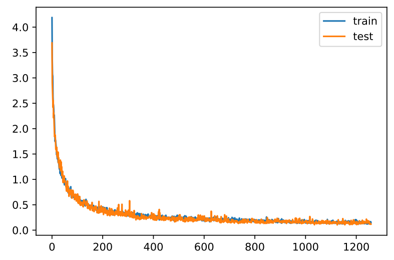
Figure 3: Residual Block as described in [6]

Layer (type)	Input Shape	Output Shape
Fully Connected	196	5,000
Batch Normalization	5,000	5,000
Rectified Linear Unit (ReLU)	5,000	5,000
Fully Connected	5,000	1,000
Batch Normalization	1,000	1,000
Rectified Linear Unit (ReLU)	1,000	1,000
Residual Block	1,000	1,000
Residual Block	1,000	1,000
Residual Block	1,000	1,000
Residual Block	1,000	1,000
Fully Connected	1,000	1

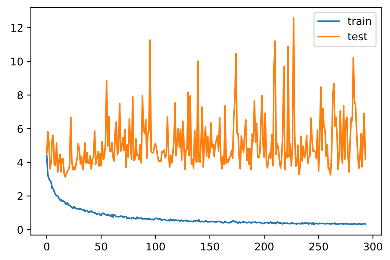
Table 3: Architecture of cost-to-go function

The network was trained with a batch size of 64 and the ADAM optimizer was used to minimize the mean square error. When training with a fixed number of states (= pregenerated data), the model already overfitted after a few epochs. To prevent this, new data is generated for each epoch, so that training with the same data is not repeated.

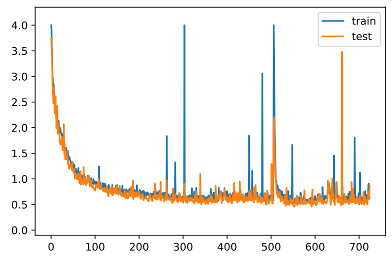
The model had to be altered as the batch normalization layers during evaluation caused for wrong predictions when the batch size was not 64. Figure 4a shows the results after 24h training, a minimum loss of 0.2 was reached. When trying to evaluate a single state with this model, the predictions were always very poor. In Figure 4b the same model can be seen but with a test batch size of 1. Here the problem is clearly visible, the training loss drops while the test loss stays constant. The model was altered and all batch normalization layers where removed. After 13 hours, 750 epochs with



(a) Training and testing loss with a batch size of 64 each. The model was constructed as presented in the paper (see Table 3). After 24 hours the loss dropped to 0.2.



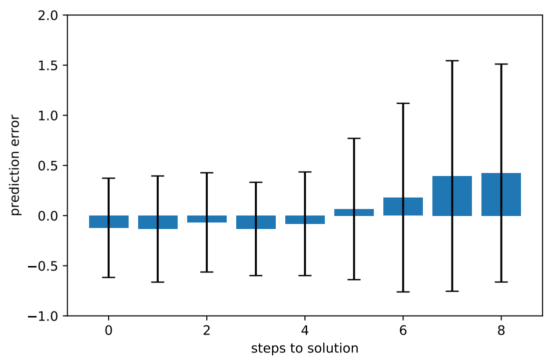
(b) Training loss with a batch size of 64 and testing loss with a batch size of 1. Same model architecture as in Figure 4a. It is clearly visible that the testing loss stays constant.



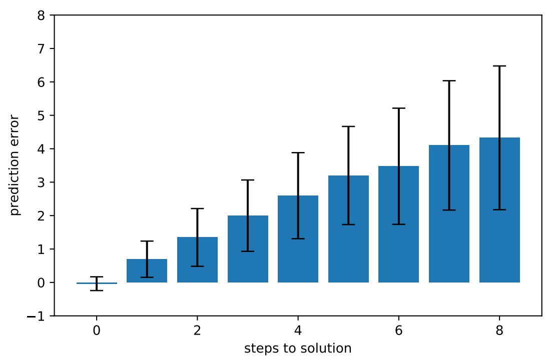
(c) Training loss with a batch size of 64 and testing loss with a batch size of 1. Same model architecture as in Table 3 but without the normalization layers. Training is more unstable than in Figure 4a, with random spikes in the loss. But in contrast to Figure 4b the testing loss drops to 0.5.

Figure 4: Loss of different models with different batch sizes. The x-axis shows the number of epochs and the y-axis shows the loss.

100,000 states each, the minimum loss reached was 0.5 with a test batch size of 1, see Figure 4c. Compared to Figure 4a the training without the batch normalization layers is much more unstable and the loss does not drop as low as before.



(a) Prediction error on a 7 × 7 grid with two boxes. For 4 steps and fewer to the solution the error is nearly 0 with a standard deviation of 1.

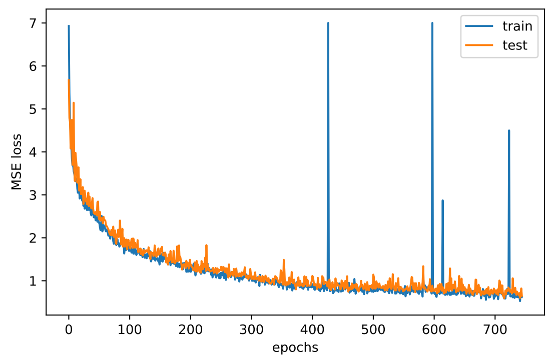


(b) Prediction error on a 7 × 7 grid with three boxes. The model is the same as for Figure 5a, trained on a 7 × 7 grid with only two boxes. The mean error is much higher compared to the model in Figure 5a.

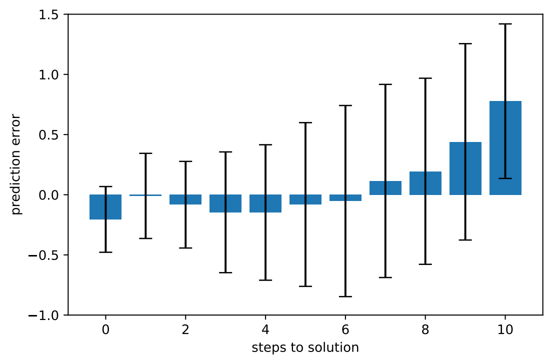
Figure 5: Prediction error (absolute error of the heuristic) of 7 × 7 grid compared against steps (=distance) to solution.

The model from Figure 4c was trained on a 7 × 7 grid with two boxes. The prediction error (prediction - real value) plot Figure 5a (7 × 7 grid with two boxes) shows that for 0 to 4 steps the mean of the prediction error is near 0 and the standard deviation is about 1. These starts to increase with a maximum error of 0.4 and a standard deviation of 2 at 8 steps away from the solution. Trying to use the same model on a 7 × 7 grid with three boxes (see Figure 5b) shows the same prediction error and standard deviation only for 0 steps away from the solution. The prediction error and standard deviation increases with every step further away from the solution.

For comparison another model was trained on a 7 × 7 grid with three boxes and a maximum of 10 steps. The models loss, see Figure 6a, is comparable with the loss the model seen in Figure 4c. After 750 epoch, 28 hours, of training the loss reached a minimum of 0.56. When comparing the steps to solution prediction error of the model trained on a 7 × 7 grid with three boxes, see Figure 6b, to the model trained on a 7 × 7 grid with two boxes, Figure 5b, it can be seen that the error is much lower. This should translate into a better heuristic for A* search.



(a) Training loss with a batch size of 64 and testing loss with a batch size of 1. Same model as in Figure 5b.



(b) Prediction error on a 7×7 grid with three boxes. Lower error compared to Figure 5b.

Figure 6: Loss and prediction error of the model trained on a 7×7 grid with three boxes.

A third model was trained on a 10×10 grid with two boxes, the architecture is the same as described in Table 3, except that the first layer has an input of 400 (=10*10*4). Two boxes were chosen as the amount of possible states is much lower compared to a 10×10 grid with the boxes, see Figure 2. The model was trained with newly generated data with a maximum step distance of 13 after every epoch. After 75 hours of training, 1700 epochs with 100,000 states each, a minimum loss of 5.28 with a test batch size of 1 was reached, see Figure 7. Compared to Figure 4c the loss is much more stable but does not drop as low. The training was stopped after 75 hours as there was not much improvement in the loss.

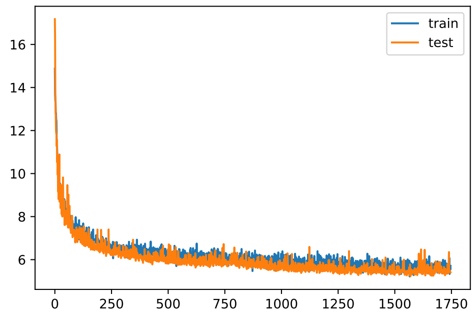
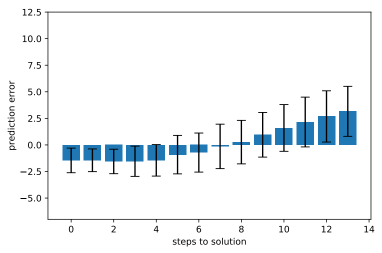
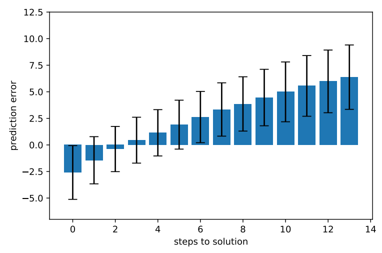


Figure 7: loss of the "10×10 grid with two boxes" model with a batch size of 1

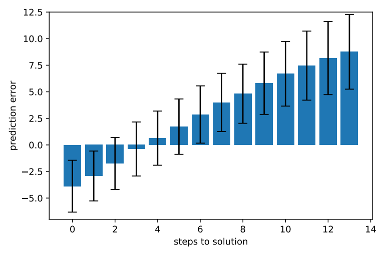
When comparing the prediction error of the model trained on a 10×10 grid (Figure 8) to the model trained on a 7×7 grid (Figure 5 and Figure 6b) the higher loss of the former model is apparent.



(a) Prediction error on a 10×10 grid with two boxes. For 7 steps and fewer to solution the average predicated value is lower than the actual value with a standard deviation of about 2.5. For 8 steps or more to the solution the predicated distance is higher with a standard deviation of about 5.



(b) Prediction error on a 10×10 grid with three boxes. The model is the same as for Figure 8a, trained on a 10×10 grid with only two boxes. The diagram looks similar to the one in Figure 8a shifted to the left.



(c) Prediction error on a 10×10 grid with four boxes. The prediction error is similar to the prediction error as in Figure 8b but with a much higher error rate.

Figure 8: Prediction error (absolute error of the heuristic) of 10×10 grid compared against steps (=distance) to solution

2.1.3 A* search

Once a cost-to-go function has been learned, it can be used as a heuristic to search for a path between the start state and the destination state.

In standard A* search the cost of a node x is determined by the function $f(x) = g(x) + h(x)$ where $g(x)$ describes the previous costs from the start node to x and $h(x)$ is the estimated cost (by the cost-to-go function (= heuristic)) from x to the target node.

The A* search keeps an "open" set from which it expands the node with the lowest cost (= $f(x)$ value) and expands the node. The algorithm starts with only the start node in the "open" set. Once a node has been expanded, that node is moved to the "closed" set, and any subnodes/children that are not already in the "closed" set or whose $g(x)$ is smaller than the $g(x)$ of the equivalent node in the "closed" set are added to the "open" set. Once the target node is removed from the "open" set, the algorithm terminates. The complete algorithm is shown in algorithm 1.

```
function a_star ($start\_position$, $weight = 1$);
Input  : Start position of the game and a weight for weighted A* search
Output : List of moves to solve the game or $False$ if no solution is found

$closeSet \leftarrow$ empty set
$openHeap \leftarrow$ empty heap
$cameFromMap \leftarrow$ empty map
$gScoreMap \leftarrow \{start\_position : 0\}$
$fScoreMap \leftarrow \{start\_position : heuristic(start\_position)\}$
$actionsMap \leftarrow$ empty map
$openHeap$.push($start\_position$, $fScoreMap[start\_position]$)

while $openHeap$ is not empty do
    $current$ = $openHeap$.pop() // node with lowest $f(x)$ value
    if $current == target$ then
        $steps \leftarrow$ empty list
        while $neighbor \in cameFromMap$ do
            $steps$.append($actions[neighbor]$)
            $neighbor \leftarrow cameFromMap[neighbor]$
        end
        return reversed($steps$)
    end
    foreach $action \in [up, right, down, left]$ do
        $gScoreTentative \leftarrow gScoreMap[current] + 1$
        $neighbor \leftarrow current$
        $neighbor$.step($action$)
        if $neighbor \in closeSet$ and $gScoreTentative >= gScoreMap[neighbor]$ then
            continue
        end
        if $neighbor \notin openHeap$ then
            $cameFromMap[neighbor] = current$
            $gScoreMap[neighbor] = gScoreTentative$
            $fScoreMap[neighbor] = gScoreTentative * weight + heuristic(neighbor)$
            $actionsMap[neighbor] = action$
            $openHeap$.push($neighbor$, $fScoreMap[neighbor]$)
        end
    end
end
return $False$
```

Algorithm 1: The algorithm for weighted A* search. A weight of 1 equals regular A* search.

One condition of A* search is that the heuristic is admissible [7, p. 94], which means it should never overestimate the cost to reach the goal. Overestimation causes the A* search to explore nodes that may have a poorer $f(x)$ value. This condition is not met as the trained cost-to-go function often overestimates the distance to the solution, see error bars in Figure 5 and Figure 8. This results in a not optimal search.

The second condition is a consistent heuristic [7, p. 95], where the estimated cost of reaching the goal from the current state is not greater than the step cost of reaching a neighboring state plus the estimated cost of reaching the goal from that neighbor, is also not met, see error bars in Figure 5 and Figure 8. Because of the non-consistent heuristic, it is not guaranteed that another algorithm expands fewer nodes than A* search. [7, p. 98]

The complexity of the A* search depends on the heuristics. In the worst case, the number of expanded nodes of a typical node is exponential to length of the shortest path $O((b^{\epsilon})^{d})$ (for constant step costs), where b^{ϵ} is the average number of successors per state and d is the solution depth (= length of the solution). The average number of successors per state (b^{ϵ}) can be calculated by Equation 2 where N is the total number of explored nodes. An optimal heuristic would have a value of $b^{\epsilon} = 1$. [7, p. 100-101]

$$N + 1 = 1 + b^{\epsilon} + (b^{\epsilon})^2 + \cdots + (b^{\epsilon})^d = \sum_{n=0}^{d} (b^{\epsilon})^n \tag{2}$$

The authors suggest in the paper to use weighted A* search since it trades potentially longer solutions for potentially less memory usage. This is achieved by weighting the previous costs from the start node to x with a factor between zero and one, the so-called path-cost coefficient λ. The function for weighted A* search is $f(x) = \lambda g(x) + h(x)$ with $\lambda \in [0, 1]$. The use of the weighted A* search was not necessary as with the efficient implementation used, even very long searches, never consumed more than 3 GB of memory.

2.1.4 Results

For each generated game DeepCubeA was able to find a solution, the solution paths can be found in Appendix A. This is due to A* search, which is always able to find a way between two points after exploring all possible states if one exists (= A* search is both complete and optimal) [7, p. 93]. For this reason, in addition to the number of steps required to solve the game, it is also interesting how many states the algorithm had to explore before finding a solution, see Table 4 and compare the effective branching factors, see Table 5. The implementation was able to check 100 states per second using a single core on a `Intel Xeon E3-1231 v3 @ 3.40GHz`, the slowest part was the Sokoban gym [2].

Table 4: avg. steps to solution and percentage of explored states needed to solve the test dataset for different room types. "7×7, 2 boxes" stands for the model trained on a 7×7 grid with 2 boxes.

Room type	avg. steps to solution	avg. % of explored states	avg. of possible states
Sokoban-small-v0 (7×7, 2 boxes)	11.38	17.34	1,674
Sokoban-small-v1 (7×7, 2 boxes)	16.62	22.85	10,323
Sokoban-small-v1 (7×7, 3 boxes)	16.62	17.06	10,323
Sokoban-v0	24.97	24.36	102,924
Sokoban-v1	29.10	38.26	865,870
Boxoban (medium)	46.42	29.22	1,119,188
Boxoban (hard)	55.55	36.76	1,110,871

The effective branching factor can be calculated, as described in subsubsection 2.1.3, once a solution has been found. This is one way to describe the quality of a heuristic. A lower effective branching factor does not guarantee that the heuristic finds a shorter solution, but it guarantees that it will not expand more nodes than a heuristic with a higher effective branching factor (= better efficiency) [7, p. 104]. As shown in Table 5 the effective branching factor of the trained models is between 1.22 and 1.52 which is about half of the maximal branching factor which averages about 2.7. The branching factor does not become worse with the increase of boxes, although it was only trained on 2 boxes, see column "7×7, 2 boxes" and "10×10, 2 boxes" in Table 5. This shows that the cost-to-go function is able to generalize. However, "7×7, 3 boxes" provides a better branching factor than "7×7, 2 boxes" on a 7×7 grid with 3 boxes.

Table 5: comparison of the effective branching factor calculated according to Equation 2; max. branching factor is the average amount of possible actions per state. The best value is in bold.

Room type	max. branching factor	7×7, 2 boxes	7×7, 3 boxes	10×10, 2 boxes
Sokoban-small-v0	2.69987	**1.52050**	-	-
Sokoban-small-v1	2.77278	1.51945	**1.48817**	-
Sokoban-v0	2.83565	-	-	**1.45281**
Sokoban-v1	2.87597	-	-	**1.49189**
Boxoban (medium)	2.71001	-	-	**1.27109**
Boxoban (hard)	2.71607	-	-	**1.22412**

Figure 9 shows the average available and explored states for each room type. It can be clearly seen that with Sokoban-v1 the number of available states increases rapidly. This implies that the search time also increases. When comparing Sokoban-v0 with Sokoban-v1, the average percentage of explored states only increases from 24.36 to 38.26 by 57%, but when looking at the numbers of explored states, they increase from 25,077 to 331,305 by 1,221%.

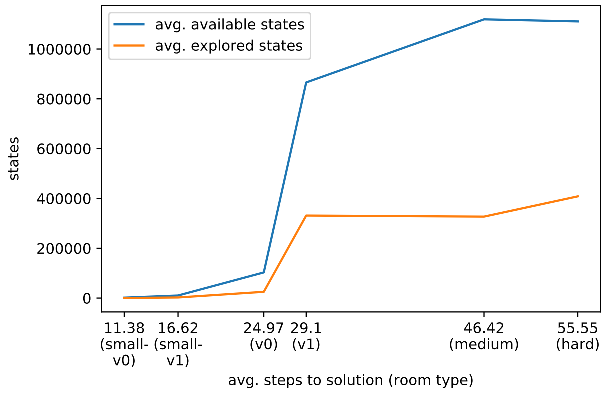
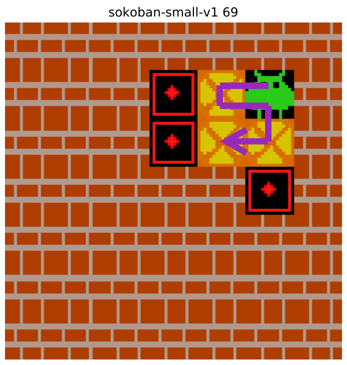
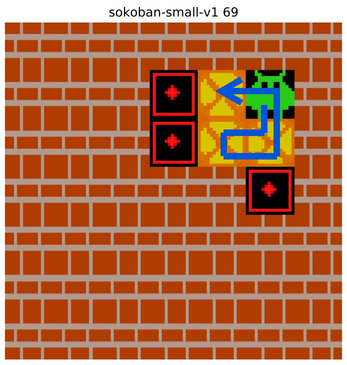


Figure 9: comparison of average available and explored states for each room type, for 7 × 7 with 2 boxes and 10 × 10 with two boxes

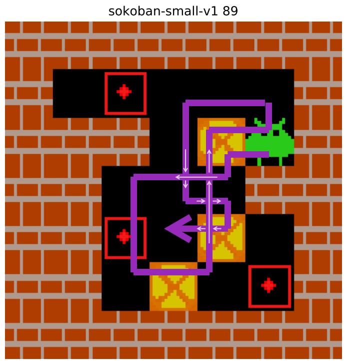
For the 100 games in the test dataset of room type "Sokoban-small-v1" the two different models returned 41 different solutions, 8 of them with different lengths, see subsubsection A.2.2. Two examples of solutions with different lengths can be seen in Figure 10. The average solution length, of 16.62, is the same for both models, see Table 4.



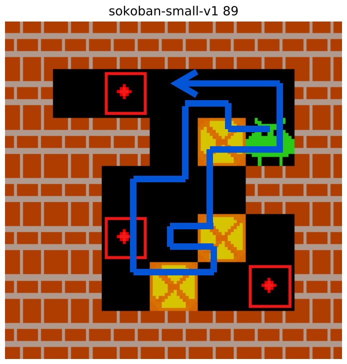
(a) Solution was calculated using the 7 × 7 with two boxes model. A total of 4 steps is need to solve the game.



(b) Solution was calculated using the 7 × 7 with three boxes model. One step more is needed to solve the game.



(c) Solution was calculated using the 7 × 7 with two boxes model. A total of 20 steps is need to solve the game.



(d) Solution was calculated using the 7 × 7 with three boxes model. One step less is needed to solve the game.

Figure 10: Two examples of solutions with different lengths.

2.2 DeepCubeA with Environment Model

The A* search and the cost-to-go function of DeepCubeA are combined with an environment model, which results in a model-based approach. In DeepCubeA, a model-free approach, observations are made from the environment directly. In a model-based approach, a model of the environment is learned to minimize the dependency on the real world (= the Sokoban environment). With an accurate environment model, the agent can generate all the necessary states by only using the environment model instead of performing the actions in the real world. Since dependence on the real world is minimal, sample efficiency (= speed) is improved, with a perfect model there is no dependence on the real world. In real applications, however, it is not always possible to have an accurate model of the real world. Even an imperfect model can considerably reduce the number of samples needed from the real world.

2.2.1 Environment Model

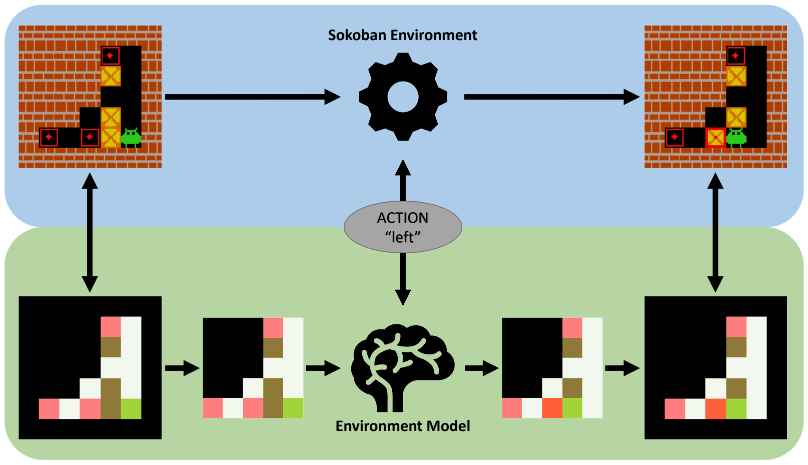


Figure 11: Comparison between the processes of the Sokoban environment and the environment model. The result is the same.

The environment model creates the next observation from the current observation and the given action. The initial input observation is created by retrieving the room_state from the environment which is provided by the Sokoban environment for OpenAI Gym [2]. This has the advantage that no convolution layers are needed to extract features from pixels. Another modification for the model to learn faster is to remove the outer row of walls (zeros in room_state), as these are constant in every Sokoban game, see Figure 11 for a visual representation of the algorithm.

Layer (type)	Input Shape	Output Shape
FC + PReLU	26	512
FC + PReLU	512	1,024
FC + PReLU	1,024	2,048
FC + PReLU	2,048	5,012
FC + PReLU	5,012	2,048
FC + PReLU	2,048	1,024
FC + PReLU	1,024	512
FC + PReLU	512	25

Table 6: Architecture of environment model, FC stands for Fully Connected.

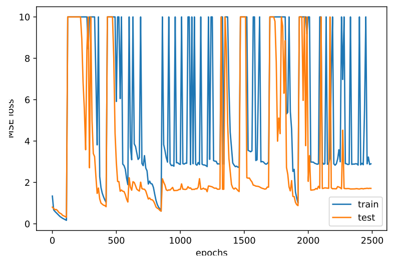
Layer (type)	Input Shape	Output Shape
Fully Connected	26	5,000
Parametric ReLU (PReLU)	5,000	5,000
Fully Connected	5,000	1,000
Parametric ReLU (PReLU)	1,000	1,000
Residual Block with PReLU	1,000	1,000
Fully Connected	1,000	25

Table 7: Architecture of environment model, with Residual Blocks.

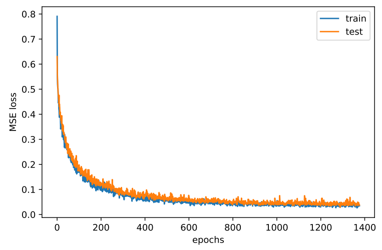
Three different model where trained. The first model uses the architecture seen in Table 6 and was trained to minimize the mean square error with the ADAM optimizer. After 2,500 epochs, 12 hours, of training the loss did not converge, see Figure 12a.
The second model uses the architecture seen in Table 7. Mean square error as the loss function and the ADAM optimizer were used to train the model. A minimal loss of 0.057, was reached after 1,350 epochs, 12 hours, of training, see Figure 12b.

The third model uses the same architecture and optimizer as the previous model but uses the L1 norm for the loss function instead. After 820 epochs, 9 hours, of training a minimal of 0.07 was reached, see Figure 12c.

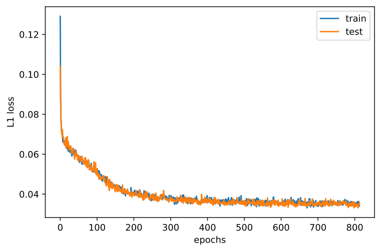
The data generation is the same as described in subsubsection 2.1.1.



(a) The loss did not converge even after a long time of training. The loss was clipped to a max of 10 in the diagram.



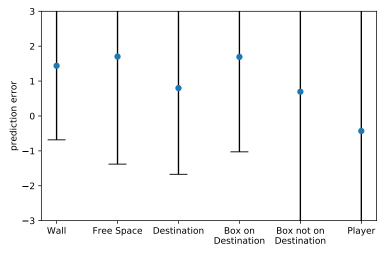
(b) Constantly falling loss, much better results compared to Figure 12a. Training was stopped after 12 hours.



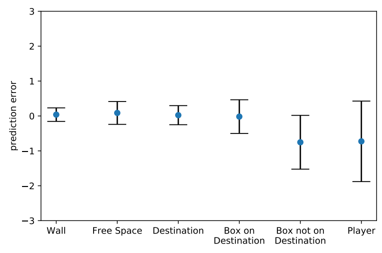
(c) Same architecture as Figure 12b, the loss did also drop but stayed constant after 400 epochs.

Figure 12: Training loss of three different environment models.

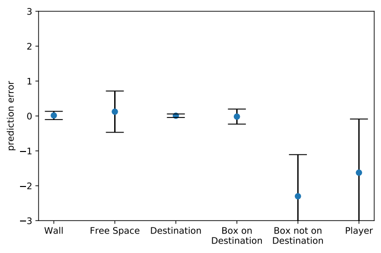
The absolute error of each field type (0=Wall, 1=Free Space, 2=Destination, 3=Box on Destination, 4=Box not on Destination, 5=Player) of the predicted observation can be seen in Figure 13. As expected, the model with the high loss, see Figure 12a, has a high absolute error for the field types. The second model has a much lower absolute error for all the field types. So are all predictions of the "Wall", "Free Space", "Destination" and the "Box on Destination" exact after rounding. The other two field cannot be predicted accurately. The third model improves the prediction accuracy of the "Wall", "Destination" and the "Box on Destination", see Figure 13c. The prediction accuracy of the other three field type is worse than in Figure 13b.



(a) The loss did not converge even after a long time of training. The loss was clipped to a max of 10 in the diagram.



(b) Constantly falling loss, much better results compared to Figure 12a. Training was stopped after 12 hours.



(c) Same architecture as Figure 12b, the loss did also drop but stayed constant after 400 epochs.

Figure 13: Prediction error of three different environment models.

2.2.2 Results

The model based on the architecture of Table 7 with mean square error is used as has the lowest prediction error. The values are rounded to integers between 0-5 and a border of walls (=0) is added around the output to produce a valid observation.

This implementation is twice as fast as the DeepCubeA implementation with the Sokoban environment at about 200 states per second. But when trying to solve "Sokoban-small-v1" from test dataset, none of the game could be solved. This is because the model is only able to accurately predict the next environment state with an accuracy of 56,11%, tested on 8,000 states.

2.3 Imagination augmented agents

In imagination augmented agents (I2A) [8] the action performed by the agent is the result of the model-based, where future trajectories are imagined, as well as the model-free path.

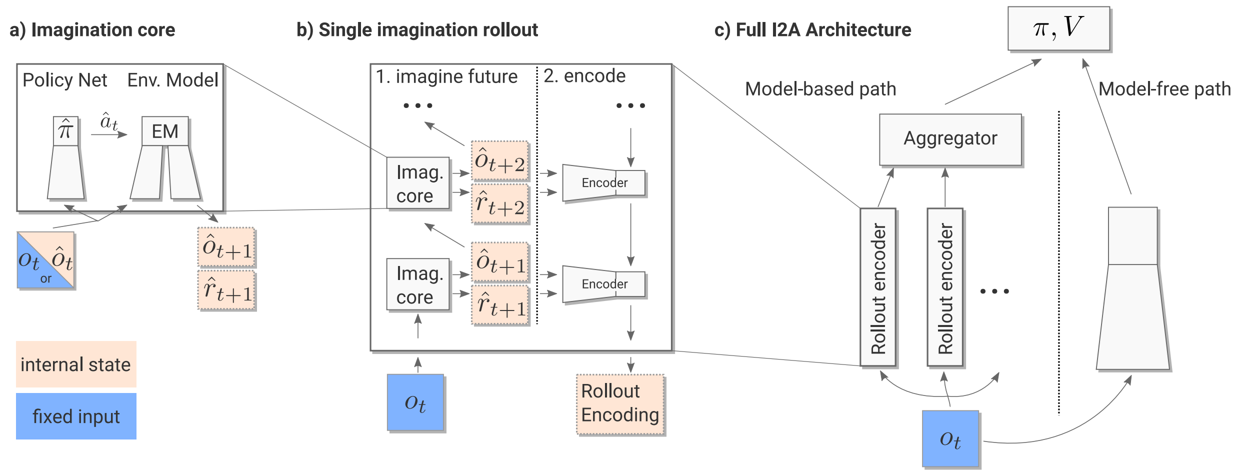


Figure 14: The architecture of imagination augmented agents.

2.3.1 Rollout Encoders

The model-based path consists of so-called rollout encoders; these rollout encoders contain the imaginative power of the agent. Rollout encoders consist of two parts, the "Imagine Future" and the "Encoder" part. The "Imagine Future" part consists of the "Imagination Core"; this "Imagination Core" provides the next state as well as a reward. The "Imagination Core" consists of a "Policy Network" and an "Environment Model". The "Environment Model" learns from all actions the agent has performed so far. It takes the information about the condition and the chosen action and imagines the future scenario as well as the reward under consideration of the past experiences. The imagined state of the first "Imagine Future" is used as input for the next "Imagine Future". This is repeated a fixed number of times, the authors suggest 5, in order to obtain a rollout consisting of a pair of states and rewards. An encoder (e.g. an LSTM) is used to encode this rollout into a fixed-size vector. These rollout encodings are in fact embeddings that describe the imagined path. An encoded rollout is created for each imagined path, this means for every possible action in the environment. Aggregators are used to aggregate these encoded rollouts.

The model-free agent is a standard advantage actor-critic (A2C) [9, p. 331-332].

2.3.2 Results

Layer (type)	Input Shape	Output Shape
Conv 2D (8x8 kernel, stride of 4) + ReLU	(3, 80, 80)	(32, 19, 19)
Conv 2D (4x4 kernel, stride of 2) + ReLU	(32, 19, 19)	(64, 8, 8)
Conv 2D (3x3 kernel, stride of 1) + ReLU	(32, 19, 19)	(64, 6, 6)
Fully Connected + ReLU	(64, 6, 6)	512
Fully Connected (Output 1/policy logits)	512	1
Fully Connected (Output 2/value function)	512	1

Table 8: Architecture of the advantage actor-critic (A2C).

The implementation is based on the code by https://github.com/higgsfield/Imagination-Augmented-Agents and was adjusted to work with the Sokoban environment for OpenAI Gym [2]. The Sokoban environment had to be adapted as the implementation of I2A expects tiles to have a size of 8x8 pixels.

After training the standard model-free agent with advantage actor-critic (A2C), for architecture see Table 8, for 200,000 epochs, 9 hours, the reward did not improve, see Figure 15. The reward is calculated as described in subsection 1.1. According to the authors in order to solve 7.72% of random games the agent was trained on 10,00,00,000 which would have taken about 4,500 hours (or 187.5 days). This was confirmed in an interview with a Deep Mind employee who said "students probably won't have the computing resources to train that architecture in a responsible time", see Figure 28. Since the hardware required to train this algorithm was not available, this approach was not pursued further. An implementation of Asynchronous Advantage Actor Critic (A3C) [10] could improve the training speed but not by a factor high enough to train a network in a few days without the appropriate hardware.

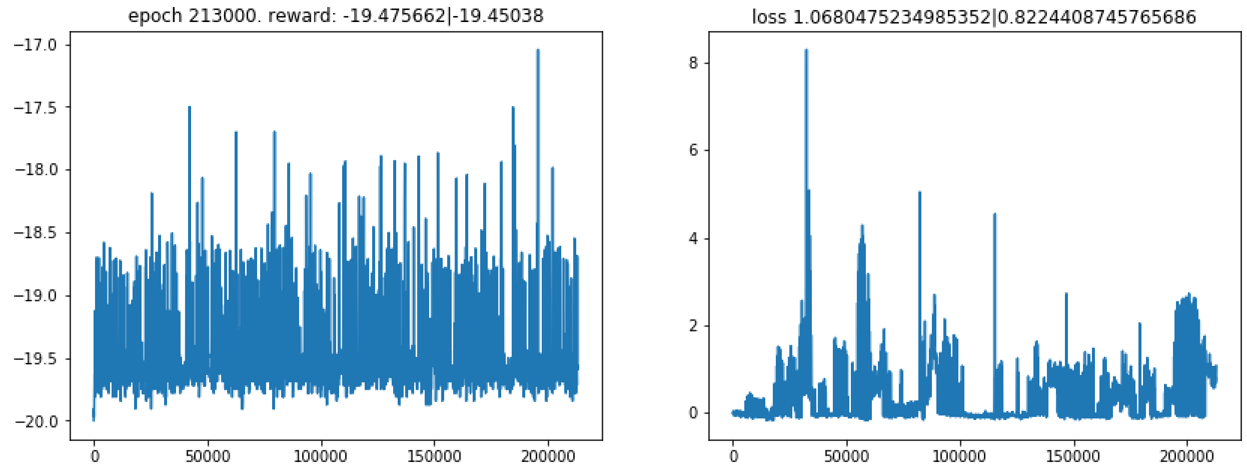


Figure 15: The reward achieved by the A2C and the loss of the value function, as the value is about -19.45 most of the time the loss is not too high.

3 Stochastic Synapse Reinforcement Learning

> Stochastic Synapse Reinforcement Learning (SSRL) is a reinforcement learning algorithm "for artificial neural networks (ANNs) to learn an episodic task in which there is discrete input with perceptual aliasing, continuous output, delayed reward, an unknown reward structure, and environmental change". [11]

SSRL was selected for its problem-agnostic design, its relative ease of implementation, and because its design specification (delayed reward, environmental change) appeared suitable for the Sokoban task.

This section will discuss the design of the algorithm, the training procedure used to adapt it to Sokoban, the experimental results of training, and further explorations that were made in light of those results.

Section 4 will discuss shortcomings of the algorithm and my implementation of its training and give suggestions for improvement and further research.

Finally, my correspondence with Prof. Hougen, a co-author of the SSRL paper [11], is included with permission in appendix C.

3.1 Algorithm design

3.1.1 Overview

$$w_{ij}(k) \sim \Psi(\mu_{ij}(k), \sigma_{ij}(k)) \tag{3}$$

In brief, the algorithm works by sampling neural network weights from a learned normal distribution (see 3) at each time step k in an episode and updating those distribution parameters (mean and standard deviation) at the end of the episode τ depending on

1) whether the performance in the episode using those sampled weights positively or negatively exceeded a past average reward

and

2) how much the sampled weights differed from the average values suggested by that weight's normal distribution at any given time step k.

Because the algorithm associates each weight in the network with its own normal distribution and updates its parameters independently for their contribution to the network's output at every time step, it can address the problem of credit assignment on a per-node, per-time-step level.

3.1.2 Detail

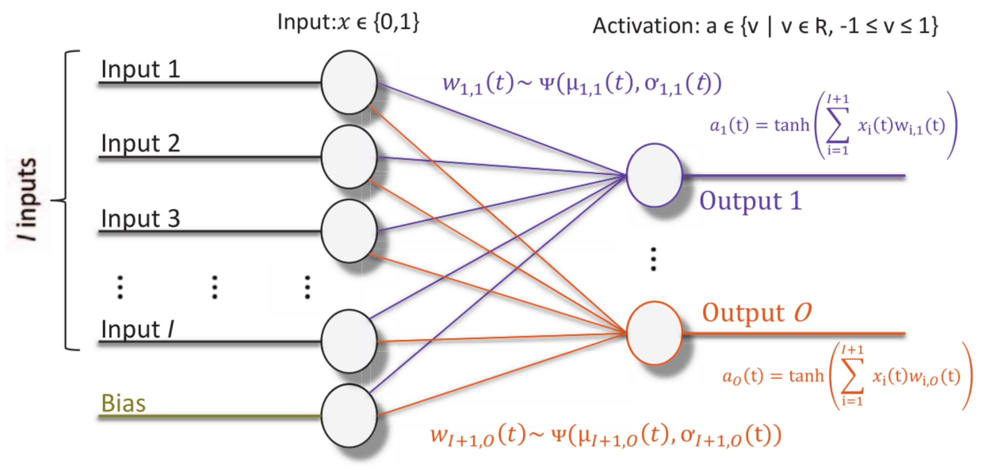

Input: $x \in \{0,1\}$
Activation: $a \in \{v \mid v \in R, -1 \le v \le 1\}$
I inputs
Input 1
Input 2
Input 3
Input I
Bias
$w_{1,1}(t) \sim \Psi(\mu_{1,1}(t), \sigma_{1,1}(t))$
$a_1(t) = \tanh\left(\sum_{i=1}^{I+1} x_i(t) w_{i,1}(t)\right)$
Output 1
Output O
$a_O(t) = \tanh\left(\sum_{i=1}^{I+1} x_i(t) w_{i,O}(t)\right)$
$w_{I+1,O}(t) \sim \Psi(\mu_{I+1,O}(t), \sigma_{I+1,O}(t))$

Figure 16: The single-layer architecture proposed in the paper. [11]

The architecture proposed by Shah and Hougen uses a neural network with only an input layer and output layer, as well as a bias neuron (figure 16). In order to keep track of the behavior of a neuron during an episode, the authors define the "eligibility trace" of a weight's parameters at a time step k according to equations 4 and 5. Network input is denoted by x.

$$e_{\mu,ij}(k) = x_i(k) \cdot (w_{ij}(k) - \mu_{ij}(k)) \tag{4}$$

$$e_{\mu,ij}(k) = x_i(k) \cdot (|w_{ij}(k) - \mu_{ij}(k)| - \sigma_{ij}(k)) \tag{5}$$

The update formula for μ_{ij} applied at the end of an episode τ is shown in equation 6 with parameters **learning rate** (η_μ, η_σ) and **decay** (d_μ, d_σ). Here, t is the final episode length, $r(\tau)$ is the sum reward collected in this episode, and $\overline{r}(\tau)$ is an "average reward" computed over a sliding window of rewards earned in past episodes. The update formula for σ_{ij} (equation 7) has an additional clamping step (8) to control the maximum level of exploration, to prevent excessive divergence of the algorithm's sampled weights.

$$\mu_{ij}(\tau+1) = \mu_{ij}(\tau) + \eta_\mu \cdot (r(\tau) - \overline{r}(\tau)) \sum_{k=1}^{k=t} e_{\mu,ij}(k) d_\mu^{(t-k)} \tag{6}$$

$$\widetilde{\sigma_{ij}}(\tau+1) = \sigma_{ij}(\tau) + \eta_\sigma \cdot (r(\tau) - \overline{r}(\tau)) \sum_{k=1}^{k=t} e_{\sigma,ij}(k) d_\sigma^{(t-k)} \tag{7}$$

$$\sigma_{ij}(\tau+1) = max(0.05, min(1, \widetilde{\sigma_{ij}}(\tau+1))) \tag{8}$$

Observe that μ_{ij} is increased at the end of an episode **if and only if** $(r(\tau) - \overline{r}(\tau))$ and $\sum_{k=1}^{k=t} e_{\mu,ij}(k) d_\mu^{(t-k)}$ have the same sign when inputs are positive. Otherwise, it is decreased (analogously for $\widetilde{\sigma_{ij}}$).

By updating both μ_{ij} and σ_{ij} separately, the algorithm can control its level of exploitation and exploration in search of an optimal policy - if weights very far from the mean are associated with higher sum reward, the standard deviation of the distribution is increased and the agent shifts its focus toward more exploration. If weights sampled on one side of the mean tend to do better than on another, the mean of the distribution is increased in that direction in order to increase the suitability of the current guess of the optimal policy.

3.1.3 Training

To train the algorithm, the Sokoban environment was initialized to a board state from which a solution state could be reached in $step\_distance$ steps. The beginning training case is $step\_distance = 1$. As soon as the algorithm is able to solve it consistently or a maximum number of training episodes have been reached, $step\_distance \leftarrow step\_distance + 1$ and the process repeats with the new $step\_distance$. Games are truncated after reaching $step\_distance \cdot 5$ steps to end obviously failed games at an appropriate time. Generating these states was not straightforward and proved to be a bottleneck in the implementation and success of this algorithm, as is described in section 4.2.1.

For simplicity, the game was played with a 7×7 board state and 2 boxes. The agent's neural network received the flattened board state as input.

Rewards were tallied at every step and delivered as a sum at the end of the episode according to the default settings of the Sokoban environment:

Complete the game	10
Move player	-0.1
Move box onto goal	1
Move box off of goal	-1

Input to the game was defined as

$$\underset{out \in \{0,\ldots,O-1\}}{\text{argmax}} \quad x_{1,out}$$

with $O = 4 = \#\{up, down, left, right\}$ and x_1 denoting the feed-forward output vector of the network.

Because the paper makes no mention of what the average reward should be for an empty list of past rewards (when accessing the past reward for the first time), I chose to use the first reward, i.e. the current reward, as the "average past reward", which has less bias than using a default value like 0. This means that the update at the end of the first episode does not change the weight parameters. The average past reward $\overline{r}(\tau)$ was otherwise computed as the mean of the past $min(len(past\_rewards\_earned), max(200, int(0.2 \cdot len(past\_rewards\_earned))))$ games. 200 was chosen as a sensible minimum window by me.

Learning rates and **decay** were set to 0.5 and $1.$ (no decay), respectively. The former choice was used by the paper's authors in their experiment, and the latter was used because Sokoban episodes are very short and decay seemed unnecessary.

3.2 Performance of the single-layer architecture

When applying the algorithm to Sokoban, weight parameters were initialized with μ_{ij} uniformly randomly distributed in $[-1, 1]$ and σ_{ij} uniformly distributed in $[0.05, 1]$. Note that the paper did not suggest any initialization principles for general use. The paper presented a simple robotic experiment in which the robot was expected to move forwards, so parameters were initialized with certain positive values because the researchers knew these were better suited to the task. Because the no beneficial initializations were known for Sokoban, a conservative zero-centered initialization was chosen.

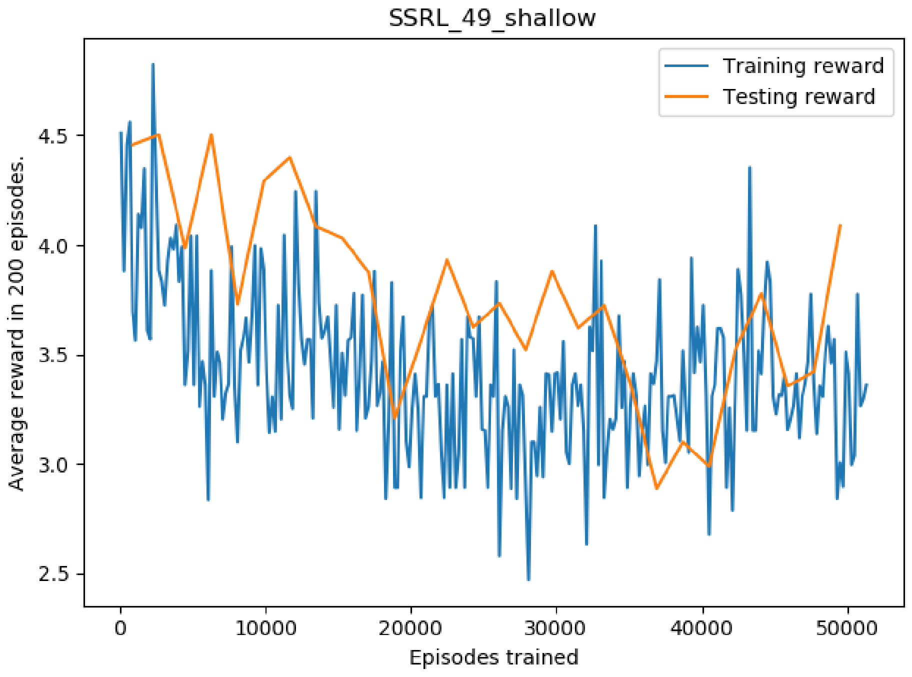


Figure 17: Training and testing reward of the shallow architecture

The algorithm, with the architecture presented in the paper and using the initialization above, failed to adequately solve the Sokoban task even for the simplest case $step\_distance = 1$. Figure 17 shows stagnant training and testing reward over $5e4$ episodes using the architecture described in the paper.

In the trivial case $step\_distance = 1$, the game can always be solved by simply moving against the box adjacent to the player (thereby pushing the box onto its goal), but a single-layer network is not able to learn this nonlinear mapping. Much more interesting, then, is the question of how deeper networks trained using this algorithm would perform.

3.3 Generalization to multiple layers

Preliminarily, it seems clear that a single-layer architecture is insufficient for combinatorial puzzles like Sokoban, as these games require the ability to make abstract inferences over combinations of multiple features (board tiles) in a way that is impossible with just a single layer. Because the paper only discusses a single-layer approach and makes no mention of generalization to multiple layers, expanding the architecture was a somewhat experimental task, even with the help of the authors.

Before reviewing the multi-layer update strategy, it is worth considering the intended function of this algorithm. The motivation behind formulae 4 and 5 is to account for *the influence a weight's deviation from its parameterization had on the output of the network*. This interpretation was confirmed to me by Hougen in an email exchange (figures 29 and 30).

As suggested in the first email (figure 29), this becomes more complicated with potentially negative inputs, and Hougen suggested that in such a case, one would replace $x_i(k)$ in formulae 4 and 5 with $|x_i(k)|$ in order to maintain this effect.

To begin with, a new update formula was needed which could be used in the multi-layer case. Subsequent emails with Hougen outlined the approach that was to be used (figures 31 and 32): Eligibility traces for distributions parameters μ_{ij} and σ_{ij} would be proportional to the gradients of the output nodes w.r.t the sampled weight at time step k. This requires backpropagation through the network for hidden layers.

The new eligibility trace formula for weights in layer b is described in equations 9 and 10, where x_b is the vector of network layer activations (with input layer $b = 0$ and output layer $b = H - 1$), $b \in \{0, ..., H - 1\}$ and $x_{H-1} \in \mathbb{R}^O$ (vectors indexed starting at 0). The network would use the same component-wise activation function $tanh(\cdot)$ as in the single-layer case. Note that w_b is only defined for $b \in \{0, ..., H - 2\}$.

$$e_{\mu_{b,ij}}(k) = |x_{b-1,i}(k)| \cdot (w_{b,ij}(k) - \mu_{b,ij}(k)) \cdot \sum_{out=0}^{O-1} \frac{\partial x_{H-1,out}}{\partial w_{b,ij}(k)} \tag{9}$$

$$e_{\sigma_{b,ij}}(k) = |x_{b-1,i}(k)| \cdot (|w_{b,ij}(k) - \mu_{b,ij}(k)| - \sigma_{b,ij}(k)) \cdot \sum_{out=0}^{O-1} \frac{\partial x_{H-1,out}}{\partial w_{b,ij}(k)} \tag{10}$$

A note on the validity of these formulae: Because of the difficulty in communicating detailed mathematical concepts per email, I do not claim that these update rules accurately reflect Prof. Hougen's instructions regarding backpropagation. Unfortunately, he was not able to respond to a final email (not added in this report) seeking confirmation that these formulae correspond to what he said in his email. The reader is encouraged to read Prof. Hougen's description (figure 32) for reference. Section 4.1 discusses problems with the update formula in more detail, and I argue that even minor tweaks to the last summed-gradient terms in equations 9 and 10, the principal potential source of confusion, won't solve fundamental problems with the algorithm's update formulae that result from the way it handles rewards.

With this new update formula, the interpretation is again that differences of sampled weights from their mean $(w_{b,ij}(k) - \mu_{b,ij}(k))$ and $(|w_{b,ij}(k) - \mu_{b,ij}(k)| - \sigma_{b,ij}(k))$ are "experiments" of the network to discover beneficial parameter update directions (either greater or lesser) and magnitudes, and the episode performance $(r(\tau) - \overline{r}(\tau))$ is feedback on the "result" of these experiments.

The sum of the gradients is a new measure for how much influence this "experimental" weight change had on the output of the network. Updates to the weight parameters should be in proportion to how much of an effect changes to the weights have on the output, and whether that change is correlated with a positive or negative change in reward collected. The goal of the algorithm is to discover beneficial changes in the weight parameters through experiments (stochastic sampling) and observation of their outcomes.

3.4 Performance of the multi-layer architecture

3.4.1 Sokoban

To test the function of the new update rules, the algorithm was initialized in the same fashion as in the single-layer case. Because initial testing would be restricted to $step\_distance = 1$, a relatively shallow architecture with $layersizes = (49, 100, 50, 10, 4)$ was chosen, because this was presumed to be enough to learn to move the agent in the direction of an adjacent box without slowing training down excessively through network depth.

The algorithm was not able to solve even the simplest training case for Sokoban. Figure 18 shows the training and testing error in the Sokoban environment with $step\_distance = 1$. Performance was equivalent to random play, even after 17,000 episodes. Training was terminated at this point due to the extreme demand on CPU and memory (see 4.2.1) and lack of training progress.

Some immediate explanations for this behavior come to mind:

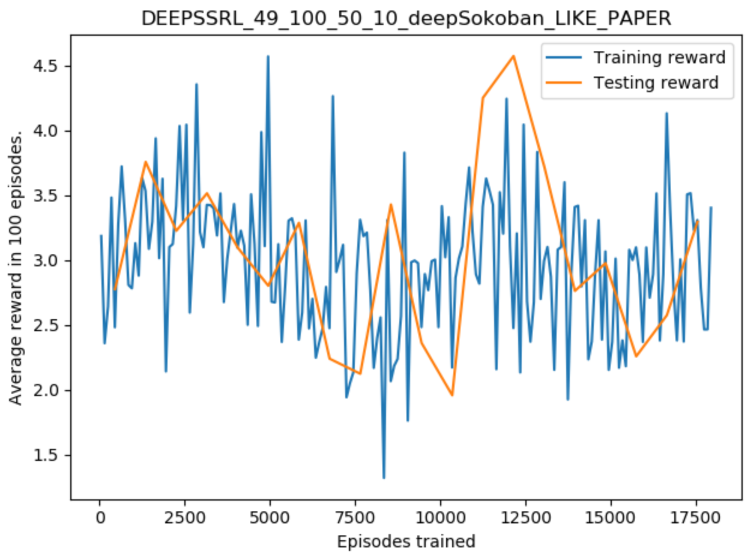


Figure 18: Training and testing reward of the deep architecture.

1) The algorithm is not effective in this kind of task - it is meant for dynamic, continuous-output environments in which the entire output layer is used as input to a system (e.g. controlling a robot arm with n degrees of freedom), whereas in Sokoban,

$$\underset{out \in \{0,\ldots,O-1\}}{\operatorname{argmax}} \quad x_{H-1,out}$$

is used as game input. This explanation was evaluated in section 3.4.2.

2) The algorithm was appropriately chosen, but training hyperparameters such as episodes in training, weight parameter initialization, or network depth were incorrectly chosen for the task.

Regarding 2): While it cannot be ruled out that another configuration of hyperparameters would have been more optimally suited to the task, training should still have improved at least marginally over tens of thousands of episodes instead of stagnating or even receding like in the evaluations.

3.4.2 Mountain Car

In order to evaluate whether it was the choice of game that caused the failure of the algorithm, a similar architecture with $layersizes = (2, 10, 10, 10, 1)$ was used to learn the simple trial game Mountain Car (implemented in the OpenAi Gym environment 'MountainCarContinuous-v0') with approximately continuous floating-point input and output.

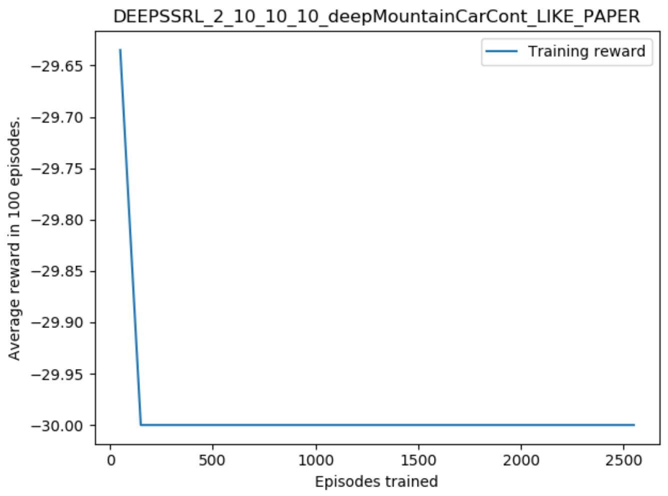


Figure 19: Training reward in Mountain Car.

Similarly disappointing performance (figure 19) suggests that the algorithm itself, either in its implementation or its design, is to blame. Training reward completely flatlined at the minimal value after a few hundred episodes, and because $(r(\tau) - \overline{r}(\tau)) = 0$ from that point on, the algorithm ceased to make changes to its parameters. Presumably, the parameters were stuck in a local minimum such that even stochastic weight sampling (3) at every time step was not sufficient to find an action yielding more than minimal reward. Hence, training was stuck without possibility for recovery.

Training was stopped early at less than 2000 episodes due to the obvious failure of the algorithm and because training even ≈ 1 episode per second used memory to the point where a standard Lenovo Thinkpad X250 laptop crashed, making overnight training impossible.

4 SSRL: Problems and Critical Discussion

4.1 Algorithmic

4.1.1 Parameter divergence

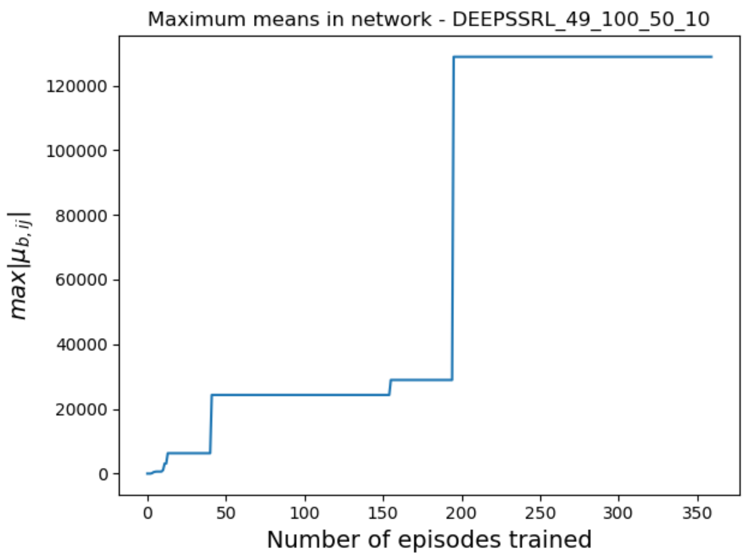


Figure 20: Means explode in a 4 layer network (Sokoban)

When training an agent using the algorithm, it could be observed (see figure 20) that, despite very conservative zero-centered initialization and clamping of standard deviations at the end of every episode, some means would begin to diverge to very large values, with this behavior getting more extreme as time went on.

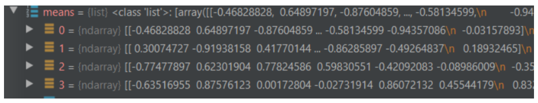
This would lead to the effect that an agent would always make the same decision (i.e. the network would always produce the same arg max) over long periods of time spent training, because a single weight or group of weights would have an extreme influence on the output of the network.

This problem appears to occur for games like Sokoban in which all but few rewards are near zero and some are large. Of particular importance is the factor $(r(\tau) - \overline{r}(\tau))$ in weight updates 6 and 8: The average reward for random play in Sokoban is somewhere near 3.5 for $step\_distance = 1$, and successful completion of the game in one step gives 10.9 points. This means that for a game solved in a single step, which is about a quarter of the time for $step\_distance = 1$ with initial uninformed play, all weight means are incremented by their deviation from their mean $(w_{b,ij} - \mu_{b,ij}) =: y_{b,ij}$ multiplied by a factor of $10.9 - 3.5 \approx 7.4$, not yet considering gradients.

This effect is drastically apparent in deep networks, which can hold thousands of weights and in which it is statistically likely that *some* weight is very far from its mean, which essentially guarantees that at least some weight in the network will receive an inappropriately large update.

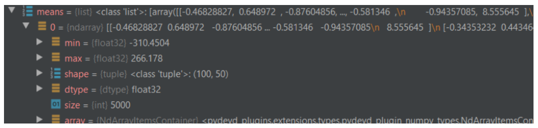
Consider the effect winning for the first time has on weight updates. Until then, the agent will have earned an average reward somewhere between -0.5 and 0.5. Applying the factor $(r(\tau) - \overline{r}(\tau))$ represents 10.9-fold multiplication of all eligibility traces and $y_{b,ij}$ values for the network. Making matters worse, the gradients for some weights may be large with initial random initialization, and are generally large when means in the network are large, further increasing the size of the parameter update. All factors compounded together mean that some means may grow orders of magnitude greater in a single step, which in turn leads back to the original problem of potentially causing the network to produce the same arg max every time and getting stuck in a rut in which the average reward corresponds to random play. About once every four games the algorithm wins, further increasing $\mu_{b,ij}$ magnitudes.

Consider a particularly bad case that was observed in the 4-layer architecture used in training (figure 21): Because the agent had won its first game and lost the second game, weight updates were immediately multiplied by a factor on the order of 10, resulting in an explosion in some weight means by two orders of magnitude in a single step.



```
▼ means = {list} <class 'list'>: [array([[-0.46828828,  0.64897197, -0.87604859, ..., -0.58134599,\n        -0.94
  ▶ 0 = {ndarray} [[-0.46828828  0.64897197 -0.87604859 ... -0.58134599 -0.94357086\n  -0.03157893]\n
  ▶ 1 = {ndarray} [[ 0.30074727 -0.91938158  0.41770144 ... -0.86285897 -0.49264837\n   0.18932465]\n
  ▶ 2 = {ndarray} [[-0.77477897  0.62301904  0.77824586  0.59830551 -0.42092083 -0.08986009\n  -0.35
  ▶ 3 = {ndarray} [[-0.63516955  0.87576123  0.00172804 -0.02731914  0.86072132  0.45544179\n   0.83
```

(a) Randomly initialized means in the first episode ≈ 0.



```
▼ means = {list} <class 'list'>: [array([[-0.46828827,  0.648972  , -0.87604856, ..., -0.581346  ,\n        -0.94357085,  8.555645  ],\
  ▼ 0 = {ndarray} [[-0.46828827  0.648972   -0.87604856 ... -0.581346   -0.94357085\n   8.555645  ]\n [-0.34353232  0.44346
    ▶ min = {float32} -310.4504
    ▶ max = {float32} 266.178
    ▶ shape = {tuple} <class 'tuple'>: (100, 50)
    ▶ dtype = {dtype} float32
      size = {int} 5000
    ▶ array = {NdArrayItemsContainer} <pydevd_plugins.extensions.types.pydevd_plugin_numpy_types.NdArrayItemsCon
```

(b) At least two values on the order of $1e2$ at the end of the second episode

Figure 21: The results of a second episode update

It could also be observed that most $\sigma_{b,ij}$ values tended to approach 1.0 or values close to it after thousands of episodes spent training, despite the fact that

$$\forall \sigma_{ij} \in [0.05, 1], \forall \mu_{ij} \in \mathbb{R} : \mathop{\mathbb{E}}_{w_{ij} \sim \Psi(\mu_{ij}, \sigma_{ij})} [|w_{ij}(k) - \mu_{ij}(k)| - \sigma_{ij}(k)] < 0$$

This would be expected to lead to asymptotically convergent behavior of the algorithm as long as reward tends to improve over time, because it would mean that eligibility traces $e_{\sigma,b,ij}$ would tend to be negative, therefore leading to smaller variance in parameter sampling. However, because the algorithm did not improve, it had the effect of increasing variance. With increasing variance, $y_{b,ij}$ values increase in value, meaning that rare and random wins lead to even bigger explosions in $\mu_{b,ij}$ updates, further worsening the problem of exploding means in a vicious cycle of ineffective parameter updates.

4.1.2 Efficacy of update formulae

It is my belief that one of the weakest links in the algorithm is in the update formulae for $\mu_{b,ij}$ and $\sigma_{b,ij}$ (both single and multi-layer versions). This is because credit assignment occurs in a very broad way: The algorithm updates all weight parameters according to whether their deviations from normal behavior were observed to correlate with increased reward. This sounds appropriate in theory, but in practice the correlations between deviations $y_{b,ij}$ and $z_{b,ij} := |w_{ij}(k) - \mu_{ij}(k)| - \sigma_{ij}(k)$ and the weight configurations which resulted in high or low performance in an episode are too low for weight updates to be meaningful on non-asymptotic time scales. Credit assignment is not sufficiently granular; as long as *any* part of the network influences $(r(\tau) - \overline{r}(\tau))$, *all* weight parameters are updated accordingly.

For a game like Sokoban, this effect is compounded by the fact that it is the *highest output value* which determines agent action choice. As long as $\operatorname{argmax} x_{H-1}$ remains unchanged, any number of weight combinations will lead to the same agent behavior. This means that whenever the network produces the correct output decision, even those weights which "voted for" a different action are positively updated, and in fact, because weight updates are proportional to the gradient, weights which voted very strongly **against** the correct output are also updated very strongly in the direction of their vote.

The network learns only "more of this", whatever "this" was, without regard for whether more is always better or to what extent "this" even resulted in a highly rewarded action. Unfortunately, the algorithm cannot know *which part* of its output led to increased or decreased reward, so it has to update its parameters as if *all* changes were responsible for the changed feedback.

One could argue at this point that because Sokoban is a game with discrete output, where only the argmax is important, these considerations would apply differently in a more continuous game in which this algorithm is more "at home". Firstly, the algorithm was not successful in the "Mountain Car" trial game of this nature when evaluated.

Secondly, imagine a hypothetical game in which an agent controls the virtual hand movement of a human player controlling a joystick at an arcade booth and receives the pixel output of the arcade booth as part of the game state at every time step. The virtual human is playing the game Sokoban in the arcade booth, and the agent's output into the game is a vector $x \in \mathbb{R}^4$. The virtual player moves the joystick in the direction of $\widetilde{y} := \frac{y}{\|y\|}$,

$$y := \begin{pmatrix} 1 \\ 1 \end{pmatrix} + \begin{pmatrix} tanh(x_0) \textbf{ if } x_0 \geq x_1 \textbf{ else } -2 - tanh(x_1) \\ tanh(x_2) \textbf{ if } x_2 \geq x_3 \textbf{ else } -2 - tanh(x_3) \end{pmatrix}$$

with $\widetilde{y} = \begin{pmatrix} 1 \\ 0 \end{pmatrix}$ corresponding to a movement of the joystick exactly to the right and $\widetilde{y} = \begin{pmatrix} 0 \\ 1 \end{pmatrix}$ corresponding to an upward movement (analogously for left and down). The virtual arcade booth then translates this joystick movement into an internal command converting the movement direction of the joystick into the closest move direction $d \in \{up, down, left, right\}$ within Sokoban.

This would be a continuous-input game with hidden internal discretization of the input corresponding exactly to the setup from the original (discrete) Sokoban game. Simply moving the discontinuous mapping $x \mapsto \operatorname{argmax} x$ from the agent design to the structure of the game itself turns Sokoban into a "continuous-input" task. Therefore, a black-and-white distinction between discrete and continuous games is in general not warranted (one could easily imagine a game that mixes discrete and continuous elements - for instance if the same hand which controls the joystick in one part of the game would then have to learn to carry a cup through the arcade without spilling its contents). A general-purpose reinforcement learning algorithm must therefore be able to master both types of output.

4.1.3 Potential Improvements

If one wanted to adapt this algorithm to be more effective, the first area of focus should be on fixing the exploding means problem. A potential solution might involve finding a way to standardize reward structure or preprocessing received rewards in order to minimize reward variance and circumvent the exploding means problem.

Two problems come to mind here:

1) Not all tasks have a known reward structure which invites reward preprocessing.

2) High reward variance is not itself a bad thing - in many games, rewards have a linear structure in which orders of magnitude of difference are possible and meaningful. The 100th unit of currency won on the stock market is worth as much as the first, and $10n$ units are always worth 10 times as much as n units.

Clearly, it is not sufficient to do away with large reward variance alltogether. The problem might therefore be rephrased in terms of orders of magnitude: How can parameters be learned so that different orders of magnitude in reward structure do not lead to orders of magnitude of difference in weight parameter update behavior? The solution to this problem should also consider the sizes of gradients in the network. Gradient sizes are the other source of runaway parameter updates, but backpropagation is too powerful an idea to give up entirely.

Perhaps the solution lies not in changing how the agent treats rewards, but in how the agent sets its weights. If there were some way to control $\mu_{b,ij}$ values, for instance by applying an additional step similar to the clamping step 8 for $\sigma_{b,ij}$ and forcing all weight means into the range $(-1, 1)$, it would be possible to maintain orders of magnitude of difference in weight sizes (making them arbitrarily small), without allowing weights to become arbitrarily large.

Next, consider the logic behind using the backpropagated gradient for determining eligibility: It serves as an approximation for the "degree of influence" the weight change had on the output of the network.

Gradients are only *locally* good approximations to the the neural network. Specifically,

$$\|f(x+a) - [f(x) + \mathcal{D}_f(x) \cdot a]\| \in \mathcal{O}(a^2)$$

, where $\mathcal{D}_f$ denotes the total derivative of $f \in \mathcal{C}^2$. In areas with strong curvature (large second total derivative), sampled weights may be so far from their mean that they exceed the range of usefulness of the derivative as an approximation to their effect on the behavior of the network, i.e. the approximation error grows too large to be useful, of quadratic order.

Finding a good way to normalize weights might bring this gradient problem under control, too: If weights can be made to only minimally vary from their mean, then gradients can be be used as a more accurate approximation of the effect of deviations $y_{b,ij}$ and $z_{b,ij}$ on the output of the network, and eligibility traces would in turn be expected to be become more informative.

The next problem to address would be that of credit assignment: Can one find a better way to discover which weights are responsible for better/worse changes $r(\tau) - \overline{r}(\tau)$ in network performance?

At the cost of training speed, one could experiment with "shutting off" weight sampling for some weights, i.e. $w_{b,ij}(k) \leftarrow \mu_{b,ij} \; \forall (b, ij) \in S \subset N$ for some S, (N being all network weights). This would allow for a more fine-grained approach to parameter search. In the extreme case, just a single parameter would be sampled and updated over an episode.

In this vein, perhaps **pruning** techniques could be employed in finding useful $S \subset N$ which have the greatest effect on the output of the network after training. Pruning techniques add to a standard training cycle an additional step: Weights with the smallest effect on the feed-forward output of the network are set to 0. There are different criteria for defining "smallest effect", e.g. by gradient magnitude or by absolute weight size. The advantage of pruning in this case is that as the size of the network decreases, it is possible to examine fewer weights at a time without slowing training down to extreme levels (because there would otherwise be too many weights to examine each in detail), thereby avoiding the bigger issue of having to update very many weights for the contributions of a few.

This approach is inspired by the *Lottery Winner Hypothesis* [12], which states that deep, randomly initialized networks contain a subnetwork which is well-initialized for training, and that this subnetwork can be discovered by iteratively training and pruning. After training a network for one cycle, the network is pruned and then reinitialized to the original (random) weights for all non-zero weights. After some iterations, this cycle yields a much smaller network which (so the hypothesis states) can reach the same level of performance after training as the original network.

Assuming one has found a way to update network parameters so that the update rules produce convergent parameter sizes (i.e. one has solved the exploding means problem), one could wrap training in an algorithm meant to improve sparsity of weight matrices:

```
**Data:** Non-sparse network $M \subset N$ with predefined and constant random initializations $X_{M \subset N}$
**Result:** Non-zero weight indices of sparse, trainable network $\overline{M} \subset M$
$subnetworks \leftarrow [M]$
**for** *i = 1, ..., T* **do**
    $net \leftarrow subnetworks[-1]$
    train $net$ with $SSRL$ and random initializations $X_{net}$
    $pruned\_net \leftarrow net.prune().get\_non\_zero\_weights()$
    $subnetworks.append(pruned\_net)$
**end**
**return** $\overline{M} \leftarrow subnetworks[-1]$
```

4.2 Technical

4.2.1 Training and game environments

Much of the effort that went into this project was in manipulating the Sokoban game environment for training. The game environment [2] provided by OpenAi's "Gym" library was of poor code quality and didn't provide a simple way to initialize the game at a certain move distance from a solved state, as would usually be required in a learning task for this game. Coauthor Perle wrote a custom method to randomly play games in reverse to move from a solved state to an unsolved state and which returned the sequence of states and moves that yielded the final state using adapted game code. This output was used as the basis for training at a given $step\_distance$.

However, the game environment didn't allow for simple initialization "to a given state", meaning that custom functions were needed to load and set the game state within the environment, as well as changing all internal variables of the game environment to ensure correct rewarding etc. This was not a trivial task, because the internal variables of the Sokoban environment were used in many places in complicated ways, and it wasn't initially clear how many such variables even existed. Classes and functions within the environment were only minimally documented, if at all.

These efforts were hamstrung at every step by minor bugs and inconsistencies in the game code; for instance, games generated by the above function had the error that fields containing a "box not on goal" were marked with the integer "3" and fields containing a "box on goal" were marked as "4", but the game environment itself represented game states with these two values switched! It was up to the user to discover this bug and then manually change the game state vectors (and other field values that relied on it) with the correct values.

Another annoying inconsistency was that the OpenAi specification for discrete game input asks that actions begin indexed at 0. However, the actions "push": up, down, left, right were numbered between 1 and 4, and the actions "move": up, down, left, right were numbered between 5 and 8. Action 0 was "wait". This makes no sense, because "wait" is never an appropriate action in Sokoban (unlike in a shooter game, for example), meaning it should not have been added or should at least have been moved to the last action, and it was up to the user to figure out that "move" is a completely superfluous input, because the action "push" internally calls the function "move" whenever there is no box adjacent to the agent in the move direction. This was annoying because other Gym games have discrete output beginning at 0, so if the user wants to apply the same algorithm to a different Gym game, they have to implement an if-else check to determine whether to add 1 to the game input or not.

Additionally, all Gym environments were very slow and used extreme amounts of memory, meaning that it wasn't realistic to train agents in parallel on a standard laptop, not to mention generate games at the same time. Sokoban environments specifically tended to spontaneously crash due to a memory error, meaning the code needed to be failproofed in multiple locations to reload the environments in case of a memory exception. Mountain Car in particular could only be trained at the rate of about one episode per second, which is painfully slow when considering the game's simplicity.

All of this was compounded by the fact that much development work had to go into loading and accessing stored game states and randomly using them during training. In a perfect world, the Gym library could be used to write a game loop that is 20 lines long, in which $environment.reset()$ accepts a parameter to specify to what step distance the game environment should be initialized. Instead, what was required was a relatively large engineering effort to coordinate data access and storage in order to not have to train the algorithm starting at 20+ moves from the destination.

Training and generating game states could be done in parallel with some engineering effort, but was still frustratingly inefficient because games were generated many more steps away from the solution than was needed (28 steps) both once during data generation *as well as once during environment initialization from the environment's constructor*, meaning that the computational overhead just to play a single game was extremely large and involved multiple classes generating, saving, loading, and manually initializing the game environment for use.

4.2.2 Numerical libraries

Initially, implementation of SSRL was designed to rely on numpy for all data manipulation, Gaussian sampling, number generation, etc. This was entirely sufficient for the single-layer algorithm and was supported by the game environments, which internally saved game states and related vectors as numpy arrays.

Numpy also provided convenient methods for managing the parameters of the network. The initial approach was to interpret the entire set of neural network weights as a (reshaped and partitioned) vector sampled from a diagonal-covariance normal distribution, i.e.

$$\boldsymbol{w_b} \sim \Psi(\boldsymbol{\mu_b}, diag(\boldsymbol{\sigma_b}))$$

This approach was extremely slow, and it turned out to be much faster to use numpy's $apply\_along\_axis$ function and generating each weight as the output of a one-dimensional normal distribution.

When implementing the deep architecture, however, backpropagation through the network was required, so pytorch libraries were used to compute the gradients. This was done with the help of the nn.Module class, which is the pytorch class for simple construction of neural networks. All the mechanics for storing and sampling from the parameterized distributions of the network were already implemented in numpy, however, so functions were written to translate between tensors and arrays to take advantage of the existing classes for handling weight sampling. Despite the superfluous array copying this involved, numerical libraries only took up a small fraction of the runtime of the application, far outweighed by the slow game environments the agent interacted with.

Some difficulty was involved in accessing the internal tensors of the nn.Module child class which held the agent's ANN; linear layers stored the bias weights and the conventional weights in a separate $n \times 1$ vector and $n \times m$ matrix, respectively, whereas the design choice until then had been to store all weights in one $n \times (m + 1)$ matrix, where bias nodes would be appended as a $1.$ value to the end of a layer's activation. When accessing the network's parameters for backpropagation, the method $net.parameters()$ iterated over the weight matrix and the bias vector of a weight layer separately, meaning that there was no longer a one-to-one correspondence in the data structure used by the agent to hold the weight parameters and the internals of the nn.Module class, and there was an overhead in translating and accessing the right data fields when applying the update rules of the algorithm.

Finally, my (Weichselbaumer) laptop did not support performing backpropagation calculations on the GPU using CUDA, meaning that training in the already slow Gym libraries was slowed down further by backpropagation calculations. Minor tweaks or changes to the algorithm would require a new model to be trained for many hours.

5 Summary and Conclusion

In summary, by far the best performance was achieved with DeepCubeA, a designated combinatorial/planning algorithm. DeepCubeA was able to accurately solve Sokoban board states when training with only a standard laptop.

DeepCubeA with Environment Model produced inferior results when compared with the standard DeepCubeA algorithm.

Finally, Imagination augmented agents and the general-purpose learning algorithm Stochastic Synapse Reinforcement Learning was not able to learn Sokoban on evaluation. More research is necessary in order for Stochastic Synapse Reinforcement Learning to be robust and effective in game environments with highly variable reward structure like Sokoban.

The authors would like to thank professor Köthe and the coordinators and assistants of the lecture Advanced Machine Learning for the fun project and semester.

Appendices

A Test dataset for different room types.

A.1 Sokoban-small-v0

A.1.1 Unsolved

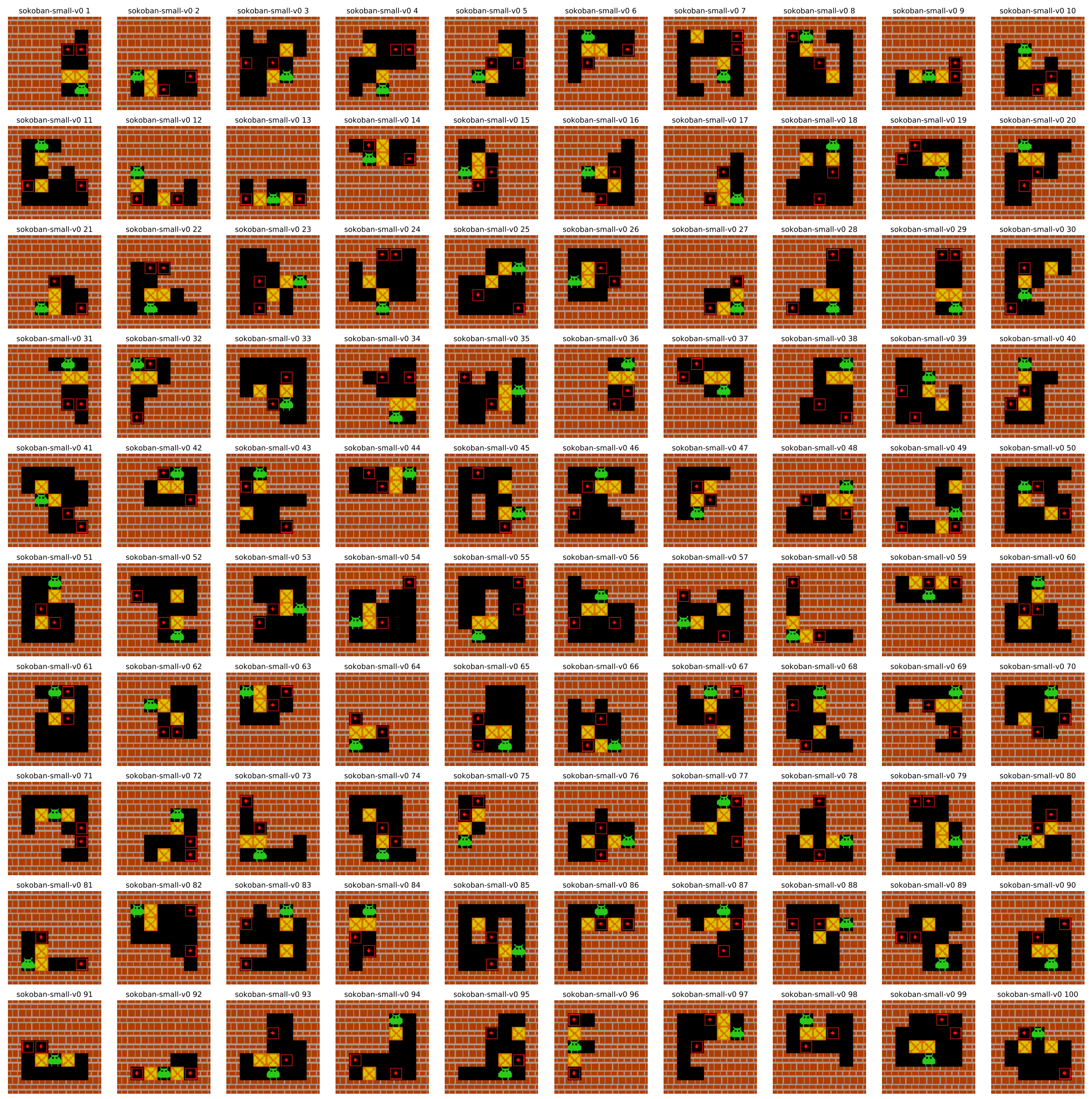


Figure 22: loss of the "10 $\times$ 10 grid with two boxes" model with a batch size of 1

A.1.2 Solutions

Table 9: Solutions for Sokoban-small-v0.

game name	DeepCubeA (7 × 7 with 2 boxes model)
sokoban-small-v0 1	↑↑↓↓←↑↑
sokoban-small-v0 2	↓→←↑→→→
sokoban-small-v0 3	←←↓←↑→→↑↑↑→→↓←↑←↓
sokoban-small-v0 4	↑↑→↑↑←←↓→→←↓↓←←↑→↓→↑←↑→
sokoban-small-v0 5	→→↑→↑↑←↓→↓←↓↓→↑
sokoban-small-v0 6	↓→→
sokoban-small-v0 7	↑→↑↑←←↓→←←←↑→→→
sokoban-small-v0 8	↓←↓→↓→↓→→↑←←←↓←↑↑↑
sokoban-small-v0 9	→↓→↑←←↓←←↑→→→
sokoban-small-v0 10	←↓↓→→→→↓←↑←←←↑↑→↓←↓→→
sokoban-small-v0 11	↓←↓↓→→→←←←↑↑→↓←↓↓→→↑←
sokoban-small-v0 12	↓→↓→
sokoban-small-v0 13	←→→
sokoban-small-v0 14	→→↑←
sokoban-small-v0 15	→←↑↑→↓↓
sokoban-small-v0 16	↓→↓→→↑↑←↓→↓←↑←←↑→
sokoban-small-v0 17	←↑
sokoban-small-v0 18	↓←↑←↓↓↓←↓→
sokoban-small-v0 19	↑←←→→→↑←←
sokoban-small-v0 20	→↓↑←↓↓
sokoban-small-v0 21	→↑↓→
sokoban-small-v0 22	↑↑←↑→↓↓↓→→↑←↓←↑↑
sokoban-small-v0 23	←←→↓←↑↑←←↓→↓→→↑←
sokoban-small-v0 24	↑↑↓←←↑→→↑↓↓→↑↑
sokoban-small-v0 25	↑←↓←↓↑→↓↓←↓→
sokoban-small-v0 26	→→←←↑→
sokoban-small-v0 27	←→↑
sokoban-small-v0 28	→↑↑←↓→↓←←←↑→↓→↑↑↑
sokoban-small-v0 29	↑↑↑↓↓↓←↑↑↑
sokoban-small-v0 30	↑→↑←→↑→→↓←←↑←←↓↓↓
sokoban-small-v0 31	↓↓↑↑→↓↓
sokoban-small-v0 32	↓↓↓↑→↑
sokoban-small-v0 33	↑←↑←←↓→↑→↓
sokoban-small-v0 34	↑↑→↑←↓↓↓→↑↑
sokoban-small-v0 35	←→↓←↑←←↓←↑
sokoban-small-v0 36	↓↑←↓↓
sokoban-small-v0 37	↑←→→↑←←↓←
sokoban-small-v0 38	↓↓↓↑↑←↑←↓↓
sokoban-small-v0 39	←↓↓↓→→→↑←←←↓←↑→↑↑→↓↓→↓←
sokoban-small-v0 40	↓→↓↓←
sokoban-small-v0 41	→↑→↓↓←↓→↑←↑↑↑←←↓→→↑→↓↓
sokoban-small-v0 42	←↓←↓→→↑→↑←
sokoban-small-v0 43	↓↓→↓↓←←↑↑→↓←↓→→
sokoban-small-v0 44	↓←→↑←←
sokoban-small-v0 45	←↓←↑↓←←↑↑↑→↑→→↓↓↓←↑↑→↑←
sokoban-small-v0 46	↓↑→→↓←←↓←↓↓→→↑←←
sokoban-small-v0 47	↑←↑↑→→↓
sokoban-small-v0 48	↓←←
sokoban-small-v0 49	↓←←←→→→↑↑←↑↑→↓↓↓
sokoban-small-v0 50	←↓↓→→→←←↑←↑→
sokoban-small-v0 51	↓↓←←↓↓→↑
sokoban-small-v0 52	↑←↑↑↑→→↓←→↓←↑←←→↓
sokoban-small-v0 53	←→↑↑←↓↓→↓←←
sokoban-small-v0 54	→←↑→→→↓→↑↑

sokoban-small-v0 55	↑→→↓→↑←←←↑↑←↑→→→
sokoban-small-v0 56	↓←↑←↓→↓→
sokoban-small-v0 57	→→↓→→↑↑←↓↑←←↓←↑
sokoban-small-v0 58	→←↑↑↑
sokoban-small-v0 59	↑→←↓←←↑→
sokoban-small-v0 60	↓←↓←↓↓→↑
sokoban-small-v0 61	↓↓→↑↓↓↓←↑←↑→
sokoban-small-v0 62	→↓↓→→↑←→↑↑←↓←↓↑→↓
sokoban-small-v0 63	↓→←↑→→
sokoban-small-v0 64	↑→
sokoban-small-v0 65	↑→↑←←↓→↓←
sokoban-small-v0 66	↑↑←←←↓→↑→→↓↓←↑
sokoban-small-v0 67	↓→↓→↓↓←↑↑←↑→↓→↑←←←←↓→
sokoban-small-v0 68	←↓↓↓↓→→↑←↓←↑↑←↑↑→→↓←→↓↓→↓←
sokoban-small-v0 69	↓↓↑←
sokoban-small-v0 70	↓←←←↓→↑→↓→
sokoban-small-v0 71	→↑→↓↓←↑←↑←←↓→→→↑→↓
sokoban-small-v0 72	→↓↓←←←↓→→↑→↑↑←↓→↓←↓←←↑→→
sokoban-small-v0 73	↑↓←↑↑↑
sokoban-small-v0 74	↑←↑←↑→→↑→↓↓
sokoban-small-v0 75	↑→↑
sokoban-small-v0 76	←↑←↓↑←←↓↓→↑←↑→
sokoban-small-v0 77	←↓→←↓→↓↓←←↑→→↑→↑
sokoban-small-v0 78	←↓←↑↑↑↓←↓←↓→
sokoban-small-v0 79	←↓←↑↑↑→↑←↓↓↓→→↑←↓←↑↑
sokoban-small-v0 80	↓→↑→↑
sokoban-small-v0 81	→↑↓→→
sokoban-small-v0 82	→→→←←↓←↓→→→↑→↓
sokoban-small-v0 83	↓→↓←↓→↓←←←→↑↑→↑←←
sokoban-small-v0 84	↓↓↑↑←↓
sokoban-small-v0 85	←→↑↑↑←←↓↓↓←←↑↑→↑→↓
sokoban-small-v0 86	↓→←↑←←↓→
sokoban-small-v0 87	↓↑←←↓→→↓
sokoban-small-v0 88	←←←↓↓→↑
sokoban-small-v0 89	→↑←↓←↑→↑←←→→↑←↑←↓
sokoban-small-v0 90	←↑→↓→↑↑
sokoban-small-v0 91	↓←↑→↓→→↑←←←↓←↑
sokoban-small-v0 92	←→→
sokoban-small-v0 93	↑↑↓↓←←↑→→
sokoban-small-v0 94	↓→↓↓←←←←↓→→↑→→↑↑←↓→↓←←←
sokoban-small-v0 95	→↑←↓←↑↑→↑↑→↓↓
sokoban-small-v0 96	↑↓↓
sokoban-small-v0 97	←←↑←↓→→→↑←
sokoban-small-v0 98	↓→
sokoban-small-v0 99	←↑←↑→→→←←↓→↓→↑↑
sokoban-small-v0 100	←↓←↓→↑→↓←↓→→↑→↑←←↓←↑

A.2 Sokoban-small-v1

A.2.1 Unsolved

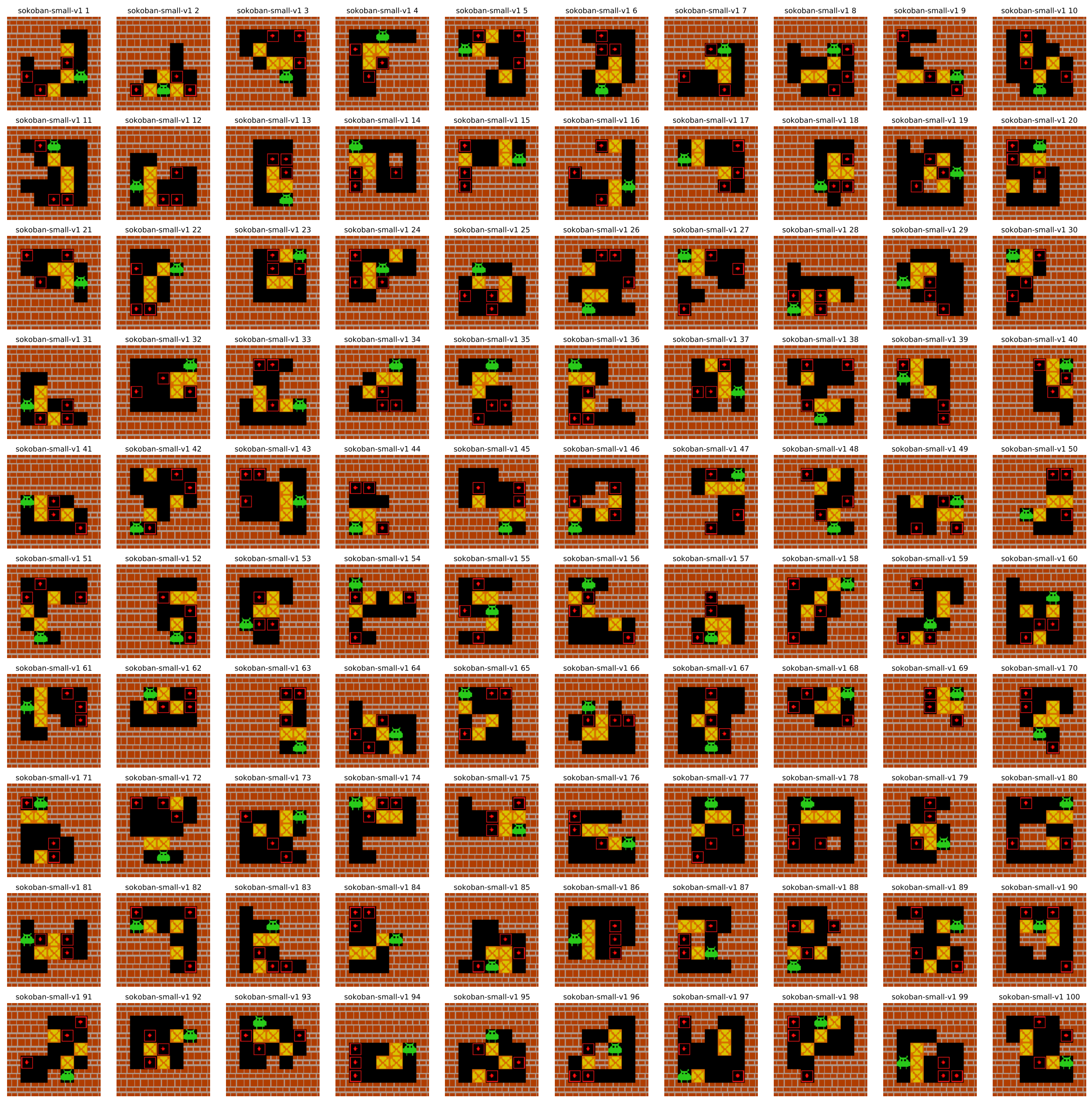


Figure 23: loss of the "10 $\times$ 10 grid with two boxes" model with a batch size of 1

A.2.2 Solutions

Table 10: Solutions for Sokoban-small-v1.

game name	DeepCubeA (7 × 7 with 2 boxes model)	DeepCubeA (7 × 7 with 3 boxes model)
sokoban-small-v1 1	←↓←→↑←←→→↑→↑↑←↓	←←←→→↓←→→↑↑↑↑←↓
sokoban-small-v1 2	→←←↑→	←↑→↓→
sokoban-small-v1 3	↑↑←↑→←←←↓→↓→↑↓→	↑↑←↑→←←←↓→↓→↑↓→
sokoban-small-v1 4	↓←↓	↓←↓
sokoban-small-v1 5	→→→↑←→↓↓→↑↓↓↓←←↑→↓→↑	→→→↑←↓→↓→↑↓↓↓←←↑→↓→↑
sokoban-small-v1 6	↑↑↑→→↓↓←↓←↑↑↓→↑	↑↑↑→→↓↓←↑↓↓←↑↑
sokoban-small-v1 7	←↓↑→→↓←↓←↑↓←	←↓↑→→↓←↓←↑↓←
sokoban-small-v1 8	↓←↓→↓→↑↑←←←←↓→→←↓→↑→	↓←↓→↓→↑↑←←←←↓→→→←←↓→
sokoban-small-v1 9	←↓←←←↑↑↑↓→↓←↓→→→	←↓←←←↑↑↑↓→↓←↓→→→
sokoban-small-v1 10	↑←↑↑→→↓←↓→	↑←↑↑→→↓←↓→
sokoban-small-v1 11	↓↓↓↑↑→→↓←↓←↑↑→↑←	↓↓↓↑↑→→↓←↓←↑↑→↑←
sokoban-small-v1 12	→→↓→↑←←←↓→→←←↑↑↑→↓↓←↓→	→→↓→↑←←←↓→→←←↑↑↑→↓↓←↓→
sokoban-small-v1 13	↑↑↓↓←←↑→↓→↑←↑	↑↑↓↓←←↑→↓→↑←↑
sokoban-small-v1 14	↓↓→→↑←↑←↓	↓↓→→↑←↑←↓
sokoban-small-v1 15	←←↑←←↓↓↑↑→→↓←↑←↓→→→→↑←←←	←←↑←←↓↓↑↑→→↓←↑←↓→→→→↑←←←
sokoban-small-v1 16	↓←→↑←←←↓←↑→→→→↑↑↑←	↓←→↑←←←↓←↑→→→→↑↑↑←
sokoban-small-v1 17	→→→↓↑↑→↓←←←←↑→→→	→→→↑→↓←↓↑←←←↑→→→
sokoban-small-v1 18	↑↑↑→→↓↓↑↑←←↓→↓	↑↑↑→→↓↓↑↑←←↓→↓
sokoban-small-v1 19	←←↑←↓→→→↓↓←↑→↑↑←	←←↑←↓→→→↑←→↓↓↓←↑
sokoban-small-v1 20	↓←↓←↑↓→↓↓←↑↑	↓←↓←↑↓→↓↓←↑↑
sokoban-small-v1 21	←←↑↓→→↑←↑←←↓→↓→↑	←←↑↓→→↑←↑←←↓→↓→↑
sokoban-small-v1 22	←↓↓←→↑↑←↓↓↑←↓	←↓↓←→↑↑←↓↓↑←↓
sokoban-small-v1 23	←↓←←↓↓→↑→↓→↑	←↓←←↓↓→↑→↓→↑
sokoban-small-v1 24	↓←→↑↑←←↓→↑→↓→→↑←←←	↓←→↑↑←←↓→↑→↓→→↑←←←
sokoban-small-v1 25	↓←↓↓→→↑←↓→→→↑←↓←←↑↑↑→→↓↓→↓←	↓←↓↓→→↑←→↓→→↑←↓←←↑↑↑→→↓↓→↓←
sokoban-small-v1 26	↑←↑→→→←←↓→↓→↑↑↑←↑←↓↓←↓→↓→↑↑↑	↑←↑→→→←←↓→↓→↑↑↑←↑←↓↓←↓→↓→↑↑↑
sokoban-small-v1 27	→←↓↓↓↑↑→→→	→←↓↓↓↑↑→→→
sokoban-small-v1 28	→←↑↑→→↓←→↑→→↓←	→←↑↑→→↓←↑→→→↓←
sokoban-small-v1 29	→→↓→→↑↑←↑←↓↓←←↑→↑→↓→↓↓	→→↓→→↑↑←↑←↓↓←←↑→↑→↓→↓↓
sokoban-small-v1 30	→↓↑←↓↓	→↓↑←↓↓
sokoban-small-v1 31	→↓→←↑←↑↑→↓↓→	→↓→←↑←↑↑→↓↓→
sokoban-small-v1 32	←←↓↓↓→→↑←←←↑↑→→→↓←	←←↓↓↓→→↑←←←→↑↑→→↓←
sokoban-small-v1 33	←↓←↑↑↑↓↓↓←←↑→↑↑	←↓←↑↑↑↓↓↓←←↑→↑↑
sokoban-small-v1 34	↓↓←↓←←↑→↑→↓→→↑↑←↓→↓←	↓↓←↓←←↑→↑→↓→→↑↑←↓→↓←
sokoban-small-v1 35	↓↓↑↑←←↓→↓↓→↑←↑↑→↓↓	↓↓↑↑←←↓→↓↓→←↑↑↑→↓↓
sokoban-small-v1 36	↓↓↓↑→↓↑←↑↑→↓	↓↓↓↑→↓↑←↑↑→↓
sokoban-small-v1 37	←←→→↑↑←↓→↓←↑←←↑→	←←→→↑↑←↓→↓←↑←←↑→
sokoban-small-v1 38	↑↓→→↑←←↑←↑←↑→→→↓←↓←↑	↑↓→→↑←←↑←↑←↑→→→↓←↓←↑
sokoban-small-v1 39	→→↑←→→↓↓↑←↓↓←↓→	→→↑←→→↓↓↑←↓↓←↓→
sokoban-small-v1 40	←↓←↓↓→→↑	←↓←↓↓→→↑
sokoban-small-v1 41	↓↓→→↑↑→↓←↓→←←←↑→←↑→	↓↓→→↑↑→↓←↓→←←←↑→←↑→
sokoban-small-v1 42	→↑→↑↑→→↓←↑←←↓↓→↑←↑←↑→→←↓→→	→↑→↑↑→→↓←↑←←↓↓→↑←↑←↑→→←↓→→
sokoban-small-v1 43	←→↑←←→↓←←↓←↑↑↓→→→→↓↓←↑→↑←←←↑	←→↑←←←↓←↑→→→↓←→→↓↓←↑↑→↑←←↓←↑
sokoban-small-v1 44	↑↑↓↓→↑↑	↑↑↓↓→↑↑
sokoban-small-v1 45	↑↑→↑←↓↓↓→↑↑←↑←↑←←↓↓→→→	↑↑→↑←↓↓↓→↑↑←↑←↑←←↓↓→→→
sokoban-small-v1 46	→→→↑↑↓←↓←←↑→→	→→→↑↑↓←↓←←↑→→
sokoban-small-v1 47	←↓↓↓→↓←↑↑↑↑←←↓→↑→→↓↓←↑→↑←	←↓↓↓→↓←↑↑↑↑←←↓→↑→→↓↓←↑→↑←
sokoban-small-v1 48	↑←↑→↑→↑←←↓↓	↑←↑→↑→↑←←↓↓
sokoban-small-v1 49	↓←←↓←←↑↑→↓↑→	↓←←↓←←↑↑→→←↓
sokoban-small-v1 50	→↓→→↑↑↑↓↓↓←←↑→↑↑	→↓→→↑↑↑↓↓↓←←↑→↑↑
sokoban-small-v1 51	↑←↑↓→↑↑→→	↑↑↓←↑→↑→→
sokoban-small-v1 52	↑←↑↑↑→→↓←→↓↓←←↑→	↑←↑↑↑→→↓←→↓↓←←↑→
sokoban-small-v1 53	↑↑→↓↑↑→→↓←↓	↑↑→↓↑↑→→↓←↓
sokoban-small-v1 54	↓↓↓↑→→↑←→→	↓↓↓↑→→↑→←←

sokoban-small-v1 55	↓↑←←↑↑→→↓↓→↓↓←↑→↑←←↑	↓↑←←↑↑→→↓↓→↓↓←↑→↑←←↑
sokoban-small-v1 56	↓↓←↓↓→→→→↑←↓←←←↑↑→↑↑←↓→↓↓←↓→→→↑←↓←↑↑	↓↓←↓↓→→→→↑←↓←←←↑↑→↑↑←↓→↓↓←↓→→→↑←↓←↑↑
sokoban-small-v1 57	↑↑→→↓←→↓←←↑	↑↑→→↓←→↓←←↑
sokoban-small-v1 58	←↓←←↑←↓↓↓↑→↑→→↑←←	←←←→→↓↓←←↑↑→↓←←↓↓
sokoban-small-v1 59	↓←→↑↑↑↑→→↓←↑←↓↓↓→↑↑→↑←←	↓←→↑↑↑↑→→↓←↑←↓↓↓→↑↑→↑←←
sokoban-small-v1 60	↓→↓↓←←↑←←↑→↑→↓	↓→↓↓←←↑←←↑→↑→↓
sokoban-small-v1 61	→→→↑→↓←←←←↓↓→↑←↑→→←←↑→→↓→	→→→↑→↓←←←←↓↓→↑←↑→→→←←←↑→→
sokoban-small-v1 62	→→←↓→←↓←←↑→	→→←↓→←↓←←↑→
sokoban-small-v1 63	↑↑↑↓←↑↓→↓↓←↑	↑↑↑↓←↑↓→↓↓←↑
sokoban-small-v1 64	←↓←←↑↑→←↓↓→→↑→→↓←←↑←	←↓←←↑↑→→↓←→→→↑←→↓↓←←
sokoban-small-v1 65	↓→→→↓↓←↓←←↑→→↓→↑↑↑↓↓←↑↑←←↓	↓→→→↓↓↓←←←↑→→↓→↑↑↑↓↓←↑↑←←↓
sokoban-small-v1 66	↓→→↓↓←←↑→↓→↑	↓→→↓↓←←↑→↓→↑
sokoban-small-v1 67	←↑→↑↑←←↓→	←↑→↑↑←←↓→
sokoban-small-v1 68	←←→↓←←→↓→	←←→↓←←→↓→
sokoban-small-v1 69	←→↓←	↓←→↑←
sokoban-small-v1 70	↑←↑↑→↓↑→→↓←←→↓↓←↑↑→↑←	↑←↑↑→↓↑→→↓←←→↓↓←↑↑→↑←
sokoban-small-v1 71	↓↓←↑↓↓↓→←↑→	↓↓←↑↓↓→←↓→
sokoban-small-v1 72	↑↑→↑→↑←↓←←↓→↓↓←↑↑←↑→→	↑↑→↑→↑←↓←←→↓↓↓←↑↑←↑→→
sokoban-small-v1 73	←↓↓↑↑←←↓←↓→	←↓↓↑↑←←↓←↓→
sokoban-small-v1 74	→↓↓→→↑←	→↓↓→→↑←
sokoban-small-v1 75	↑←→↓←	←→↑←
sokoban-small-v1 76	↓←←↑←←↑↑→→↓←↓←↑↓↓→→↑↓→→↑←←←↓←↑→↑↑→↓	↓←←←←↑↑↑→→↓←↓←↑↓↓→→↑↓→→↑←←←↓←↑→↑↑→↓
sokoban-small-v1 77	→↓↑←↓←↓↓↑↑→↓→	↓↑→↓←↓→←↑←↓↓
sokoban-small-v1 78	→↓↓↑←↑←↓↓↑↑→→→↓→↓←←←	↓↓→↓←↑↑↑→→↓→↓←←←↑↑→↓↓
sokoban-small-v1 79	←↑↑↓↓↓←←↑↑→←↓↓→→↑↑	←↑↑↓↓↓←←↑↑→←↓↓→→↑↑
sokoban-small-v1 80	↓↓←↑←←←↓↓↓→→→↑↑→↑←←←↑←↓↓↑→→→→↑←←	↓↓←↑←←←↓↓↓→→→↑↑→↑←←←↑←↓↓↑→→→→↑←←
sokoban-small-v1 81	→→←←↓↓→↑→	→→←←↓↓→↑→
sokoban-small-v1 82	→↑→→→↓↓←↑↓→↓↓←↑→↑↑↑←←←↓→→↑→↓↓↓←↑↑←↑→	→↑→→→↓↓←↑↓→↓↓←↑→↑↑↑←←←↓→→↑→↓↓↓←↑↑←↑→
sokoban-small-v1 83	↓↑←↓←↓↓→→←←↑↑→→↓	←↓←↓↓→→←←↑↑↑→→↓↓
sokoban-small-v1 84	←↓↓←←↑↑↑↓↓↓→→↑←↓←↑↑↓→↑↑	←↓↓←←↑↑↑↓↓↓→→↑←↑↓↓←↑↑→↑
sokoban-small-v1 85	←↑↑→→→↓↓←←↑→←←↑→	←↑↑→→→↓↓←←↑←↑→↓→
sokoban-small-v1 86	→→←←↑→→←↓←↓→→	→→←←↑→→←↓←↓→→
sokoban-small-v1 87	←→↑↑↑←←↓→	←→↑↑↑←←↓→
sokoban-small-v1 88	→→→↑←↑←←↑→→↓↓↓←←↑↑	→→→↑←↑←←↑→→↓↓↓←←↑↑
sokoban-small-v1 89	←↓←←↓↓→→↑↑↑→↑←←↓↓→↓←	←↑←↓↓←↓↓→→↑←→↑↑→↑←←
sokoban-small-v1 90	→↑→↓↓↓←↑↑→↑←←↓→↓↓↓←←↑←↑↑→→↑→→↓↓←↑	→↑→↓↓↓←↑↑→↑←←→↓↓↓↓←←←↑↑↑→→↑→→↓↓←↑
sokoban-small-v1 91	↑→↑↑←↑←↓↓↑→→↓↓←↑↓←←	↑→↑↑←↑←↓↓↑→→↓↓←↑↓←←
sokoban-small-v1 92	←←↓→↓←↑↑↑←←↓↓→	←←↑←←↓↓→←↑↑→→→↓↓↓←
sokoban-small-v1 93	→→↓→↓←←←↓←↑→→→↑↑←←↓→→	→→↓→↓←←←↓←↑→→→↑↑←←↓→→
sokoban-small-v1 94	←←←→→↓←←→→→↓←←←	←←←→→↓←←→→→↓←←←
sokoban-small-v1 95	→↓←↓←←↓→↑→→	→↓←↓→←←←↓→
sokoban-small-v1 96	↓→↓←←←↑↓→→↑↑→↑↑←↓↓↓→↓←←	↓→↓←←←↑↓→→→↑↑↑↑←↓↓↓→↓←←
sokoban-small-v1 97	→→→↑→↑←←←↓←↑↑↓→→→→↑↑←↓→↓←←←	→→→↑→↑←←←↓←↑↑↓→→→→↑↑←↓→↓←←←
sokoban-small-v1 98	←↓↓↓↑→→↑←←→→→↑←←←	←↓↓↓↑→→↑←←→→→↑←←←
sokoban-small-v1 99	↓→→→←←←↑→←↑↑→↓↓→←←↓→→	→←↓→→→←←←↑↑↑→↓↓→←←↓→→
sokoban-small-v1 100	←↑←↑↑←←↓→→→↓↓↓←←↑↑↑←↑→	←↑↑←↑←←↓→→→↓↓↓←←↑↑↑←↑→

A.3 Sokoban-v0

A.3.1 Unsolved

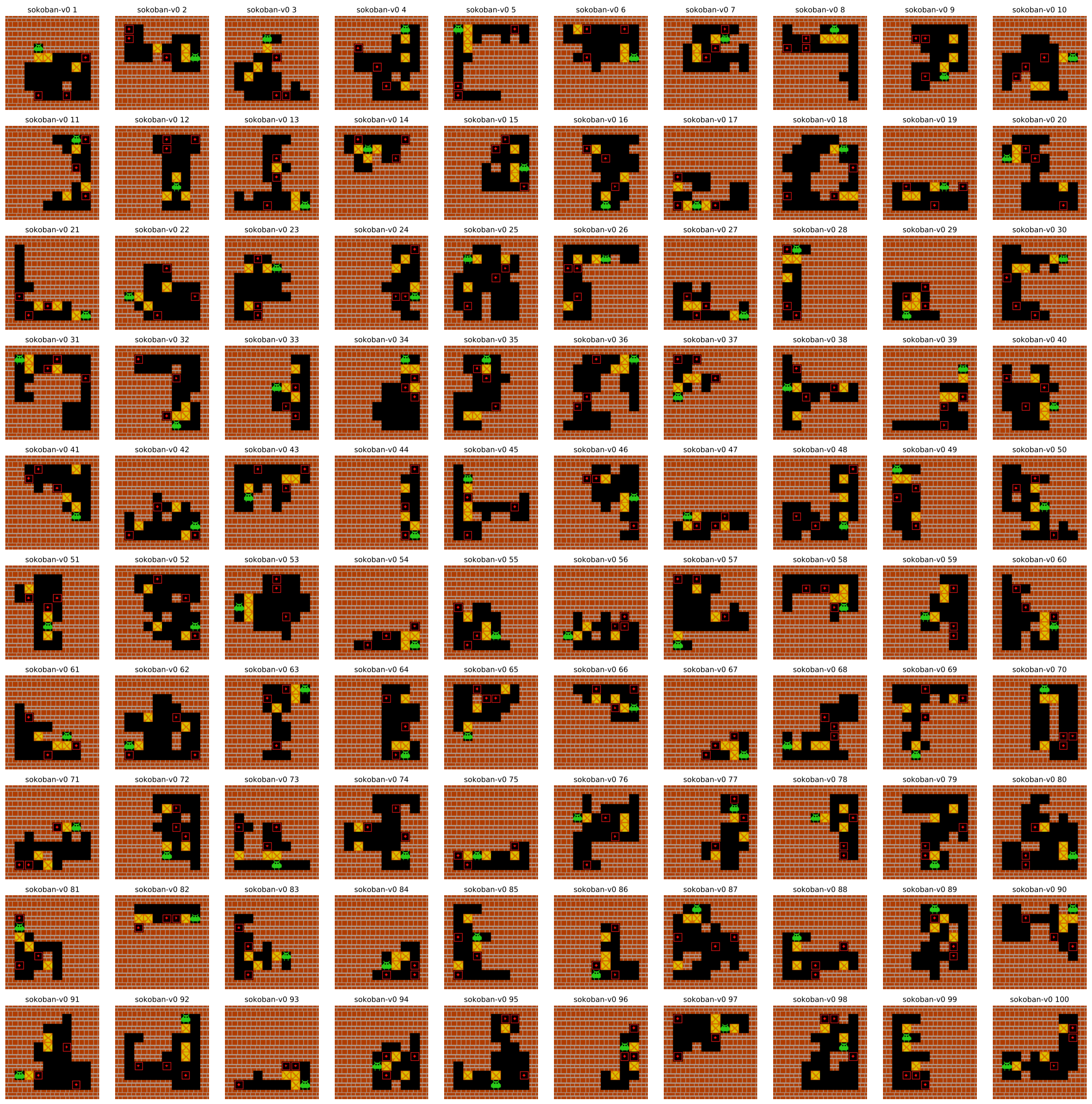


Figure 24: loss of the "10 $\times$ 10 grid with two boxes" model with a batch size of 1

A.3.2 Solutions

Table 11: Solutions for Sokoban-v0.

game name	DeepCubeA (10 × 10 with 2 boxes model)
sokoban-v0 1	↓↓↓↓↑↑↑→→→→↓↓↓→↓←
sokoban-v0 2	←←↑←←←←↓←↑↑→↓→→→→↑→→↓←←←←←←↓←↑
sokoban-v0 3	↓↓←↓←↓←↓→→→↑↑←↑→↓←↓↓←↓→→→←↑↑→↓
sokoban-v0 4	↓→↓←↓→↓←←↑→↑↑←↓↓←↓↓←↓↓→→→↑↑↑←←→→→↑←←←←←↓←↑
sokoban-v0 5	→→↓←↑←↓↓↓↓↓↓↑↑↑→↑←↑↑→→↓←↑←↓↓↓↓↓↑↑↑↑↑→→→→→
sokoban-v0 6	←↑↑↓↓←←←↑↑←←←↑→
sokoban-v0 7	←↓↓←↓←←↑→←↑→→↑←↑→→
sokoban-v0 8	↓←←→→↑→→↓←↓←←←↓←←←→→→↑↑→↓→↓←←←
sokoban-v0 9	↑→↑←→→↑↑←←←←↓→↓↓→↓←↑↑→→↑→↑←←←
sokoban-v0 10	←←↓↓↓↓←←↑↑→↑←←↑←↓→→↓↓↓→↑↑→↑←←
sokoban-v0 11	↓↓→↓↓↓←↓↓→↑↑←↓↓←←↑→→↑→↑↑↑↑
sokoban-v0 12	↑↑↑↓↓↓↓↓←↑↑↑↑↑↑↓↓↓↓↓↓→→↑↑↑↑↑↑←↑→
sokoban-v0 13	←←↑←↑↑↑↓↓↓→↓→→↑←←↓←↑↑
sokoban-v0 14	→→↑←←←←↓→→←↓↓→↑↑←↑→→→→↓←←←↓←↑
sokoban-v0 15	←↓←↑↑↑↓→→↑←←↓↓↓↓→
sokoban-v0 16	↑←↑→↓→→↑↑↑→↑←←↑←←
sokoban-v0 17	→←←↑←↑
sokoban-v0 18	←←←↑←↓↓↓↓↓←↓↓→→↑←←→→↓→→→→↑←→↑↑
sokoban-v0 19	↓←←↑←←↓→→→→→↓→↑←←↑←←←↓←←↓→→→
sokoban-v0 20	→→←←↑→↑→→↓↓↓↓↓↓←↓→→→↑→→↑←←←↓←↑↑↑
sokoban-v0 21	←←←←↑←←↓←↑→→→↓←→→→↑←
sokoban-v0 22	→→←↓→↑→↓↑→↑↑↓↓→→
sokoban-v0 23	←↑←←↓↓←↓→↑→↑↓↓↓←↓←↓→
sokoban-v0 24	↑←↑↑→↑↑←↓←↓↓↓→→↑↑↑←↓↓↓
sokoban-v0 25	→→↑→↓↓→→↑←←↑←↓←↓→→←←↓←↓↓
sokoban-v0 26	←↑←←↓↓↓↓→↓↓←↓←↑↑↑↑→↑↑↑→→↓←↑←↓
sokoban-v0 27	←←←←↑↑←←←↓↓→↑→
sokoban-v0 28	↓↓←↓↓↓↑↑↑↑→↓↓↓↓↓
sokoban-v0 29	↑→↑↓←←↑↑→→
sokoban-v0 30	←←←←←↑←↓↓↑→↓↓↓↓↓←↓→→↑←↑↑↑↑↑→→↓←↑←↓↓↓↓
sokoban-v0 31	→→→↓→→→↑→↓←←←←↑←←←↓→→
sokoban-v0 32	↑↓→→↑←←↑↑↑↓↓↓→↑↑↑↑↑→↑←←←←←
sokoban-v0 33	↓↓→→→↑←↑↑→↑↑←↓↓↓↓↓←←↑↑→↓
sokoban-v0 34	↓↓→↓↑←↑↑→↓
sokoban-v0 35	←↓↓↓↓↓←←↓↓→↑↓→↑←↑→←←↑→→↓→↑↑→↑↑←
sokoban-v0 36	←←→↓←←←↑←↓↓↓↓↓←↓↓→→↑←↓←↑→↑↑↑→↑↑→→→→↓↓←↑→↑←←←←↑←↓↓↓↑↑→→→↑←
sokoban-v0 37	↑→↑↑←↑→↓↓→→←←↓←↑↑
sokoban-v0 38	↓↓↓↓→↑↑←↑→↑↑↓→→→→↓→→↑←
sokoban-v0 39	↓↓←↓↑←↓→↓↓→↓←
sokoban-v0 40	←↑↑↑↑←↓↓→↓↓←↓↓→↑↑↑←←←↓←↑↑
sokoban-v0 41	↑←↑↓→↑↑←→→↑←←→→↑←←←←→→↓←←←
sokoban-v0 42	←←←←↓←←←↑→→→→↓→→→↑↑↑↑↑↑←←←←→↓↓→↓←←←←←
sokoban-v0 43	↑↑←↑→→↓→→↑→↓↑→→↓←←←←←↑→→→→←←↓←←↓←↑←↑→
sokoban-v0 44	↑↑↑←↓→↑↑↑↑←↓↓→↓↓←↓
sokoban-v0 45	↓←↓↓→↓↓↑↑→→→→
sokoban-v0 46	↑←←↑←←↓→→↓→↑→↓↓↓←↑↑↑→↑←←←
sokoban-v0 47	→↓→→↑→→↓←↑←←↓↓←←←↑→↑→→→
sokoban-v0 48	←↑↑→→↓←→↑↑↑←↑←↑→↓↓↓↓←↓↑→→↓←↓←←←←↑←←↓↓→↑→→→→→↑↑←↓→↓←←
sokoban-v0 49	↓↑→↓←↓→↓→↓↓
sokoban-v0 50	←→↓↓↓←↑↑←↑↑→↑←→↓↓←↓→↑→↓↓←↓→
sokoban-v0 51	↑↑←↓↓↓↓→↑↑↑←↑↑←←↑→↓→→↑↑←↓→↓←
sokoban-v0 52	←←↓→←←←↑↑↑↑←↑←↑→↓→→→↑→↓
sokoban-v0 53	→←↑→→→↓→↑↑←↓←←↓→→←←←↓→→→→↑↑
sokoban-v0 54	←←←←→→→→↑↑←←←

sokoban-v0 55	←→↑←←↑←↓↓←↑↑
sokoban-v0 56	→↑→→→↓→↓→↑↑↓←←↑→↓←↓←←↑←↑→→→
sokoban-v0 57	↑↑↑↑↑↑↓→→↑↓↓→↓→
sokoban-v0 58	→↑↑←←↓↑←←→→↓→↓→↓↓↓←↑→↑←←
sokoban-v0 59	→→↑←↑→↓→↓↓↑→↑↑↑↑←←↓↓↓←↓→↑→↑↓↓
sokoban-v0 60	↑←←←↑←←↓→↓→↑→→→↓↓↓←←↑→↑→↑←↓↓↓→↑↑←↑←←↓←↑↑↑
sokoban-v0 61	↓→↓←↑←←←↓←↑↑↑→↓←↓→→→→↓←
sokoban-v0 62	→→↑→↓→↓←←←→↑↑↑←↑→→→↓→→↑→↓↓↓
sokoban-v0 63	←→↓←←←↓↓↓↓
sokoban-v0 64	←↑↑↑↑↑←↑↑→→↓←→↓↓←↓↓→↓→↓←
sokoban-v0 65	↑↑↑→→↑↑←←↓←↓→→↓→↑↓→↑→↑→↑←←←
sokoban-v0 66	↓←←↑↑←↑←←↓→→→↓→↓→↑↑←↑←←→↓
sokoban-v0 67	←↑←→↓←←
sokoban-v0 68	→↓→→↑↓→↑↓←←←↑→→→↓→↑↑↓←↑↓←←↑→→↑
sokoban-v0 69	↑↑↑↑←↑↑→↑→→→↓→←↑←←←↓↓←↓→↑←↑←↑→→
sokoban-v0 70	↓→→↑→↓↓↓↓←↑↑↑←←↓↓↓←↓↓→→↓→↑←←←↑↑→↓←↓→→
sokoban-v0 71	←↓↓←↓↓←←←↑↑→↓↑→→↓↓←←
sokoban-v0 72	↑↑↑→↑↑↑←←↓→↑→↓↓↓←↓↓→↓→↑←←→↑←↑↑→↑↑←←↓→
sokoban-v0 73	↑↑↑↓↓↓←↑↓←←←↑↑↑
sokoban-v0 74	↓←↑↑←↑→→→←←←←←↑→↓←↓→→→→↓→↑↑↑↑
sokoban-v0 75	←→→↓→→→→↑↑←↓→→→↓←←←←←→↑→→↓→↑
sokoban-v0 76	→→→↑→↓↑→→↓←←↓↑→→↓←←←←←↑←↓↓↓↓
sokoban-v0 77	→↓↓←↓↓←←←↓→↑→→↓↓←↑→↑←→→↑↑←↑↑
sokoban-v0 78	→→→↓↓↓↑↑←↑↑↑→↓↓↓↓
sokoban-v0 79	→↑←↑↑↑→↑↑←↓↓→↓←↑←↓↑→↑→↑→↑→↓↓
sokoban-v0 80	←←←←→→→→↑←←↑↑←↓↓↓→↓←←
sokoban-v0 81	↓→↓→→→↓↓←←←↑←↑↑↑↓→↓←↓→
sokoban-v0 82	←↑←←←←←↓→→
sokoban-v0 83	←←↓←←↓←↑↑↑↑→↓↓↓→→↑←↓←↑
sokoban-v0 84	↓→→↑→↑↑←↓→↓↓←←←↑↑→→↑→↓←↓↓←↑←↓→↓→
sokoban-v0 85	←←↑↑↑→↓↓↓→↓←↓←↓↑↑
sokoban-v0 86	↑→→↑↑←↓↑→↑↑←↓↓→↓↓↓→→↑←↓←↑↑↓↓←←↑→↑→↑↑←↓↓→↓←
sokoban-v0 87	↓↓↑↑←↓↓↓→→→→↓←↓←↓↓←←←↑↑↑→→→
sokoban-v0 88	↓←↓→→→↓→→↑→→↓←←←←↑→→↓→↑←←←←←↓→←←↓→→
sokoban-v0 89	↓↓↓↓←↓→→←↑↑↑↑↑→→↓←→→↓↓↓↓↓←↑→↑↑←←↑←↓↓←↓→→
sokoban-v0 90	↓←↓↑→↓↓←↑↑←←←
sokoban-v0 91	→→→↑↑←↑→→↑↑←↓↓↓↓→↓←↓→→
sokoban-v0 92	↓→↓↓←↓→↓←←←←←→→→→↑→↑↑↑←↓↓↓↑↑←↓↓→↓←
sokoban-v0 93	←↑↓←↑↓←←←←←←←
sokoban-v0 94	↑→↓→→↑←↓↓→↓→↑↑
sokoban-v0 95	←↑→→→↑←↑↑↑↑←↑→↓↓↓↓↓←↓←←↓←←↑→→→→↓→↑↑↑↑↑↑
sokoban-v0 96	↓→↑↑↓↓←↓↓↓←←↑→↓→↑↑↓→↑
sokoban-v0 97	↑←↓→↑→→↓←←←←→↑←←←→↓→↓←←↑←↓↓
sokoban-v0 98	→↑←↓←↑↑↓←↑↓↓↓↓←←↓→→→→↓→↑↑
sokoban-v0 99	↓↓↓↓←↑↑↑↑↑↑→↓↓↓↓↓↑→↓←←↓↓→→
sokoban-v0 100	→→→↑→→→↓→↑←↑↑→↑↑←↓↓↓↓→↑↑↓↓→↓←↓←←↑←↑→↑→→↓→↑↑

A.4 Sokoban-v1

A.4.1 Unsolved

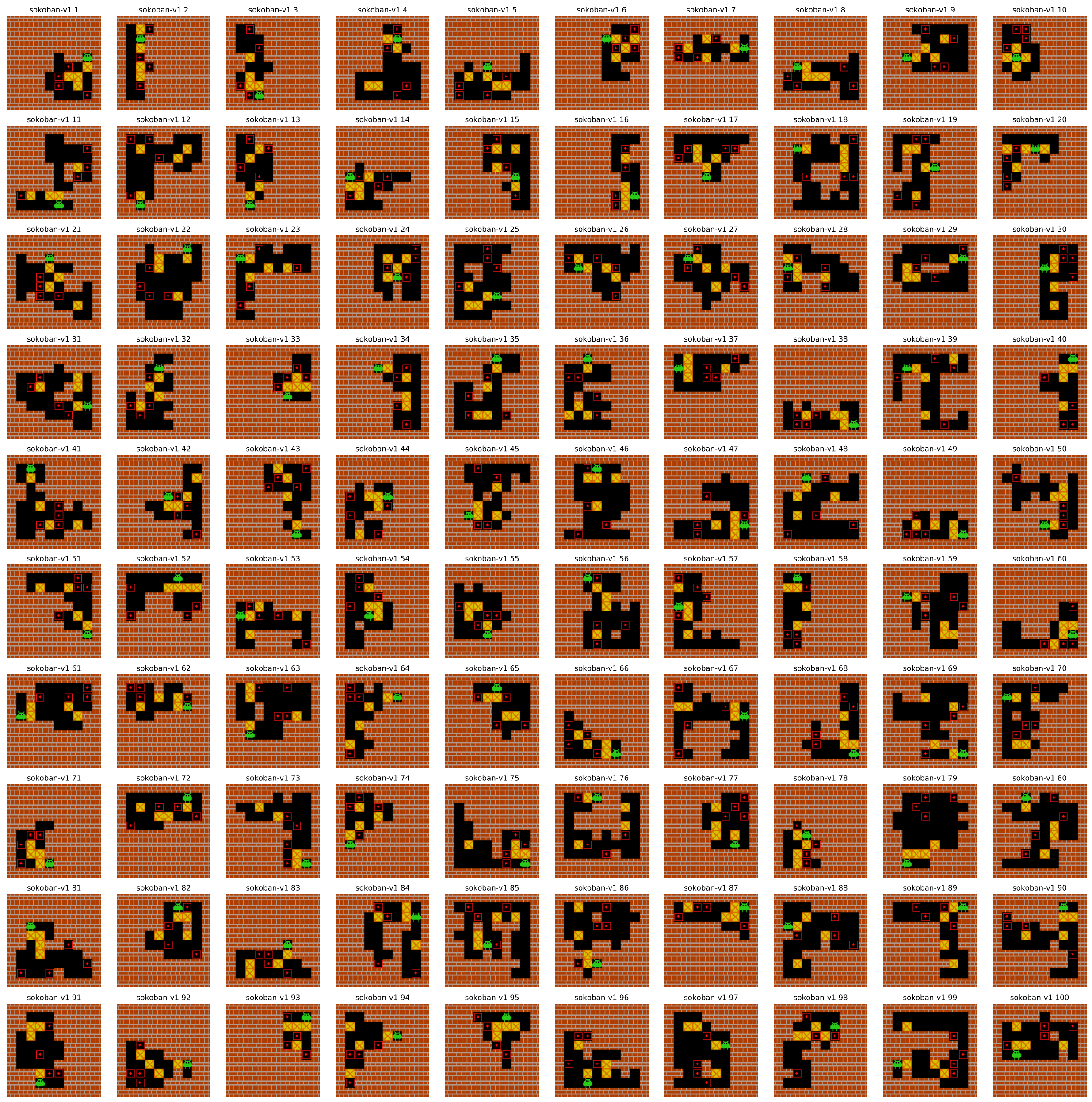


Figure 25: loss of the "10 $\times$ 10 grid with two boxes" model with a batch size of 1

A.4.2 Solutions

Table 12: Solutions for Sokoban-v1.

game name	DeepCubeA (10 × 10 with 2 boxes model)
sokoban-v1 1	↓↓↓←←→↑←→→↑←
sokoban-v1 2	←↑→↓↓←↓↓→↓
sokoban-v1 3	↑←→↑↑↑↓←↓↓↑↑↑↑↑
sokoban-v1 4	↑←↓→↓↓↓←↓←←↓→→←←←↓→→→↑→↑↑↑↑↑
sokoban-v1 5	↓→↓↑→→↓←↓←←←←↑←↑→→→→↓→↓←↑←←←
sokoban-v1 6	→↓↓→→↑↑↓↓←←↑→↓←←↓↓→↑↑
sokoban-v1 7	←←←↓←←←→↑↑→↓→↓←↑←←
sokoban-v1 8	→→↓←→→→→↑→↓←←←↓←←↑←↑→→→→
sokoban-v1 9	→↓→→↑↓←←↑→↑→↓↑←↑→→↓↓
sokoban-v1 10	↑→→↓↓←↑↑↑↓↓↓↓←↑←↑→↓→↑↑←↑
sokoban-v1 11	↑↑↑↑↑←↑↑→↓←↓→→→↑→↓←↓↑←←↓↓↓↓↓←←↑←→→↓→↑↑↑↑↑←↑→→→
sokoban-v1 12	↑→↑←↑←↓↑↑↑↑→↓→↑→→→↑→→↓←←→↓←←→↑←←←←↓←↑
sokoban-v1 13	↑↑↑↓↓→↑→↑↑↑←↑↑→↓↓
sokoban-v1 14	↓→→←↑→→←↓↓↓←↑↑
sokoban-v1 15	↓↑↑↑↑→↑←←↓←↓←↓→
sokoban-v1 16	←↑→↑↑←↑↓→↓↓↓↓←↑↑↑
sokoban-v1 17	→↑↑←↑←→→→→→↓←↓↓←↑↑←←→↑→→
sokoban-v1 18	→→↓→→↑→↑→↓↓←↓↑←←←↑←↓↓↑↑→→→→↓→↑↓↓
sokoban-v1 19	←↑←↓↓→↓↓←←→→↑↑↑↑↑↑←↓↑←←↓→→↓→↑↓↓↓←↓↓↑↑↑←↑↑↑
sokoban-v1 20	←↑←←↓↓↓↓→↑←↑↑↑→→↑↓←←←↑↑←←↓↓↓↓↓→↑←↑↑→→↓←↑←↓→→→↑→→↓←←←
sokoban-v1 21	↓→↓←↑←←↓↓→↓→→→→→↓↓←↑→↑←←←←→↑←↑↑→↓↓
sokoban-v1 22	↓↓→↓←←↓→↓←←←↑↑←↑↑→↓↓↑↑→→↓→→↑↑←↓→↓←←←←→↓↓←←
sokoban-v1 23	→↓→↑→→↓→←←←←↓↑↑←↓↓←↓↓
sokoban-v1 24	↑←←↓→→↑←↑←↑→→→←↓→↓←
sokoban-v1 25	↓←↓←↑→↑↑↑↑↑←←←↓↓↓→→←↓→→↓→↑↑↑↓↓←↑↑↑↓↓↓↓←←↓←↑↑
sokoban-v1 26	→↑→→→↓↓↓←↑←↑→←←↑←
sokoban-v1 27	↓→↑→↓←↓→→→→↑←←←←↓↓→↓→↑←↑→→↑→→↓
sokoban-v1 28	→←↓↓→↑↑→↑↑←←↓↓→→→→↓→→↑←
sokoban-v1 29	←←←→→↓↓←←←↓←↑↑↑←←↓→←↓→→→←←↑↑↑→→→→→
sokoban-v1 30	→↓←↓↓→↑←↑→→↑←↑→↓↓←↓↓→↓↓←↑↑↑↑↑←↑→←↑→
sokoban-v1 31	←→↑↑↑←←←←↓←→↑→→→↓↓↓←→→↓←
sokoban-v1 32	←↓↓↓↓↑↑→→↑←↑←↓↓→↓↓←←↓↓→→↑←→↑↑↑→↑←↓←↓
sokoban-v1 33	↑↓→→↑←←→↑
sokoban-v1 34	→→↓↓→→↓↓←↑→↑↑←↓↓↓↓
sokoban-v1 35	↓↓↓↓←↓←←↓↓→→↑→↑←←→↓↓←←↑↑→→↑↑→↓↓←↓←→→
sokoban-v1 36	↓→↓←↑←←↓↓↓↓→→←↓↓→→↑↑←←↓→←↓←↑↑↑↑
sokoban-v1 37	→←↓→→→↑→↑→→↓←←↑←←↓←↓→←←↑↑→→↑→→→
sokoban-v1 38	←←←←←→↑←→→↓→→→↑↑←↓←←→→→↓←←←←
sokoban-v1 39	→→↑→↓→→→↑←←↓←↓↓←↑→↑←←←→→↓↓↓↓→↓↓←↑↑↑↑↑→↑←←↑←←↓→→→→→←↓↓↓↓↓←↓→
sokoban-v1 40	↓↓←→↓↓↓↓↑↑↑↑↑↑←↓↓←←←↓→↑→↓↓←↓↓
sokoban-v1 41	↓↓↓↓→→↓→↓←↑←↑←←↓→→→↓→→↑→→↓←←
sokoban-v1 42	↓←↓→→↑→→↓←→↑↑↑←↓↑↑↑→↓↓↓↓↓↓
sokoban-v1 43	↑↑↑←↑↓→↑↑←←↑←↑→→→↓←
sokoban-v1 44	↓←←←↓←↓↓→→←←↑↑→↑↑←↓→→→→↑↑←↓←↓→↑←←
sokoban-v1 45	→↓→→↑↑←↓←↑↑↑↑↑→→→↓←←↑←↓↓↓↓↑→→↑↑
sokoban-v1 46	←↓↓↓↓↓↑↑→↓↑↑↑→↑←→→↓↓↓→↓←←←↓←↓→↓←
sokoban-v1 47	←↑↑→↑←←←→→↓↓↓→↓←←←←→→↑←←→↑→→
sokoban-v1 48	↓↓↓←←↑→→↓→↑←↑→↓→↑→→↓←←↑←↓↓←↓←↑←↓↓→→→→→→
sokoban-v1 49	←←←←←↑↓→→→↑↑→↓→↓←←←←↑→→↑→↓→↓←↓←←←
sokoban-v1 50	→↓→↑→↑↑←↓→↑↑↑←↓↓↑↑←←←←↓←↑→→→↓→↑→↓
sokoban-v1 51	↑↑←←→→↑↑←←↑←↑→←←←←↓→→→→
sokoban-v1 52	↓←←←←↑←↓↓↓↑↑→→→→↑→→↓↓↓←↑→↑↑→↓↓↑←←←←←←
sokoban-v1 53	→↓→→→→→→↑←←↓←←↑↑←→↓↓←←←↓↓→↑←↑→→→→→→↑→↓↓
sokoban-v1 54	↑↓↓→→↑↑←↑←→↓←←↑←↑↑→→↓↓←↑↓↓↓→↓→↑↑←↓←↑↑

sokoban-v1 55	↑←↑→↑↑←↓←↓→←←↓→
sokoban-v1 56	→↓↓↓↓←↓→↓↑→→↑↑←↑↑→↑←↓←↓↓↓
sokoban-v1 57	↑→→↓↓←↑↑↓↓↓←↓↓→↑↑↑↑←↑→→←←↑→
sokoban-v1 58	↓↓↑↑←↓↓↓↓↓↑→↓↓↑↑→↑↑←↓↓↓
sokoban-v1 59	→↓↓→→↓↓↓→→↑↑↓←↑↓↓←↑↑→↑←↑→→←↓↓→↑↑
sokoban-v1 60	↑↑↓↓↓←←↑↑→↑→↓↓↑←↓
sokoban-v1 61	↑→←↓→→↑↓→→↓→→↑↑↓←↑↑→←↓↓↓←↑↑←↑→↑→→
sokoban-v1 62	←←↑←←←→↓↓←↑↑→→→→↓←←←↓←↑←↑
sokoban-v1 63	↑↑←↑↑→→→←←←↑→→→→↓↓→→↓→↓↓←↑←←←↓↓←←↑↑↑←↑→
sokoban-v1 64	←←←←↓←↑→→→↓←←↓←↑↓↓↓↓→↑↑↑↑
sokoban-v1 65	↓←→↓→→↓↑←←↓→↓→↑↑→↑←
sokoban-v1 66	←←↑←←↓←↑↑→↓→→↓→→↑←←←←↓→
sokoban-v1 67	↑←→↓←←←↑←←↓←←↑↑→↓↓↑→→→↓←←←↑←↓↓↓↓
sokoban-v1 68	↑←←←↓←↑→→→→↑↑↑↓←↑↑↓↓↓→↓←←←
sokoban-v1 69	←←←↑↑↑←↑→→↑→↑↑←←←↓↓↓
sokoban-v1 70	→↑→↓↑→→↓←←↓↓↑←↑←↑→→←↓←↓↓↓→→↓←←↓↓→↑↓→↑↑
sokoban-v1 71	←←→↑↑←→↓↓←↑↑
sokoban-v1 72	→↓←↑←←←←←↓↓→↓←→→↑→→→
sokoban-v1 73	←↑↑→↑↑←↓→↓←↑↑↑↑←←←↓←←↑→→→→↓→→↑↑←↓↓↑←←←↓→
sokoban-v1 74	↑↑↑→↑→→→↓←↑←←↓↓→↑←↑→
sokoban-v1 75	↑←↑↓↓←←←←↑←←↑↑→↓↓↓←↓→→→→
sokoban-v1 76	←↓←←↓↓↓→↓←↓→↓→→→↑→→→→↑↑↑←↓↓↑↑→↑↑←↓↓↓→↓↓←←←←
sokoban-v1 77	↑↑↑↑↓←←←↓→→→↓→↑←←↓↓←↑↑←↑→←↑→→↑→
sokoban-v1 78	↓←↑←↓↓↓→↑←↑→→
sokoban-v1 79	↑↑↓↓→↑↑↑↓↓↓→↑↑↑↑↑↑↓→↓↓→↓←←↓←↑←↑→↑↑←←↓→→↓→↑↓→→↓→↑↑↑↑
sokoban-v1 80	↓←↓→→→↓→↑↑→↓→↓↓←↑↓←←↓←↓↓→→↑←←
sokoban-v1 81	↓↓↓→←↑↑↑→→↓←↓↓→→←←↓←←↓→↑↑↑↑↑←↓↓↓←↓→→→→→↑↓→→
sokoban-v1 82	↓↓↓←↓←←↑→↓→↑→↑↑→↑→↓↓↓←→↓←←↑↑↑←↑→
sokoban-v1 83	↓←←↓↓→↑↓→→↑←←→↑←↓↓←←↑↑←←↓→←↓→→↑↓→→
sokoban-v1 84	←←↑←←↓↓↓↓↓↑↑↑↑↑→→↓→→↑←←←→→→↓↓↓↓↓↓
sokoban-v1 85	↑↑↑↑→→↓←↑←←↓←↓↓↓→→↑↑↑↓↓←↓←←↑↑↑→↑→↓→→↑→→↓←←←←↑→
sokoban-v1 86	←↑↑↑↓←↑↑→→←←←↑→←↑→→→←↓←↓→
sokoban-v1 87	←←←↓←←←←↑→↓→→→↑→→↓←↓→↑←←←←←
sokoban-v1 88	→→→→←↓→←←←↓←↓↓↓→↑↑↑→↑↑←↑→→→↓→→↑→↓↓
sokoban-v1 89	←←←←→→→↓←→↓↓↓↓→↓↓←↑↑↑↑↓↓←↑←↑←↑↑←
sokoban-v1 90	↓↑→↓→↓↓←←←↓←←↑↓→→↑←←←↓←↑←↑↑↓→→↑↑→→→→→→→→→→→→↑↑←↓→↓↓↑←←←←←←↓←↑
sokoban-v1 91	↑↑←↑→←←↑↑→↓→→↓↓←↓→↑↑↑↑↑↑←←↓→↓↑←↓↓←↓→↑→↑→↓↓↓
sokoban-v1 92	←←↓←←←↑↑→→↓←↑←↓↓→↑→→↓←←
sokoban-v1 93	↓↓↓←↑←↑↓→→↑↑←↓
sokoban-v1 94	←←↑←↓←↓←↓↓↓↑↑↑→→↑↑←↑←↓↓↓
sokoban-v1 95	↓↓↓↑↑←↓→→↑→↑←←←←
sokoban-v1 96	↑→↓→→↑←←↑↑→↓←←←↑↑←↑→↓↓↓↓→↓→↑↓→→↑←←←↓←↑↑↑↑
sokoban-v1 97	←←↑←←←↑↑→↓↑→↓←↓→↓↓↓←↑←↑→↑→↓↓→↓→↓←↓→↓←
sokoban-v1 98	↑←←←↓→↓→↑→↑←←↓←←↓→←↓↓↓
sokoban-v1 99	→→↑→↓→↓→→←←↑→←←←↓→→→↓→→↑↑↑↑↑←↑←←←↑←←←↓→→→→→↑→↓↑→↓↓↓
sokoban-v1 100	↑←↑→↑←←↑↑→→→→→→→→↓→↓←←↓←↑↑←↑←←↓→→↓←←

A.5 Boxoban (medium)

A.5.1 Unsolved

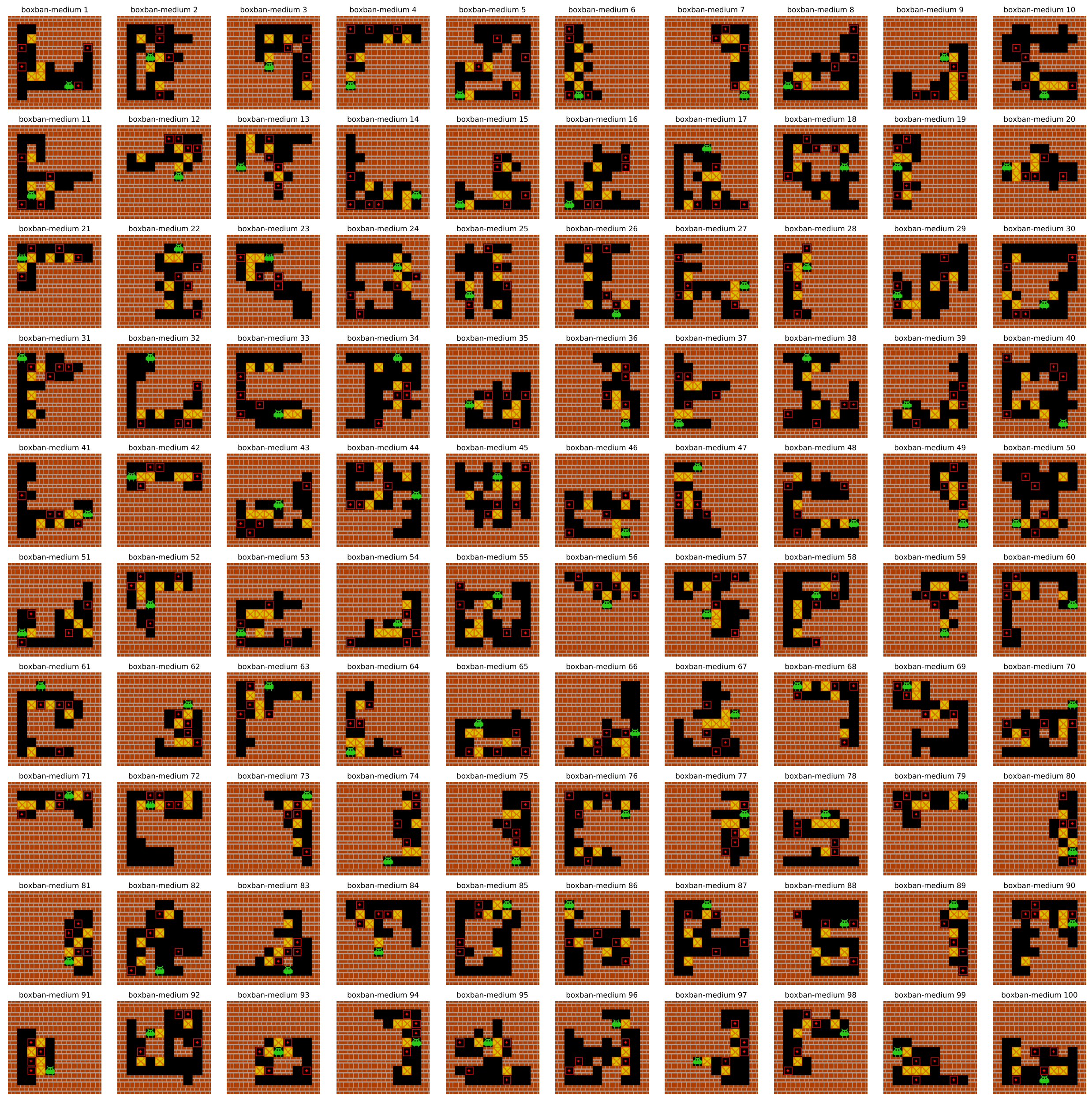


Figure 26: loss of the "10 $\times$ 10 grid with two boxes" model with a batch size of 1

A.5.2 Solutions

Table 13: Solutions for Boxoban (medium)

game name	DeepCubeA (10 × 10 with 2 boxes model)
boxban-medium 1	←←↑↑←↓←↓←↑↑→↑↑←↑↑→↓↓↓↓←↑↓↓→→→→→→↓→↑↑↑↓↓←←←←↑←←↓↓→→→→→→→
boxban-medium 2	→↓↓←↓↓→↑↑↑↑←↓↓↓↑↑↑→↑↑←↑←←↓→→←←↓↓↓↓↓↓→→
boxban-medium 3	↑←↑→→↑→↑→→↓↓↓←↓↓↓↓→↑←↑↑→↑↑↑↑←←↓→↑→↓↓↑←←←↑←←↓→→→→
boxban-medium 4	↑↑→↑↑←↑↑→→→→→↓←←↑←←←↓↓→↑←↑→→→↓←↑←←↓↓→↑←↑→↓→→→→↑→→↓←←←←↑→↓←←←↓←↑
boxban-medium 5	↑↑↑→↓→→↑↑↑→↑→↑→→↓↓↓↓↓↓↓←←↑→↓→↑↑↑←↑↑→↑↑←←←←←↓→↓↓↓↓←←←↓↓→→→→→→→
boxban-medium 6	←↑↑↑↑→↓↑←↑↑→↓←↓↓↓↓↓→→↑←↓←↑↑↑→↓←↓↓→→↑←↓←↑→↑↑←↑↑→↓↓↓↓←↓→↑↑↑←↓↓
boxban-medium 7	↑←↑↑→↑↑←↑←→→↓↓↓←↓↓→↑←↑↑→↑↑↑←↓↓↓↓↓→↑↑
boxban-medium 8	→↓→→→→→↑↑←↑←←←←↓↑→→↓←↑→→→→↓→→↑↑↑↑↓↓↓↓←→↓↓←←←←↑↑↓↓←←↑↑→↑→→→→→↓↓↓←←←←↑↑← ↑→→→→↓→→↑←
boxban-medium 9	→↓→↓↓←→↓←←←←←→→→→↑↑↑←↓←↓↑→→↓↓←←↑↑→↑↑↑→↓↓←↓←↓↓→↑↑
boxban-medium 10	→↑←↑←→↓↓←↑←↑↑↑↑←↑→→→↓→→↑←←←←↑←↓→↓↓↓↓↓→→↑←↓←↑↑↓↓→→→→↑←←←↑←↓
boxban-medium 11	↓→→↑↑↓↓←←←↑↑↑→↓↓←↓→↑↑↑→→↓→→↑←↓←↓←←↑←↓↑↑↑↑↑↑↑→→↓↓←
boxban-medium 12	↑→↑↑→↑←↓←→↓↓←↑←←←↓←←↑→→→→↑→→←←↓→↓→↑
boxban-medium 13	↑↑↑→→↓→↓→↓↓↑↑→↑↑←↓←←↑←←↓→→↑→↓→→↑←↓↓
boxban-medium 14	↓←←↑←←↑←←→→↓↓←←←↑↑→→↓→↓←←→→↑→→↓←→→→↑←←←←↑←↓
boxban-medium 15	→→→→→←↑↑→↑↑↑←↑↑→↓←↓↓→↓↓←↑↑→→→↓↓←→↑↑←←←↑↑→↓←↓↓→→↑←↓←↑↑
boxban-medium 16	↑→↓→→↑↑↑←↑→→→←↓←↓↓↓→→↑←↓←↑↑↑←↑→↓↓↓↓←←↑→↑→↓↑↑→↑←↑→→←←↓←↓↓↓
boxban-medium 17	←←←↓↓↓↓↓↓→→→↑↑→↑↓←↓↓→→→→→←←↑←↓←←↑←↑↑↑↑↑→→↓→↓↓↓↓→↓←↑↑↑→↓↓
boxban-medium 18	↓↓←←←←↑←←↑↑↑→→↑→→→→→↓←↓↓↓↓←←←←↑↑←↓→↓→↓→→↓→↑↑↑↑↑↑→↑←←←←←↓←←↓↓↑↑↑→
boxban-medium 19	→↑↓←↓↓→↑↓←↓↓→↑←↑↑↑↑↑→→↑↑←←↓↓→→↑←↓←↓↓↓↑→↑↑↑
boxban-medium 20	↓→→↑→←↓←←↑→→↓→→→↑↑←↓→↓←←←↑→↓→→↑↑←↑←↓↓↑→→↓↓←↑←←←←↓→
boxban-medium 21	↑→→→→↓→←←↑←←←↓↓↓→↑←↑→→→↓←↑←←↓↓→↑
boxban-medium 22	←↓←↓→→↓↓↓←↑↓→↓↓←↑→↑↑↑↑←↑↑→↓↓↓↓←↑↑←↑→↑→↓↓↓←↓→↓↓←↓→→←←↑↑↑↑↑←↑→↑→↓↓→
boxban-medium 23	↑←↓↓↓↑↑↑←←↓↓↓→↓→→→↓→→↑←↓←↑←←↑↑→↑↑←↓↓↓↑↑←↓←↓→↓→→
boxban-medium 24	↓↓↓→↑←↑→↓↓↓↓←←←←←←↑↑↑↑↑↑→→→→↑→↓→↓↓↓↓↓→↓←←←←←→→→↑↑↑→↑↑↑↑↑←↑→↑←←←←←↑←↓↓↓ ↓
boxban-medium 25	↑↑↑→→→↓↑←↑↑→↓↓←←←←↑→→↓→→↑↑←↓→↓←↓↓→↑↓↓→↓↓←↑→↑←↓←↑↑↑↑←→↑←←↓↓↓↓
boxban-medium 26	→→↑←↓←←←↑↑↑↑↑↑↑↑←←↓→↓→↓↓←↑↑↑←↑→→↓↓↓↓←↓→↓↓←↑↑↑↑↑↑
boxban-medium 27	←↓↑←←←←←↓←↓↓→↑←↑↑→→↓←↑←↑↑→↓
boxban-medium 28	↑←←↓↓↓→↓↓↑↑←↑↑↑→→↓←↓
boxban-medium 29	↓↓→↑→→↓→↑←←←←↑↑→↓←↓→→→↑↑↑→→↓↓↓←←←←↓←↑↑→↓→→↑↑→↓→↓←←←←→→→→↑↑↑↑→→↓←
boxban-medium 30	↓←←←←↑↑↑↑↓↓→↓→←↓↓↓→→→→→→→↑→→↓←←←←→→→→↑↑↑←↓↓→↓←←←→→↑↑↑→→↑↑↑←↑←←←←←←↓→→→ →→→
boxban-medium 31	↓→→→↑→↓→↓←↑←←←←↓↓→↑←↑→→→→←←←←↓↓↓↓→↑↑↑←↑→→→←←←↓↓↓↓↓↓→↑↑↑↑↑
boxban-medium 32	←↓↓←↓↓↓↓↓→→↑→→↓→→→↑↑↑↓←↓→↓←←←↑→←←←↓←←↑→→→→↓←←→→↓→→→↑↑←↓→↓←←←↑→↓→→↑↑←↓← ←←←↓→→
boxban-medium 33	↓→↑↓→→↑←←←←→→→↑←←←↓←←↑←↑↑↑↑↑→→↓←↑←↓↓↓↑↑→→↑→→↓←←←↑←↓
boxban-medium 34	←←↓↓→→↓↓→↓↓←←→→↑↑↑←↑↑→↓→↓→↓←↑←←←↓→↓↓↓↑↑↑→→
boxban-medium 35	↓↓→→→→↑↑←↑←←↓↓←↓→→→↑→→↓→↑↑←↑←←↓↓↓→↑↑
boxban-medium 36	↑→↑↑←↑↑↑↑←←↓→↓→↓↓→↓↓←↑↑↑←←↓↓→←↑↑→→↓←
boxban-medium 37	→↑↑←↑→←↑↑→↓←↓↓↓→↓↓←↑↑↑↑→↓→→→↑←↓←↑
boxban-medium 38	↓↓↓↓↓↑→↓→↓→→←←↑←←↓↓→←↑↑↑↑↑↑→→↓←↑←↓↓↓↓↓→↓←←↓→↑→→→↓→↑↓→↑
boxban-medium 39	→↓↓→→→↑→→↓←↑↑←↓←↓←←↑→→→→↓→↑↑↑↓↓↓←←←↑←←↓←←↑→→→→↓←→←↑→→→↓→↑
boxban-medium 40	←↑←←←←→↑→→→↓→↑↑↑↑←↑↓←↓←↓↓↓←←↓←↑↑↓↓↓↓→→→→→↓→↑↑↑→↑↑←
boxban-medium 41	↑←↓←←←←←↓←↑↑↓→→→↓←→↑→→↓←←↑←←←↓↓→↑←↑→→→↓←↑←←↓↓→↑←↑→→
boxban-medium 42	↑→↓→↑→→→→→↓↓←↑→↑←←←↓→←←←↑←←↓→→→→↑←→→→→↓↓←↑→↑←←←↓→→
boxban-medium 43	←←↓↑→→↓←↓←→↑↑←↓→↓↓←←←↑↑→→↓←→↑↑→→↓←↑↑←↓→↓↓←→→→→→↑↑↑↑
boxban-medium 44	↑←←←↓←←←→→→↑↑↑←←↓←←↓↓→→→→↑→→→↑←←↓←↓←↓←↑←←↑↑→→↑→→↓
boxban-medium 45	→↓↓↓←↓↓←←↑←←←↑↑←↑→→→→←↓→↑→→←↓↓↓→→↑↑↑↓←↑←↓↓↑→→↓↓←←←↓↓→↑←↑→↑↑←↑←←←↓↓↓→→
boxban-medium 46	←↑↑→↑↓←↓↓←←←←↑←↑↑→→→↑→↓→↓→→↑←←←←←←
boxban-medium 47	←←↓↓↓↓→←↑↑→↓→↓↓→↓←↓←←↑↑↑→↑↑←↑↑→↓↓↓→↓←→↓→↓↓←↑↑↓↓←←↑→→↓→↑
boxban-medium 48	↓←←↑←←←←→→→→↓→→↑←←←←←↓←←↑↑→↓↑↑↑←↑→↓↓↓←↑↑↑→↑→→→→←←←↑↑←↓↓↓↓
boxban-medium 49	↑↑↑↑←↑↑←←↓↓→↓→↑→↓←←↑←↑↑→→→↓↑←←←↓↓→→↑↓←←↑↑→↓
boxban-medium 50	→↓→→→↑↑←↓↑→↑↑←←←↓↓↓→←↑↑↑→→↑↑→↑→→↓←←←←→→→↓←←→↓↓↓↓↓←↑

boxban-medium 51	↓→↑↑↓↓→→→→↑→↑→↑↑←↓↓←←↑→↓→→↑↑←↓←↓↓↓→→↑←→↑↑←↓↓←→↓↓↓←←←←←←←
boxban-medium 52	↑↑↑←←↓↓→↑→↑→→↓←←↓↓↓↑↑←←↑↑→→↓↓←↑→→→↑→→↓←←←←←←→↑→→
boxban-medium 53	↑→→↓↑←←↑↑→↓↑→↓↑→→↓←←↑←←↓↓↓↓→→→→←←↑↑↑←↑→←←↓↓↓↑↑→→↓↑←
boxban-medium 54	←↓→↓→→↑↑↑↑↑←↓→↓↓↓↓←←←←←←→↑→→↑→↓→←↓→↑→
boxban-medium 55	←←←←↓↓↓→→←←↓↓→↑→↑←↓→→→→→←←↑←↓←←←↑↑↑↑→→→↓↓↓←←↓←↑↑↓↓→→→→
boxban-medium 56	↑→↑→↑←↓↓←←↑→→←←↑←←↓→→↓→→↑←→↑←←←→→↑→↑→→↓←
boxban-medium 57	→→↑→↓↓←↓←↑↑↑↑↑←↑←←←↓↓→↑→→↑→→←↓↓↑←←←←↑→→→
boxban-medium 58	←←←↓→→←←↓↓→↓←↓→→←↑↑↑↑←↑→→→→↑↓←←←↑↑→→→
boxban-medium 59	↑↑↑→↑↑↓↓←←↑←↑↑→↓↑→→↓←←↓↑↑←↓
boxban-medium 60	↑↑←↑←←←↓↓←←→→↑↑←←←↓↓↓↓↓↑↑↑→→→↑↑→→→↓→↓↓←↑↑→↑←←←←←↓←→→↓↓←←→↑←
boxban-medium 61	↓→→↓↓↓→↑←↑↑←←↓→↑←←←↓→←↓↓↓↓↓→→→
boxban-medium 62	↓↓↑←←↓→↑→→↓↓↓↓←←↑↑↑↓↓→
boxban-medium 63	↓↓↓←←↑↑↓↓→→↑↑→→↑→→↓←←←↑←↓↓←
boxban-medium 64	→↑↑←↑↑↑→←↓↓↓→↓↓→→→←←←←↑↑
boxban-medium 65	←↓←↓↓→→→→→→←↑↓←←←←←↑↑→↓←↓→→→←←↑↑→↑→→↓→←←←↑←↓→↓
boxban-medium 66	←←←↓↓←←←↑→↓→→↑↑→→→↓↓←↑←→↓←←↑
boxban-medium 67	←↓↓↓→↓←↑↑←←↑→←↑↑→↑↑←↓→↓↓↓↑↓→↓↓←
boxban-medium 68	↓→→↑→←↓←←↑→↓→→↑→→↓←←←←↑→→↓→→→↓↓↓←↓
boxban-medium 69	↓↓→→→→↓→→↑←←←←↓→→↑←←↑↑←←↓↓→←↑↑→→↓←→↓
boxban-medium 70	↓←←↓↓→←↑↑→→↓←↑←←↓←←←←↓→↑→→↓↓↓→→→→↑↑←↑←←←←←↓→↑→→→↓→→↓↓←←←←↑↑←↑→↓↓↓←↑
boxban-medium 71	↓→→↑←↓←←↑←←↓→→→→↓→↑←↑←↓←←←↑←←↓→→→
boxban-medium 72	→↑←←←↓→→↑→→→↓↓←↑→↑←←←↓←←↑→↓→→↑→→↓↓→→↑←→↑←←←
boxban-medium 73	←↓↓→↓↓←↑↓→↓↓←↑↑→↑↑←↑↑←←↓→↑→→↓↓↓↓←↑↓→↓
boxban-medium 74	→→↑↑↑←↑↑→→↑↑←↓→↓↓←↓↓→↑↑↑↑←↓↓→↓↓←↓↓→↑↑↑↑↑↓←↓
boxban-medium 75	↑↓→↑↑↑↑←↑↑→↑↑←←↓↓→↓↓→↑↑←←↑↑→↓↓↓
boxban-medium 76	↑←↑←←↓←↑←↓←↓↓↓↓↓↓→→↑←↓→→→↑←←↓←←↑↑→↓←↓→↑↑←↑↑↑↑→↑→→→→↓→→↑←←
boxban-medium 77	←↓↓←↓↓→→↑↑←↑↑→↑↑←↓↓←↓↓↓↑→→↑↑←↓
boxban-medium 78	→↓→↓←←←←→→→↑↑←↓→↓↓↓↓←←↑→↓→↑↑↑←↑↑→↓↓↓↑↑←←←↓→→↑→↓
boxban-medium 79	←←←←←↓←←↑→↓→→↑→→↓←←←←↑→→←←↓↓↓→↑↑→→→↑→→↓←←←←↑←↓→→→↑←
boxban-medium 80	↓←↑→↑↑↑←←↓→↑→↓←↑↑↑→↓←↓↓→↓↓←↑↑
boxban-medium 81	↑→→↑↑←←↓↑→→↓↓←↑←↑↑→↑→↓↓↑←←↓↓→↓→↓↓←↑←↑↑→↑→↓
boxban-medium 82	↑↑↑←↑←←↓↓↓↓→↑←↑↑→→↑→→→↑↑→→↑←→↓↓↓←←←←↑←←←↓↓↓↓→↑↑←↑→
boxban-medium 83	←←←↑→↑→→←←↓↓→→→↑→↑↑←↑←↓→→↓↓←↑↓↓←←←↑↑→→↑↑→↑↑←↓↓↓
boxban-medium 84	↑↑↑←↑↑→→→→↓←↑←←←↓↓→↑←↑←←↓→→↑→→→↓→→↑←←←←←
boxban-medium 85	←↓←←←↑←↓↓↓↓↑↑↑→→→→↑←→→↓↓↓←↓↓→↑↑↑→↑←←←↓←←↑←↓↓↓↑↑→→→→→↓↓↓↓↓↓←←↑→↓→↑↑↑↑↑→ ↑←←←
boxban-medium 86	↓↓↓→→→→→→↓↓←↓←↑→←↑←↑←←←←↓→←↑←←←↓↓→↑←↑→→←↓↓↓↓←↑↑
boxban-medium 87	↓←←←↓↓↓↓↓↓→↑←↑→→→↑↑↑→↑←←↑←←↓↓↓→↑→→←←↓→→→→→→←←←↑↑↓←←←↑↑↑→→↓→
boxban-medium 88	←←←↓↓→↑↓→→→↓↓↓←←←↑↑↓←↑↑↑↑→→→←←↑←←→↓↓↓↓→↑↑↓↓←→→→→↑↑→→↓
boxban-medium 89	→↓↓←←↑→↓→↓↓←↑→↓↓↓←↑→↑↑↑←↑↑→↓↓↓↓
boxban-medium 90	↑↑←←↓←←→→↑→→↓←←↓→↓→↑←↑←←↑←←←←←↓↓↓↓←↓↓→↑↑↑↑←↑→↓↓↓↓↓←↓↓→↑↑↑↑↑
boxban-medium 91	←→↑↑←↓←←↓↓→↑←↑→←↑↑→↓→→↑←→↓↓↓←↑
boxban-medium 92	→→→↑↑→→↓←→↓↓←↑↑↓←↑↓←↓↓↓↓←←←↑→→→→→↓→↑↑
boxban-medium 93	→↑→→↓↓↓↓←←←←↑↑←↑→↓↓↓←↑↑→↑↑→→↓←↑←↓↓←↓↓→→→→→↑↑↑↑←←↓←←↑→↓→→↑→↓↓
boxban-medium 94	↑←↓↓↑↑→↑↑←↓↑←←↓→→↓→↓↓←↑↓→↓↓↓←←↑→↑↓↓→↑↑↑↑←↑↑
boxban-medium 95	←←→→→↓↓↓↓↓↓→↓→→↑←←←←←→↓→
boxban-medium 96	↑→↓↓→↓↓←←←←←↓←←↑↑→↓→→→→→→↓↓↓←↑↑→↑←←←↓→↑←←←←←←↑←↓↓
boxban-medium 97	→→↓→↓→→↑↑↑↑←↑↓↓→↓↓↓←←↑→↓→↑↑↑←↓→↓↓←←↑→↓→↑↑
boxban-medium 98	↑←↓←←↑←↑←←↓↓↓↓↓↓→↑←↑↑↑↑↑→→↓←↑←↓↓↓↓↓↑↑→↑↑←↑→↓→→↓→→↑←
boxban-medium 99	↑→↓→↓→→↓↓←←↑→↑→↓←←↓←←↑→→→↓→→←↑←↑↑←←↓↓←↓→
boxban-medium 100	←←←←↑→↑←↑↑→↓↓↓→→↓→↑↑→↑←↓↓↓→↑↑→→↓←←

A.6 Boxoban (hard)

A.6.1 Unsolved

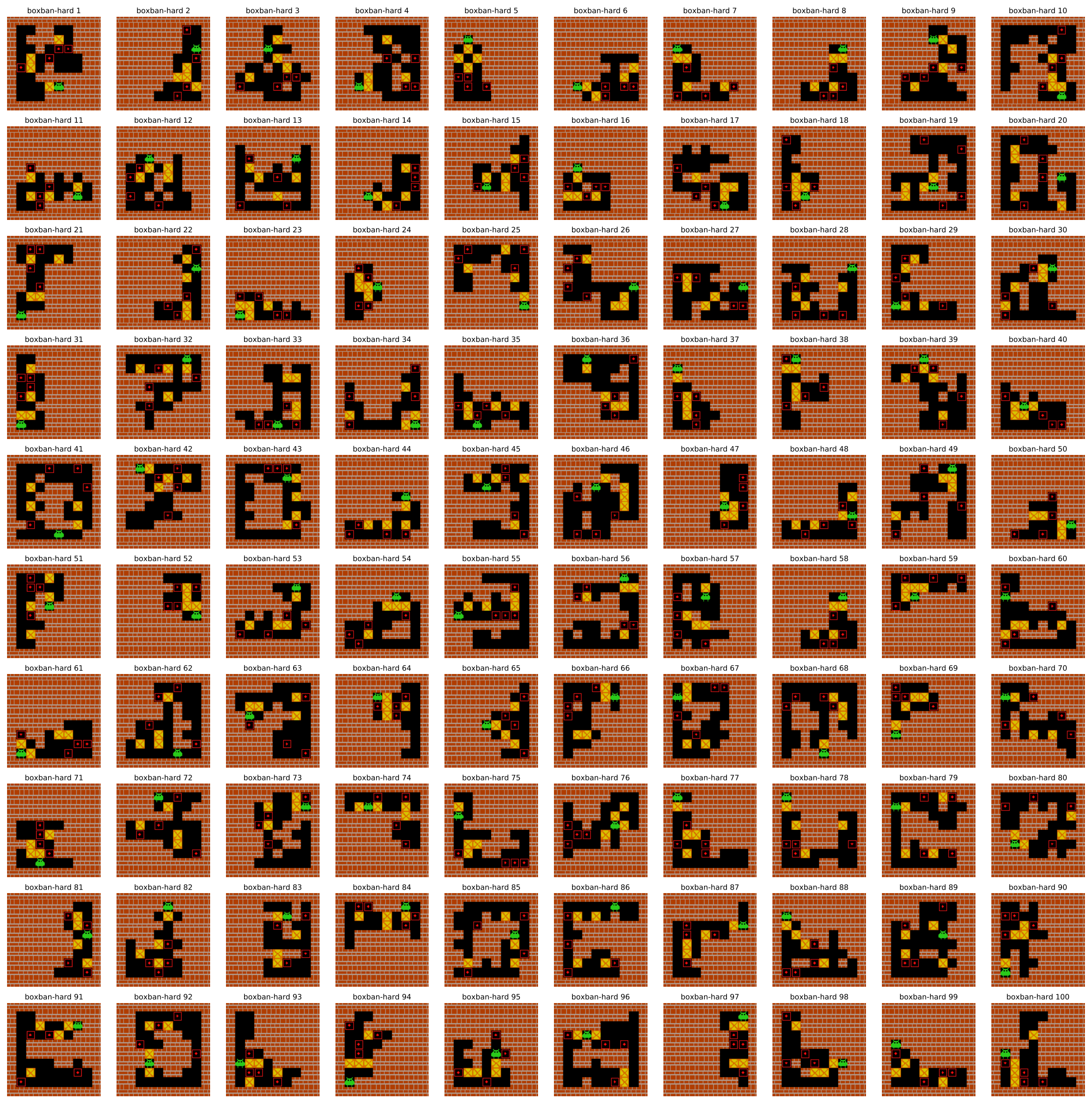


Figure 27: loss of the "10 $\times$ 10 grid with two boxes" model with a batch size of 1

A.6.2 Solutions

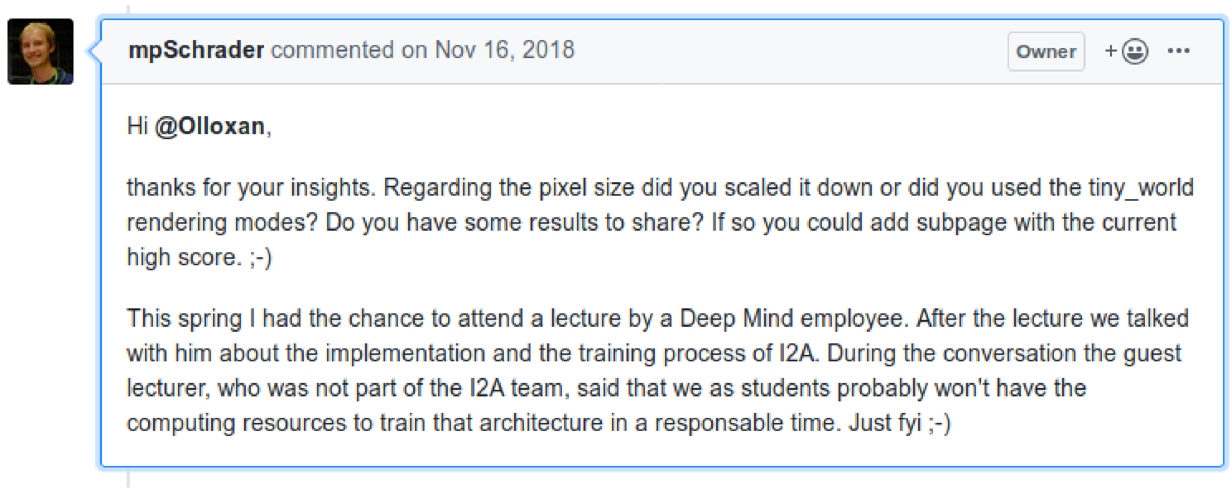
Table 14: Solutions for Boxoban (hard)

game name	DeepCubeA (10 × 10 with 2 boxes model)
boxban-hard 1	←←↓←←↑↑↑↑→↓↑→↑→↑→→↑↑←↓→↓↓←←←↓←←↑→→→→↓→↑←←←↓←←↓↓↓→→↑←↑↑←↑→→←↓↓→↓↓←↑↑
boxban-hard 2	↓←←↓↓↑↑→→↓↓←↑↓↓↓←←↑→↓→→↑↑↑←↑↑↑↓↓↓←↓↓
boxban-hard 3	→↓↓↓←←↑←↓→↓→↓→↑↓→→↑↑←←↑↑←↑↑↑→↓↓↓↓↓→→↑←↓←←↓↓→↑←↑→→←←↓←←↑→→→↑↑←↑↑→↓↓↓↓
boxban-hard 4	→→↑→↑↑↑←↑→→→↓→↓↓←↓→↓↓←←←↑↑↑↑↑←↑→→↑→→↓↓↓↓↓↑↑↑↑↑←←↓→↑→↓↓←↑←←↓↓↓↓↓←←←↑→→↓→↑↑
boxban-hard 5	↓→↓↓↓←↓↓→→→→↑←↑↑↑↑←←↓→↑→↓↓↓←↑↓←↓↓→↑→↑↑←↓↑→↑↑←↓←↓↑→↓
boxban-hard 6	→↓→→↑↑→↑↑→→↓↓←→↑↑←←↓↓→↓↓→↑↑←←←↓→←↓←←↑→↓→→↑↑→→↓↓←←↑↑↑↑→→↓↓
boxban-hard 7	↓↑→↓→↓↓←←↓↓→→→→↑→←↓←←←←↑↑→↓←↓→→←↑↑→↑↑←←↓↓↓
boxban-hard 8	↓↓↓→↓↓←←↑←↓←←↑→→↓→→→↑↑←↓→↓←←←↑←←↓→↑→→↓→→↑↑←↑←↓↓↑↑→↑↑←↓
boxban-hard 9	↑→↓→→↓↓←←↓↓←↓↓→→↑←↓←↑↑→↑↑→→↑↑←←↓↓↑→↑→↓↓←←↓↓←↓↓→→↑←←↓←↑←↓←↑
boxban-hard 10	↑↑↑↑↑↑↓↓↓↓↓↓←←↑→↓→↑↑↑←↓→↓←↑↑→↑↑←↓↑→→↑↑←↓↓↑↑←←←←↓←←↑→→→→→
boxban-hard 11	→↑←←←←←↓↓←←↑↑→↑→↓↓↑→→↓←↑←←←↓↓→↑←↑→→
boxban-hard 12	↓←↓↓←↓↓→→→→→↑↑↑↑←←←↑←↓←↓↓→→↑↓←←↓↓→→→→→↑↑↑↑←←←↓←↓↓←↓→→←↑↑↑→↑↑←↓←↓↓→→↑←↑→→→→↓↓←↑→↑←←←
boxban-hard 13	→↓↓←↓→↓↓←←←←←←←↑↑↑→→↓→→↓↑←←↑←←↓↓↓→→→→↑↑↑←↓→→→↑←↓←←↑←←↑←↓↓↓
boxban-hard 14	→→←↓→→↑↑→→↓↓←↑←←↓→↑→↑→↑↑↑←←↓→←↓↓→↓←↓←←↑→→↑→→↓↓←↑←↑→←↑→
boxban-hard 15	→↑→←↓←←↑→↓→→↑↑→→↑↑←↓→↓↓↓←←↑→↓→↑↑↓←↓←←←↑→→→
boxban-hard 16	↓→→↓→↓↓←↑↓←←←↑→→↓→→↑↑↑←←←↓↑→→→↓↓↓←←↑→↓→↑←↓←←←↑→
boxban-hard 17	←↑↑←←↑↓←←↑↑↑→→→↓←↓↓→→↓↓→→↑←→→↑↑←↓←←←←↑↑↑←←↓↓↓→←↑↑↑→→↓←→↓↓→→↓→↓→↑→↑←←←
boxban-hard 18	↑←↓←↑↑↑→↓←↓↓→→↑→↑←↓↓←←↑→→↑←↑←↑↑↓↓↓
boxban-hard 19	↑↑↑↓↓←←↓↓↓→→→→→↑↑↑↑↑←↑←↑←←↓→↓↓↓↓←↑←↓↑←←↓→↑→→→↑↑↑→→↓→↓↓↓↓↓←←←←←←↑↑←↑→→→↓→↑↑↑↓↓←←←↓↓↓→→→→→→↑↑↑↑↑←↑↑→↓↓←↑←↑←←↓→↓↓↓↓←←↓←↓→→→←←↑←↑←↑→→→↓→↑↑↑↑→↑←→↓→↓→↓↓↓
boxban-hard 20	→↓↓↓←←←←↑→↑↑↑↑↑↑↑←←←←↓→←↓↓→↑↑→→↑→↓←←←←↑→↓↓→↓↓←↓↓→→→↓→↑↑
boxban-hard 21	↑→↑↑→↑↑←↑←↑→↓→↑→↑→→↓←←←↓←↑↓→↓↓←↓↓→↑←↑↑
boxban-hard 22	←↓↓→↓↓↓←←↑←←↓→↑→→↓→↑↑↑←↓→↓←←→↑↑↑↑→↑↑←↓↓↓↓→↑↑↑
boxban-hard 23	↑↓→→↑↑←↓→↓→→↑←↓←↑←←↓→→→
boxban-hard 24	↓←↑↓↓←←↑↑→↑↓↓→→↑←↓←←↓→←↑↑↑→↓↓
boxban-hard 25	↑↑←↑↑→↑↑←←←←↓←←←↓↓→↑←↑↑→→→→→↓↓→→↑↑←←↓↓→↓↓→↑↑↑↓←←↑↑→↓↑←←←←↓←
boxban-hard 26	↓←→↓↓←↑↑→↑←←←←←↓←↑↑↓→→→↓↓→↓→↑↑→↑←←←←←↓←↑→↑↑←↑↑→↓↓↓↓→→↓↓→↑→↑←←←←←↑↑
boxban-hard 27	↓↓←↓←←←↑↑→→←↓↓→→→↑↑↑←↓←←↑←↑←←↓←↓→→↑↓→→→→↑→↓
boxban-hard 28	↓←↓↓↓→↓←←←←←↑←←↓→→→→→←←↑←←←↑↑→→↓↓←↓→→↑←↑↑←↑↑→↓↓↓↓→↓←↑←↑
boxban-hard 29	↑↑↑↑↑→↑→→↓←←↑←↓↓↓→↑↓←↓↓→→↓→→←↓←←↑↓←↑↑↑
boxban-hard 30	←↓↓←↓↓↓←←↑↑←↑↑→→↑→←↓←←↓↓→↓↓←↑↓→→→↑↑↑→↑→↑↑←↓←←↓←↓↓↓←↓→→→↑→→↓←←↑↑↑→↑→↑←↓↓↓←↓→↓←←←
boxban-hard 31	↑↑↑↓→↑→↑↑←↓↓↑↑←↑↑→↓↓→↓↓←↑↓←↑←↓←↓↓↓→→↑←↑↓↓←↑↑→↑
boxban-hard 32	←←←↓→↑←←←←↓→→↑→→→↓↓↑↑→→↓↓↓←←↓←→↑←←↓→↓←→↑→↑↑↑→↑→↓
boxban-hard 33	↑↑↑↑→→→↑↑←↓→↓↓↓↓↓←←←↑↑↑↑↑→↓←↓↓↓↓→→→↑↑↑↑←←↑←↓↓↓↓↑↑↑→→→↑↑←↓→→↓↓↓↓↓←←←←→→→→↑↑↑↑←←↑←↓↓↓↓↑↑↑→→→↓↓↓↓←←←→→→↑↑←↓
boxban-hard 34	↑↑←←↓↓←←←←←↑↓→→→→→↑↑→→↑↑↑←↓↓→↓↓↓←←↑↑→←↓↓→→↑↑↓↓←←←←←←↑←↑
boxban-hard 35	→↑→→↑←←↓←↓←←↑→→→←←←↑↑→↓←↓→↓→→↑↑←←→→→→↓→→↑←←←←↓→←↓←←↑↑→
boxban-hard 36	←↓→↓→↓→→↓↓←↑→↑→→↓↓←→↑↑←←↓←↓↓→↑→→↓←↑←←↑↑←↑↑→↓↓↓↑↑↑→←↑←←↑↑→→→→→
boxban-hard 37	↓↓↓↓→←↑↑→↓→↓→↓↓←↑↑↓↓←←↑→←↑→↑↑←↓↓→↓↓→→↑↑←→↓↓←←↑↑←↑↑→↓↓→→↓↓←←←↑↑→↓←↓→
boxban-hard 38	↓↑←↓↓↓↓→→↑→→↓←↑←←↓←↑↑↑→↓←↓→→→
boxban-hard 39	↓↓↑→↓↓↓↓↓←↓→→↑←↑↑↑↑←←↑←←↓→→→↑→↓↓↓↓→↓↓←↑↑↑←↑↑→↑←↓↓↓→↑↑↓↓↓→→↓→↓←→↑↑
boxban-hard 40	→↓↓←←←↑↑→↓→→↑←↓←←↑↑→↓←↓↓→→→→→↑↑←←←←↓←↑↓↓→→→→
boxban-hard 41	←←←↑↑←↑↑↑↑↑↑→→→↓→→→↓↓→↓↓←←↑→↓→↑↑←↑↑←←←↑←←↓←↓↓↓↓→↓→↓→→→→↑↑↑↑↑↑←←←↑←←←↓↓↓→↓↓↑↑↑↑←↑→→
boxban-hard 42	→→↓↓↓↓←↓→↑↑↑↑↑→→↓↓→→↑↑←↓→↓←↑←←
boxban-hard 43	↓↓↓→↓←↑↑↑→↓↑←↑↑→↓←←←↑←←←↓↓↓↓↓↓↓→→→→→↑↑↑↑↑↑→↑←←↓→↓↓↓↓→→↑←↓←↑↑↑↑→↑←→↓↓↓←↓↓→→↑←↓←↑↑↑↑
boxban-hard 44	↓→↓↓←→↑↑←↓←↓↑→→↓←↓←←←↑←→↓→→→↑↑←↓←↓←←←→↑→→→↑→→↓←↓←←↑←←↓→↑→→↑→↓
boxban-hard 45	↑→↑→→↓↓→↓↓↓↓↓←↑→↑↑↑↑←↑↑←←↓→←↓←←←↑→→→↑→→→↓↓←↑→↑←→↓↓↓↓↓↓
boxban-hard 46	↓↓↓→→→→↑←→↑←←←→→→↑↑←↓↑↑←←↓↓↓→↓↓←←←→↓↓↓←←↑↑↑→→→↑↑←↓↓↓↓→↓←←→↑→↑↑→→→↑←←←↑

	←↓↓↓↓
boxban-hard 47	↑→→↓↓↓←←↑→↓→↑↑↑↑↓←←↓→↓↓←←↑→↓→↑→↑↑←↑↑←↓↓↓→←↓
boxban-hard 48	↓↓←←←←←↑→→↓→→→↑↑↑←↑↑→↓↓←↓↓→↓←←←↑→←←←↓←←↑→→→→↓←→→→↑←→↑↑→↑↑←↓→↓↓
boxban-hard 49	→↓↓←←←→→→↑↑←←↓←↓→→←←↑←←↓→↓↓←←←↓↓↑→↑→→↑↑↑→→↓←↑←↓→↑→↑→→↓←←←↓←↑→→→→↓↓
boxban-hard 50	←←↑↑←←↓↓→↓←←→→→↑→→↓←←↑↑↑←←↓→
boxban-hard 51	↑→↑←←↑←←↓↓→↓↑←↑↑→↓→→→↑←←↓←←↓↓→↑←↑→↓↓←↓↓→↑↑←↓↓↓↓→↑↑←↑↑→↑
boxban-hard 52	↑←←↑↑↑→→↓↓←→↑↑←←↓↓→→↑←↓↓↓→↑↑
boxban-hard 53	→↓↓←↓↓↓←←←←←←↑→↓→→→→→↑↑↑→↑↑←↓↓↓←↓→↓←←←↑→←↓←←↑→→↓→→→↑↑←↓→↓←←←↑→↓→↑←←←←↓→
boxban-hard 54	→↓→↓←←→↑↑←↓→↓→↓↓↓←←←←↑↑↑↑→→←←↓↓→↑←↓↓↓→→→→↑↑↑←←↑↑→↓→↓↓↓↓←←←←↑↑↑↑→→↓←↑←↓→↓←↓↓→→→→↑↑↑←←←↑←↓↓↓←←→→↑↑→↑→↑→↓→↓↓↓↓←←←←←
boxban-hard 55	→→→→→→→↑←↑↓↓←←↑→↓←←←↑→→↓→→→↑↑←↓→↓←←←↑→←←←↓←←↑→→→→↓←←→→→→→↑↑←↓→↓←←←↑→↓→→↑↑←↓←←←←↓→→↑←←←←↓→→→
boxban-hard 56	→↓←↓↓→↓↓↓↓←↑→↑↑↑←↑↑→↓↓↓↑↑↑←←←←↓←
boxban-hard 57	←↑↑→↓↓→↓←←←↓↓←↓→↑↑←↑→→→↓↓←←↓←↑→→→↓←←
boxban-hard 58	↓↓↓→↓↓←←↑←↓←←↑→→↓→→→↑↑←↓→↓←←←↑←←↓→↑→→↓→→↑↑←↑←↓↓↑↑→↑↑←↓
boxban-hard 59	←↑←↓↑↑→→→←←↓↓→↑→↑→→→←↓←←←↓←←↑→↑→→↓←↓←↑
boxban-hard 60	↓→↓→→→→→↓→↓↓←←←←←←↑↑↓↓←↑↑↑↑↑↓↓→→→→←↓←↑←↓←↓↓→→→→→→→↑↑←←↑→↓↓→↓←←←←←←
boxban-hard 61	→→→→→↑→↑↑←↓→↓←↓←←↑↑→→→↓→↓←↑↑↑→↓←←←←↓→→←↓←←←←↑
boxban-hard 62	→→↑↑↑↑↑↑↑←←←←↓→→→←←↓↓↓←←↓↓→↓→↑↑↑↑↑↑←↑→↓↓↓↓↓←↓←↓←←↑→→↑↓→↓→↑↑↑↑↑↓↓↓→↓↓→→→↑↑↑↑↑←↑←←→→↑→↓↓↓↓↓
boxban-hard 63	↑←↑→→→↓←↑←↓↓↑→→↑→↑→→↓↑←←↓→→↑→↓↓↓↑↑←←↓→↓↓↓↑↑→↑↓↓↓↓←
boxban-hard 64	→→↓↓→→↑←→↑↑←↓←←←↓↓→→←←↑↑→→↓←↑→→→↓←→↓↓←↑↑
boxban-hard 65	→→→↑→↑↑←↓→↓←↓↓→↑↓↓↓←↑↑↑←←↓→→↑→↑↑←↓→↓↓↓
boxban-hard 66	←←←↓↓←↓←↓↓→↑↑→↑↑↑←↑←↓↓↓↓→↓↓→↑↑↑↑↓↓←←↑↑↑↑→→→↓→→↑←↓←←↓↓←↓←↑→↓↓↓←↑↑
boxban-hard 67	→→→→←←←←↑→→→←←←↓↓↓↓↓→→→↑→↑→↑↑↓↓←↓↓↓←↓←↑←←↓←↑↑↑
boxban-hard 68	↑↑↑↑↑←↑↑→→→→↓↓←→↓↓←↑↑→↑↑←←←←↓↓→→←←↑↑→→↓←←←←→→↑→→→→↓←←←←→→→→↓↓←↑→↑←←↑←←↓←↑←←↓
boxban-hard 69	↑↑↑↑→↑→→↓←↓→←←↓←↑
boxban-hard 70	→→←←↓↓↓→→→→→↑→↓→↓↓←↑↑←←↓→↑→→↓↓←↑→↑←←←←←↓→→→→←↑←←←←
boxban-hard 71	↑↓←←↑→↑↓↓→→↑←→↑←↑↑←←↓→↑→→↓↓←↓↓←←↑→↑↑←↑→↓↓↓↓→↑
boxban-hard 72	→→↓↓↓←←←↓←←↑→→↓→→↓→→↑←←←←→→→↑↑↑↑←←↓→↑→↓↓↓→↓→↓←←↑↑↑↑↑↑
boxban-hard 73	←←↓←←↓↑→→↑↑→←←↓→↓←↓←←↑↑→↓→→↑↑←↓→↓←↓←↓
boxban-hard 74	↑→→→→→↓←↓↓→↓↓←↑→↑←↑↑↑←←↓→↑←←←↓→→↑→→↓↓↓→↓↓←↑↑↑↑→↑←←←↓→←←←↑←←↓→→→→↑←←→→→→↓↓↓→↓↓←↑↑↑↑→↑←←←↓→↑→↓↓↓↓↑↑↑←←←←↑→→→←←←↓←←↑→→
boxban-hard 75	↓↓→→↓↓↑↑←←↓↓→↓→→→→→←←←←↑←←↑↑→→↓↓←↓→→→→←←←←↑←↑↑↑↑→↑↑←↓↓↓↓↓→↑→↓↓←↓→→→
boxban-hard 76	↓←←←↓←←↑↑→↓←↓→↑→→→↑↑↑→↑→↓→↓↓←←↑←↓→↓←←←←↓←↑→→→→→↑→→↑←←→↑↑←↓←↓↓→↓←←←→→→↑→↑←↑←↓↓→↓←←↓←←
boxban-hard 77	↓↓→↓↓←↓↓→→←←↑↑→↑↑←↓↓→↓→→↑←↓↓↓←←↑→↓→↑←↑↑↓→
boxban-hard 78	↓↓→↓↓←↓↓↓→→→↑→→→→↑↑↑↑←↓↓→↓↓←←←←↓←←↑→→→→→↓→↑↑↑←↓→↓←↑↑↑↑→↓↓←↓↓←←←←↓←←↑↑↑→↑↑←↓↓
boxban-hard 79	→→↑→→↓↓→→↑↑←←←←↓→↑→→→↓↓←←↑←↑→→←←←↓←←↑→→←←↓↓↓↓↓↓→↓→→↑→→←↓←←←↑→
boxban-hard 80	↑↑←↑↑↑→→→→→→↓↓↓←↓↓→↑↑↑↑→↑←←←←←↓←←↓↓→↑←↑↑→→↓←↓↓↓↓→→→↑→→↑→↑
boxban-hard 81	↑↑←→↓↓←↑↓→↓↓←↓↓→↑←↑↑→↑↑←↓↓↓↓←↓→↑→↑↑←↑↑→↑↑←↓↓↓↓↓→↓↓↓←→↑↑↑
boxban-hard 82	↓↓↓↓↓↓←←↑→→↑↑←↓↓→↓←↓←←←↑↑→→→↓→→↓←←↑↑↑→↓←←←←↓↓→↑←↑→→→↓←→↓→→↑←↑←←↓→←↑←←↓→↓→
boxban-hard 83	↑←↓↓←↓↓→→→→↓↓←↓←←←↑→→→↓→↑↑↑←←←↓↑↑↓→→→↑↑←↓→↓↓↓←←↓←←↑→↑↑→→↑→↓
boxban-hard 84	→↓↓←←↑→←←←↓←↑↓←↑→→↓→→
boxban-hard 85	→↓↓←←↓↓→↑↓←←←←←↑↑↑↑↑←↑→↑→↓↑→→→→↓→↓↓↓↓←→↑↑←↓←↓↓→↓←←←←→→→↑↑→↑→↑↑↑↑←←←↓→→
boxban-hard 86	←←←←←↓↓↓↓↓↓→↓→→↑↑→↓→↓←←←←↑→→→←←←←↑↑↑↑↑↑→→↓←↑←↓↓↓↑↑↑→→→→↓←←←
boxban-hard 87	←←←↓←←↑←←↓↓→↓←↑↑↑→↓→→→↑←←↓←←↓↓→→↑↓←←↑↑→→→←↓↓←↑
boxban-hard 88	↓→↓↓←↓↓↓→→→↑↑↑←→↓↓↓←←←↑↑→→↑→↓←←←↑→→←↑↑←↓↓↓↓↑→→→→↓→↓←←←
boxban-hard 89	←←←↓←↑↓↓→↓↓→→→↑↑↑↑↑↑←↓↓→↓←←↑←↓↓
boxban-hard 90	↑↑↑→↑↑←↑↑→↓→→↓↓←↓←←↑↑↓↓→→↑←↑→
boxban-hard 91	←↓←←←←↑←↓→→→↑←↓→→→↑←←↓←←←↑↑→↓←↓→→→↑→→↓←↑←←↓←←↑↑→↓←↓↓↓↓↓→→→→→
boxban-hard 92	↑↑←↑↑↑→→↓→←↑←←↓→←↓↓→↑↑←↑→→→↓←←↑←↓↓↑→→→→→↓→↓↓↓↓←↓←←←←↑↑↑↑↑←↑→↑→→↓→→↑→↓↓↓
boxban-hard 93	↑→→↓↑←←↓↓→↑→→↓←↓→→→↑←←↑←←←↓↓→→→
boxban-hard 94	↑→↑↑→↑→↑←↓↓←←↑→←↑↑→↓←↓↓→↓↓←↑↑↑↑→↓↑→→↓←↑←←↓↓↓↓→→↑↑↑↓↓←←↑↑↑→→

boxban-hard 95	←↓↓↓→→↑←↑←↑→↓↓↓←↑↓←←←↑↑→↓←↓→↑→→↓→↑↑↑↑↓←↓→↓←←←↓→
boxban-hard 96	→→↑→↓→↓←→→↓↓↓←←↓←←↑←←←↑↑↑↑→→→→→↑→↓→↓↓↓↓↓←↑←←←←←↓←↑↑↑↑
boxban-hard 97	↓↓←↓↓→↓↑←↑↑→↑↑←↓→↓↓←↓←↓↓→↑↑↓→↑
boxban-hard 98	←↓↓←←↑↑↓↓→→↑↑←↑←←←↑←↑↑↑→↓←↓↓→↓→→→↓→↓↓←↑←↑↓↓←←↑←↑↑↑↓→→↓→↓→→↑←↑←←←↑←↑↑→↓ ↓←↓→
boxban-hard 99	↓→↓→↓↓←←↑↑→→↓→↓→→↑→→↓←←←←↑→→←←←↑←←↓→→←↓→↑→→
boxban-hard 100	↓→←↓↓→↑→→↓←↑→↑↑↑↑←↑↑→↓↓↓↓↓↓←↓→↑↑↑←↓←↓←↓→

B Github Post


mpSchrader commented on Nov 16, 2018

Hi **@Olloxan**,

thanks for your insights. Regarding the pixel size did you scaled it down or did you used the tiny_world rendering modes? Do you have some results to share? If so you could add subpage with the current high score. ;-)

This spring I had the chance to attend a lecture by a Deep Mind employee. After the lecture we talked with him about the implementation and the training process of I2A. During the conversation the guest lecturer, who was not part of the I2A team, said that we as students probably won't have the computing resources to train that architecture in a responsable time. Just fyi ;-)

Figure 28: quote from `https://github.com/mpSchrader/gym-sokoban/issues/11#issuecomment-439464194`

C Email correspondence

Dear Mr. Hougen,

I am a student in computer science at Heidelberg University (Germany), and as part of my final project for the lecture Advanced Machine Learning with Prof. Ulrich Köthe, I will be implementing the SSRL algorithm to learn the game 'Sokoban'.

I have a question regarding the neural network architecture chosen for this algorithm and, relatedly, a question about the formula used for eligibility traces.

1) In the paper, you chose to use a single layer architecture. Is it possible to expand this to multiple layers?

2) The formula for the eligibility trace e_u_i,j(k) = x_i(k)*(w_i,j(k) - u_i,j) is chosen to scale with the value of the input to the network x_i. This makes sense intuitively: An input of zero has an equal (zero) effect on all output nodes of the network and therefore shouldn't take credit for the action suggested by the network.

However, in the design of your experiment, all inputs are in {0, 1}. If negative values were permitted as inputs, the eligibility traces would depend on the sign of the input.

My understanding is that the factor x_i(k) is meant to scale with the influence an input node had on the output, and because of this, if we allow negative inputs, it should read |x_i(k)|*(...) in the formula (analogously for the std. eligibility traces).

This becomes relevant in the multi-layer case: Even if all inputs are positive, outputs of nodes in hidden layers may be negative.

Thank you for your time,

Jakob Weichselbaumer

Figure 29: Initial inquiry w. Prof. Hougen

Jakob,

Thanks for your questions regarding our paper. My apologies for the
slow reply. Your email was buried under the start-of-semester email
avalanche.

1) Yes, it is possible to use SSRL with a multi-layer architecture.
We used just one layer in the original work because that was all
that was needed for that problem. However, we have papers under
review in which we took the changes made in that weight layer and
backpropagated them to a previous layer. In theory, there should be
no reason such changes couldn't be backpropagated through arbitrary
prior layers using any arbitrary gradient-adjustment algorithm.

2) Yes, your intuitive understanding of the eligibility trace is
correct -- we use the term x_i(k) to scale the change based on the
degree to which that input might have influenced the output. In our
paper, we only considered inputs of 0 or 1 and wrote Equation 4 for
values in the range 0 to 1. However, if the input values can take on
negative values, we'd need to take the absolute value of x in
Equations 4 and 7.

If you have more questions, please don't hesitate to ask. Good luck
on your projects!

Prof Hougen

PS. I've added Dr Shah to the email, in case he would like to add
any commentary.

Figure 30: Answer regarding activations and multi-layer generalization

Dear Mr. Hougen,

thank you for your reply, it was not too late for my purposes.

I have a follow-up question regarding your response to my first question, as it relates
to a multi-layer architecture:

When you speak of "backpropagating the changes made in that layer", are you suggesting
that the eligibility traces should be recursively multiplied together to account for the
cumulative effect of that weight's deviation from its mean (w_i,j(k) - u_i,j) on the
output of the network?

Or should eligibility traces depend only on how each weight amplifies its immediately
preceding input?

Finally, may I quote this email dialog as part of the final report for the project? I can
send you a copy once it is finished.

Jakob

Figure 31: Follow-up question regarding backpropagation

```
           Jakob,

Regarding backpropagation in multi-layer networks, the idea is that first we
make changes to mu values in the final layer of the network exactly as shown
in the paper, then we need to figure out how to change weights in the previous
layer(s) of the network using backprop. Here we note that changes made to mu
in Equation 3 are similar in concept to changes made to weights in the final
layer of a feedforward ANN using supervised learning. So, if the weights in the
final layer are w_ij and the weights in the penultimate layer are v_hi, the usual
backpropagation rule (not including momentum) for updating the v_hi weights is

        v_hi(tau + 1) = v_hi(tau) + delta v_hi(tau)

where delta v_hi(tau) is

        delta v_hi(tau) = learning rate * derivative of activation function
in penultimate layer * input of previous layer * summation over J of (error *
derivative of activation function in output layer * w_ij).

However, because we're not doing supervised learning, we don't have an error value.
So, instead, we take the summation over J of the changes to mu_ij (which are after
the + sign in Equation 3).

Feel free to add this dialog to your report. Please do send me a copy when it is
completed.

Prof Hougen
```

Figure 32: Response regarding backpropagation

References

[1] Joseph Culberson. Sokoban is PSPACE-complete, 1997.

[2] Max-Philipp B. Schrader. gym-sokoban. `https://github.com/mpSchrader/gym-sokoban`, 2018.

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

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

[5] Stefan Behnel, Robert Bradshaw, Craig Citro, Lisandro Dalcin, Dag Sverre Seljebotn, and Kurt Smith. Cython: The best of both worlds. *Computing in Science & Engineering*, 13(2):31, 2011.

[6] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep Residual Learning for Image Recognition. *arXiv:1512.03385 [cs]*, December 2015. arXiv: 1512.03385.

[7] Stuart J. Russell, Peter Norvig, and Ernest Davis. *Artificial intelligence: a modern approach*. Prentice Hall series in artificial intelligence. Prentice Hall, Upper Saddle River, 3rd ed edition, 2010.

[8] Théophane Weber, Sébastien Racanière, David P. Reichert, Lars Buesing, Arthur Guez, Danilo Jimenez Rezende, Adria Puigdomènech Badia, Oriol Vinyals, Nicolas Heess, Yujia Li, Razvan Pascanu, Peter Battaglia, Demis Hassabis, David Silver, and Daan Wierstra. Imagination-Augmented Agents for Deep Reinforcement Learning. *arXiv:1707.06203 [cs, stat]*, July 2017. arXiv: 1707.06203.

[9] Richard S. Sutton and Andrew G. Barto. *Reinforcement learning: an introduction*. Adaptive computation and machine learning series. The MIT Press, Cambridge, Massachusetts, second edition edition, 2018.

[10] Volodymyr Mnih, Adrià Puigdomènech Badia, Mehdi Mirza, Alex Graves, Timothy P. Lillicrap, Tim Harley, David Silver, and Koray Kavukcuoglu. Asynchronous methods for deep reinforcement learning, 2016.

[11] S. N. H. Shah and D. F. Hougen. Stochastic synapse reinforcement learning (ssrl). In *2017 IEEE Symposium Series on Computational Intelligence (SSCI)*, pages 1–8, Nov 2017.

[12] Jonathan Frankle and Michael Carbin. The lottery ticket hypothesis: Training pruned neural networks. *CoRR*, abs/1803.03635, 2018.