Master's Thesis

Policy-guided planning

Antoine Schmidt

Examiner: Prof. Dr. Nebel
Advisers: Dr. Mattmüller

Albert-Ludwigs-University Freiburg

Faculty of Engineering

Department of Computer Science

Foundations of Artificial Intelligence

August 04$^{\text{th}}$, 2020

Writing Period

04. 02. 2020 – 04. 08. 2020

Examiner

Prof. Dr. Nebel

Second Examiner

Prof. Dr. Scholl

Advisers

Dr. Mattmüller

Declaration

I hereby declare, that I am the sole author and composer of my thesis and that no other sources or learning aids, other than those listed, have been used. Furthermore, I declare that I have acknowledged the work of others by providing detailed references of said work.
I also hereby declare that my thesis has not been prepared for another examination or assignment, either in its entirety or excerpts thereof.

Place, date	Signature

Abstract

The goal of this Master thesis is to incorporate domain-wide learning into classical planning.
For that, different search strategies such as A* and MCTS are compared and possible modifications to improve their performance on classical planning tasks are presented.
In the second part, policy and heuristic networks are trained, exploring different learning strategies such as imitation learning and reinforcement learning.
Finally, the trained networks and search algorithm are combined, presenting a novel planning algorithm.

Contents

List of figures

List of tables

List of algorithms

1 Introduction

Current state-of-the-art planners, like the Fast Forward [3] planner, use domain unspecific heuristics like a relaxation based heuristic to guide their search.
While this approach generally shows good results, in some domains such general heuristic can mislead the planner, returning bad plans. Additionally, when planners are applied to multiple problems from the same domain, the acquired knowledge from previous solutions are not used, leading to a complete restart of building up the planning graph and extracting a heuristic.
By incorporating machine learning into planning, a domain specific heuristic and policy can be learned, transferring acquired knowledge to new problems from the same domain, improving future planning performance.

This work focuses on training a policy and heuristic network for the Sokoban game, which is a problem from the classic planning domain. The networks are trained using different approaches, ranging from imitation learning to reinforcement learning. By then combining the trained networks with search algorithms, a new planning algorithm is created.

Related work

As this approach is not new, different publications exist on this topic.

The paper "Learning Generalized Reactive Policies using Deep Neural Networks" [2] trains a combined policy and heuristic network with imitation learning. Using the Sokoban problem domain, they train the network on Sokoban state images with problem solutions generated by the Fast Forward planner.
Evaluating the network greedily (taking the best predicted action), they are able to demonstrate a 97% success rate for Sokoban problems with one box and a 87% success rate in two box problems using 9×9 game sizes.

Another, more general approach is introduced in "Action Schema Networks: Generalised Policies with Deep Learning" [19]. The paper presents a novel method to combine machine learning with planning. Using a dynamic neuronal network, they are able to learn a domain-wide policy directly using state variables as input.

Finally, the publications from Google Deepmind, AlphaGo [16] and AlphaGo Zero [17] are also of interest for this work. As Go and other games can be seen as non-deterministic planning problems, the approaches presented in these papers can be applied to the general planning domain.

These four papers and their solutions to the arising problems when machine learning is applied to the planning domain are described in detail in the course of this thesis.

2 Sokoban

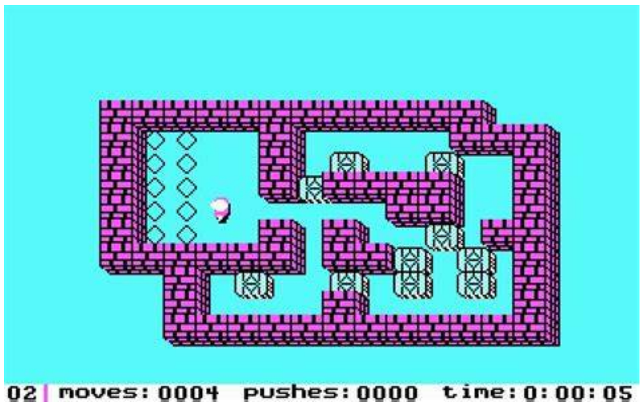


Figure 1: [Source] Sokoban for DOS

The Sokoban game (shown in figure 1, level 2 of the DOS version from 1988) was invented by Hiroyuki Imabayashi and came out for different computer-systems in 1982. It originally consisted of 20 levels, with each level containing multiple boxes which have to be pushed onto goal positions. The player can only push one box at a time and has no pull action.

2.1 Description

Sokoban is a transportation problem taking place in a grid-maze. This maze is build-up by placing walls, k boxes, k goals and the player. A particular level is solved when all boxes are on a goal position (in the original version, every box goes with every goal location). This is achieved by navigating the player in the four cardinal directions. When a box is at the players target position the box is moved into the corresponding direction when the boxes target position is empty. As the player has no pull action, a box movement can not be undone directly, creating unrecoverable dead-ends.

The game is a very interesting problem domain for planning as it easily covers all possible difficulty levels. Using only one box, turns the problem into a navigation task comparable to the Wumpus planning domain, while adding more boxes and optionally increasing the maze size can turn the game into an incredibly difficult problem.

Sokoban is a problem from the classical planning domain, as it fulfills all assumptions made in this domain:

finite, fully observable, deterministic, static, attainment goals,
sequential plans, implicit time and off-line planning.

Two important assumptions exploited in this work are:

- Fully observable: the observed state is identical to the "internal" state
- Deterministic: each action has only one outcome

2.2 Gym Sokoban

The Gym Sokoban project [12] implements the Sokoban game using the Gym interface.[1] As machine learning needs a lot of data, the project is able to randomly generate new Sokoban problems.

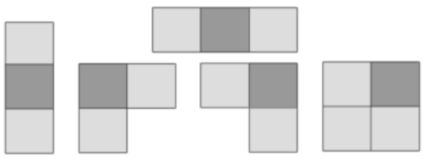


Figure 2: [Source] Sokoban room generation masks

New game instances are generated by first creating a room of a given size, filled with walls. Then, the algorithm iterates over the room, randomly switching direction with a probability of 35%. At every so called generation step, a random mask from figure 2 is selected and centered at the current position, changing the underlying room positions into free positions.
Following, the boxes are placed onto free positions, additionally marking them as goals. Finally, after placing the player, the game is randomly played in reverse (using depth-first search) moving the player and letting him pull boxes. After 300 reverse actions, the created problem is accepted when all boxes are off-goal.

2.2.1 Generating training problems

For all later policy and heuristic learning, a train set (with 9k problems) and a validation set (containing 1k problems) is created using 50 generation steps and uniformly sampling from the following parameters:

- problem width/height: 7 - 13
- number of boxes: 1 - 3

[1]Gym is a library from OpenAI, which defines a unified interface for environment interaction, commonly used in reinforcement learning.

Additionally, a final test set with 2k problems is generated. This set contains 1k problems with the same parameters as the train and validation problems and 1k instances with an increased size, ranging from 13 to 19 for both width and height using 100 generation steps. Those problems are later used to test the generalization performance of the trained networks on bigger instances.

To make the most out of the generated training instances, additional data augmentation is used. Given a Sokoban problem and a valid solution, the room can be augmented using rotation and/or mirroring. When the augmentation is equivalently applied to the solution, its properties are preserved (validity and possibly optimality). Additionally, a smaller problem can be randomly placed into a bigger room size (filled with walls), without changing the problem itself.

Analyzing the generated problems

Figure 3 plots the train and validation set by box number and Fast Forward (FF) solution length, showing a strong correlation between both values. The dotted vertical lines mark the average solution length with a length of ≈ 10 for one-box problems, ≈ 22 for two-box problems and ≈ 37 for three-box problems. Furthermore, a growing standard deviation with each additional box can be observed. With a growing long tail, a three-box problem with a 228 step FF solution exists.

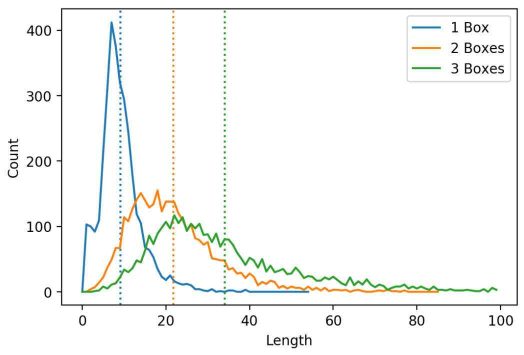
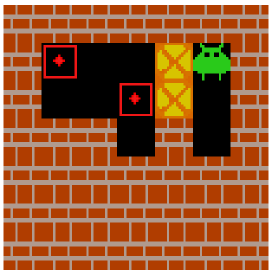


Figure 3: Sokoban FF solution length to box count

2.2.2 Pre- and post-processing


[Source] Sokoban state

0	0	0	0	0	0	0
0	1	1	1	1	1	0
0	1	1	1	1	1	0
0	0	0	1	0	1	0
0	0	0	0	0	0	0
0	0	0	0	0	0	0
0	0	0	0	0	0	0

isFree

0	0	0	0	0	0	0
0	0	0	0	1	0	0
0	0	0	0	1	0	0
0	0	0	0	0	0	0
0	0	0	0	0	0	0
0	0	0	0	0	0	0
0	0	0	0	0	0	0

isBox

0	0	0	0	0	0	0
0	1	0	0	0	0	0
0	0	0	1	0	0	0
0	0	0	0	0	0	0
0	0	0	0	0	0	0
0	0	0	0	0	0	0
0	0	0	0	0	0	0

isGoal

0	0	0	0	0	0	0
0	0	0	0	0	1	0
0	0	0	0	0	0	0
0	0	0	0	0	0	0
0	0	0	0	0	0	0
0	0	0	0	0	0	0
0	0	0	0	0	0	0

isPlayer

Figure 4: Atomic decomposition

Out of the box, the Gym Sokoban project returns an image encoded representation of the current state (as shown on top, in figure 4). As not every problem domain can be represented visually, the state is pre-processed to a more general representation. For that, the state gets decomposed into four atomic propositions *isFree*, *isBox*, *isGoal* and *isPlayer*. This leads to a $X \times Y \times 4$ state encoding, shown in figure 4. These atomic propositions simulate the possible internal representation of state variables in a planner.

As the walls are encoded implicitly (the wall positions are all zeros), the state representation harmonizes well with the later used zero padding in the convolutional network layers.

Furthermore, the actions are post-processed. Gym Sokoban originally has 4 move actions (only moving the player), 4 push actions (which optionally push a box) and a "do nothing" action. As a policy for the classic single player Sokoban is learned, the "do nothing" action is dropped. The simple move actions are also removed, allowing the policy to learn an undifferentiated action mapping. Finally, the 4 remaining actions are remapped to a clock-wise ordering, allowing for an easier state augmentation.

3 Search algorithms

Every planning algorithm has a search algorithm at its core. Below, the most common algorithms and possible modifications for classical planning are presented and evaluated on the Sokoban problem domain.

3.1 A* search

A* [11] is one of the most common search algorithm. It was invented by Peter Hart, Nils Nilsson and Bertram Raphael in 1968 with the goal to guide a mobile robot in the Shakey project.
Algorithm 1 shows the pseudo code of the A* search algorithm. Starting by adding the initial state to the search queue, the algorithm iteratively takes the state minimizing equation (1) out of the queue.

$$f(s) = g(s) + w \cdot h(s) \tag{1}$$

Where $g(s)$ is the path cost from the initial state to the node and $h(s)$ is the expected cost from the node to a goal state. With an additional variable w the heuristic $h(s)$ can be weighted, then called weighted A* search.
The algorithm then expands the selected state adding all its child states, when not in memory, to the queue. Having encountered a new child s', the state with its *path*, $g(s')$ and $h(s')$ value is saved to memory.

Algorithm 1 A* search

```
queue ← initialize with initial state
memory ← initialize empty state memory
while queue do
    state s ← queue.pop($\min_s f(s) = g(s) + w \cdot h(s)$)
    if $s$ is goal state then
        return saved path to $s$
    end if
    foreach $s' \in$ child-states of $s$ do                ▷ Expand state
        if $s' \in$ memory then
            improve path and $g(s')$ of $s'$ in memory if possible
        else
            memory ← memory $\cup$ $(s', path, g(s'), h(s'))$
            queue ← queue $\cup$ $s'$
        end if
    end for
end while
return unsolvable
```

In the classic algorithm, child nodes which are already in memory but show a lower $g(s')$ value, as child from the current state, are re-added to the queue.[1] This "re-opening" of an already seen child-state can be omitted, by dynamically improving the *path* and $g(s)$ value of a state s and all its successors retrospectively. This is possible as this work implements A* with a dynamic queue, only holding state identifiers, looking up the corresponding $g(s)$ and $h(s)$ values in memory.
With the additional use of a uniform cost function (action cost of 1) the $g(s)$ value can be simply updated to the corresponding states path length. In detail, when a state is re-encountered, showing a shorter path, all states in memory starting with the old longer path are update to the shorter path, additionally re-calculating the $g(s)$ value as the new path length.

When A* is used with a consistent heuristic, the solution optimality is given. A heuristic is consistent when:

$$h(s) \leq cost(s, s') + h(s') \; \forall s \text{ and } h(s_{\text{goal}}) = 0 \tag{2}$$

[1]This guarantees solution optimality for admissible but not consistent heuristics

A consistent heuristic implies admissibility (the heuristic never overestimates the cost). The simplest, consistent heuristic is $h(s) = 0 \; \forall s$ which turns A* into the Dijkstra algorithm.

Using weighted A* with $w > 1$ leads to a more greedy algorithm finishing faster but loosing the possible solution optimality property. With $w \to \infty$ the algorithm simulates a greedy best-first search.

Using A* with a uniform cost function and no heuristic, turns it into the breadth-first search (BFS) algorithm. BFS explores all states of a given depth, before continuing the search on deeper nodes. With depth as the nodes action-distance to the initial state.

In the following work, the A* search algorithm is only used with a uniform cost function, turning the heuristic $h(s)$ into the goal distance in terms of needed actions.

3.2 Monte-Carlo tree search

The Monte-Carlo tree search algorithm (MCTS, [4]) tries to find a solution by building up a state-space tree and step-wise expanding it, using randomized simulations.

Algorithm 2 Monte Carlo tree search

```
for iterations do
    $s_0$ ← initial state
    while $s_n$ all children initialized do                  ▷ Selection
        $s_n$ ← select child-state from $s_{n-1}$ with action $a_{n-1}$
    end while
    $s_{n+1}$ ← execute new action $a_n$ in $s_n$                ▷ Expansion
    reward ← simulate $s_{n+1}$                               ▷ Simulation
    $s_0, a_0, s_1, a_1 ... s_n, a_n$ ← backpropagate reward   ▷ Backpropagation
end for
next action ← best action in $s_0$
```

The algorithm (shown as pseudo code in 2) consists of four steps: selection, expansion, simulation and backpropagation. These steps are repeated until a termination criterion is reached. Such a criterion could be based on time, certainty or number of iterations.

Following, the 4 steps are described in detail.

Selection

The selection step navigates the expanded search tree. Starting from the root, state actions are selected until a node with un-expanded child is reached.
The goal of this step is to select the most promising tree region. For that, the algorithm has to choose good actions for every tree node, successively moving down the tree.

$$\underset{a}{\operatorname{argmax}}\, Q(s,a) + c \cdot \sqrt{\frac{\ln \sum_b N(s,b)}{N(s,a)}} \tag{3}$$

Most commonly the UCT (upper confidence bound applied to trees, [5]) equation is used (shown in equation 3). UCT is derived from UCB1, which is shown in formula 19. It selects the child which maximizes the average action result $Q(s,a)$ in addition to an uncertainty value, incorporating the action selection count $N(s,a)$. With an increasing action count, the uncertainty value decreases over time leading to a greedier behaviour. The exploration parameter c is commonly set to $\sqrt{2}$, leading to a good trade-off between exploration and exploitation.

Expansion

When the selection step reaches a tree node with an un-expanded child, the child gets added to the tree, continuing with its simulation.

Simulation

This step executes a simulation for the newly expanded child node. This part of the algorithm, also called rollout, executes, in the most basic implementation, random actions starting from the new child state. The simulation is terminated when a terminal state, dead-end or a maximum number of steps are reached. The resulting value of the rollout then gets propagated back through the tree in the next step.

Backpropagation

In the backpropagation step, the visited tree nodes during the selection and expansion phase gets updated. The rollout value gets propagated back through the search tree, updating the corresponding nodes action values $Q(s,a)$ and increasing their action execution counters $N(s,a)$.

Action selection

When the algorithm termination criterion is met, the root action to be executed is chosen. This is most commonly done by either selecting the most visited action, maximizing $N(s_0,a)$, or the best predicted action, maximizing $Q(s_0,a)$.

Improved MCTS for classical planning

In this work, the MCTS algorithm is applied to Sokoban problems, which are problems from the classical planning domain. As these problems have properties like full observability and deterministic actions, the MCTS algorithm can be improved to speed up the search process. Additionally, there is no need of an "in time" action selection, which allows MCTS to run until a solution is found. In this case, the algorithm is modified to run until a state expansion leads to a terminal state (in Sokoban all terminal states are goal states). The taken inner tree path is then returned as solution. This modification additionally eliminates the risk of prematurely choosing an action which possibly leads to a dead-end.

Following, further improvements are presented.

Easy pruning

Figure 5 shows an exemplary Sokoban state transition. With the player in cell s_0, he can move up into cell s_1. Being there, the player stands in front of a wall, giving him the possibility of moving to the left (cell s_2), the right cell (s_3), back to cell s_0 or moving up against the wall, staying in the current cell s_1. Without loosing solution optimality, the up and down actions in state s_1 can be removed. Generalised, modifying the MCTS expansion step to check whether the resulting child-state of an action is identical to the current state (action has no effect) or the same as the parent state (going back), incorporates an additional cost in the order of $\mathcal{O}(1)$, while reducing the trees branching factor from 4 to mostly 3 (there is always an action leading back when no box was pushed), or in the most extreme case to 0 (walking into an alley with a dead-end). When all children of a state are pruned away, the parents state action leading to this state can also be removed. This can be easily checked in the backpropagation step.

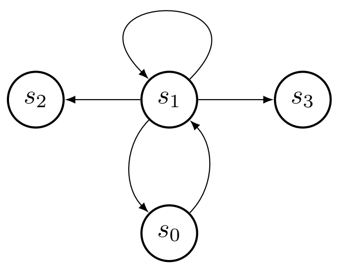

$$\begin{tikzcd} s_2 & s_1 \arrow[l] \arrow[r] \arrow[loop above] \arrow[d, bend right] & s_3 \\ & s_0 \arrow[u, bend right] & \end{tikzcd}$$

Figure 5: Sokoban transition

MCTS-T+

"Monte Carlo tree search for asymmetric trees" [7] presents a modified MCTS algorithm, improving the search performance for asymmetric trees, possibly containing loops.

While the standard MCTS algorithm is predominantly used for two-player board games, showing a mostly symmetric search tree without many loops, its performance can be quite poor if this is not the case.

To improve the performance in such cases (the Sokoban domain is one of them, as it can show a massive amount of state loops) MCTS-T+ incorporates an uncertainty variable σ. This variable, ranging from 0 (no uncertainty) to 1 (maximal uncertainty), is added to every node in the tree.

The classic MCTS algorithm steps are modified as follows:

Selection

To incorporate the additional uncertainty value in the action selection, the UCT formula is modified as shown in equation (4).

$$\underset{a}{\operatorname{argmax}}\ Q(s,a) + c \cdot \sigma(s') \cdot \sqrt{\frac{\ln \sum_b N(s,b)}{N(s,a)}} \tag{4}$$

As the uncertainty values σ, taken from the corresponding child states s', are successively lowered with progressing exploration, the selection process prefers actions with higher q-values.

Expansion

In the expansion step, the newly added state is checked for terminality or occurrence in the trace (all tree states encountered in the current iteration). If one is the case, the states σ value is set to 0 indicating a fully explored sub-tree, else it is initialized to 1. When a loop is detected (the current state being in the trace), the states value is set to the value the simulation would return, walking this loop until reaching the max rollout steps*. In the other case, where the expanded state is not terminal and not in the trace, it gets simulated as usual.

*In detail, MCTS-T+ uses a fixed tree depth, calculating the number of rollout steps as the difference between the current depth and the max tree depth. This allows to calculate a sensible loop rollout value. With the change to run the algorithm until a terminal state is present in the tree, a bad max tree depth value might prevent problem solving, while no max tree depth creates an issue evaluating loops. A too bad loop value might punish the whole sub-tree and a value which is too good could, with growing tree depth, possibly encourage selection.

Backpropagation

In the backpropagation step, the uncertainty value gets recursively backup-ed to the predecessor states in the tree, following formula (5).

$$\sigma(s) = \frac{\sum_a m(s,a) \cdot \sigma^*(s')}{\sum_a m(s,a)} \tag{5}$$

With $m(s,a)$ and $\sigma^*(s')$ defined as:

$$m(s,a) = \begin{cases} N(s,a), & \text{if } N(s,a) \geq 1 \\ 1, & \text{otherwise} \end{cases} \quad \sigma^*(s') = \begin{cases} \sigma(s'), & \text{if } N(s,a) \geq 1 \\ 1, & \text{otherwise} \end{cases}$$

As shown, if an action has never been taken the corresponding $m(s,a)$ and $\sigma^*(s')$ variable is set to 1, showing maximum uncertainty.

The $Q(s,a)$ value is then updated in an off-policy manner, as the average reward is skewed towards results stemming from deeper pathes. This is done by calculating the action, that would have been taken using the standard UCT formula (3). These off-policy action counts denoted as $C(s,a)$ are then used as shown in formula (6), weighting the child's q-values and adding the reward $r(s,a)$ for action a in state s.

$$Q(s,a) = r(s,a) + \frac{\sum_{a'} Q(s',a') \cdot C(s',a')}{\sum_{a'} C(s',a')} \tag{6}$$

Action selection

As the action count is skewed towards deeper tree regions, the action with the highest q-value is selected.

UCT*

While the previously presented MCTS-T+ approach only handles loops (reoccurring states in one trace) UCT* [15] also handles transpositions (reoccurring states in the whole tree).

UCT* modifies the following steps from the classic MCTS algorithm:

Expansion

After the selection phase terminates at a leaf node, all its children are created. Before adding the children to the tree, they are checked for existence against the current tree.

- In case that the state is already present in the tree, the cost of both pathes from the root are compared. When the new path is not better, the corresponding child gets disabled. Otherwise, the corresponding, existing sub-tree from the duplicates position is moved to the currently expanded child. The moved sub-tree is than re-evaluated, checking for better pathes for all disabled children in the sub-tree.
- When the created child is a new unseen state, it gets added as usual to the tree.

Simulation

Instead of a rollout, UCT* uses a heuristic function to directly evaluate the state-value.

Backpropagation

As the expansion step possibly changed the children of multiple tree nodes, the backpropagation step has to be run on all expanded nodes and parent-nodes of moved sub-trees.

While the classic MCTS backpropagates the rollout value, UCT* uses state-values $Q(s)$, updating them to the best child state value $Q(s')$ following formula (7).

$$Q(s) = \max_{a} k \cdot r(s,a) + Q(s') \tag{7}$$

For UCT* k has a value of 1, for GreedyUCT* (another deviation proposed in the paper) k is set to 0, ignoring the action cost inside the tree.

3.3 Comparison

With the presented MCTS improvements, some overhead is introduced. Following, the different implementations are compared on the Sokoban domain. The algorithms were tested on 50 instances of 7×7 games for each number of boxes 1-3, measuring coverage (percentage of solved problems), the average needed time[2] and the solution length (divided by the FF solution length). Using a cutoff at 10k iterations, the search was cancelled when reached, counting the problem as unsolved and omitting the elapsed time. For all MCTS implementations the reward function described in section 6 was used and c in UCT was set to $\sqrt{2}$.

UCT* was additionally tested with pruning. Instead of only disabling state actions without effect, leading to a recheck for a better path at sub-tree movement, they can be pruned. This spares the recheck-time as the path to itself with an additional step (assuming no positive step reward for actions without effect) will never be better. Actions leading back to the parent node can not be pruned (like in easy pruning) as their parent might change due to a sub-tree movement.

[2] All time measurements in this thesis are denoted in seconds, using a i7-5820K CPU and a NVIDIA GeForce GTX 980 Ti GPU for neuronal networks.

Algorithm	Boxes	Coverage	Average time	Average length
A*	1	**1.0**	0.48 (0.45)	**0.88 (0.22)**
(BFS)	2	**1.0**	5.85 (6.66)	**0.88 (0.12)**
	3	0.7	17.78 (14.13)	**0.88 (0.13)**
MCTS	1	0.78	2.48 (4.13)	0.88 (0.23)
	2	0.42	3.23 (5.5)	0.94 (0.15)
	3	0.2	6.86 (6.55)	0.96 (0.08)
MCTS	1	0.96	1.44 (3.05)	0.87 (0.23)
(easy pruning)	2	0.68	3.66 (5.97)	0.95 (0.16)
	3	0.54	7.13 (10.11)	0.99 (0.16)
MCTS-T+	1	0.94	3.44 (5.29)	0.95 (0.27)
	2	0.46	7.08 (9.41)	1.0 (0.13)
	3	0.18	10.59 (7.27)	1.02 (0.13)
UCT*	1	**1.0**	0.38 (0.42)	**0.88 (0.22)**
	2	**1.0**	1.37 (1.85)	0.93 (0.15)
	3	**1.0**	4.54 (8.84)	0.94 (0.16)
UCT*	1	**1.0**	**0.37 (0.42)**	**0.88 (0.22)**
(with pruning)	2	**1.0**	**1.36 (1.82)**	0.93 (0.15)
	3	**1.0**	**4.5 (8.81)**	0.94 (0.16)

Table 1: Search algorithm comparison

Table 1 shows the measurement results, categorized by the number of boxes, with the standard deviation in brackets and the best results written in bold. All MCTS implementations were run without simulation, as the collected rewards from unguided rollouts systematically worsened their performance.

A* (breadth-first search, as no heuristic was used) shows a good performance in terms of coverage, having solved all one- and two-box problems.
Classic MCTS performed relatively bad, getting clearly outperformed by the easy pruning version.
The MCTS-T+ algorithm (implementation taken from the corresponding Git repository [8]) shows poor results. With the coverage more or less aligned to MCTS, the average needed time to solve a problem is considerably higher. As the algorithm was run without a maximum tree depth, the loop rollout was fixed to 100 steps.

Another reasonable modification would be to prune states which re-occur in the trace from the tree, turning the algorithm into an extended version of the easy pruning MCTS version. The poor performance in terms of elapsed time, could be caused by the implementation method. While MCTS-T+ starts every iteration with the root environment executing every selected action on it while traversing the tree, the other implementations keep the corresponding environment copy in each tree node, sparing the Gym step re-execution time.
The best performing algorithm UCT*, shows the lowest time consumption over all box counts, having additionally solved every single problem. By additionally pruning some actions, the average time was further marginally reduced. GreedyUCT*, which ignores in-tree action rewards, yielded worse results when used without rollout, as this leads to a completely unguided search algorithm.

4 From planning to machine learning

A classic planning task can be defined with a 4-tuple $\sqcap = \langle A, I, O, \gamma \rangle$ where:

- A is a finite set of state variables
- I is the initial state
- O is a finite set of operators over A
- γ is the goal

For a trivial Sokoban problem with one box (as shown in figure 6), $\sqcap$ can be defined as (with $\forall X \in \{0,1,2,3,4\}$ and $\forall Y \in \{0,1,2\}$):

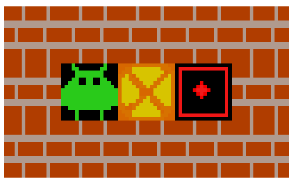


Figure 6: A trivial Sokoban problem

- $A = \{[X:Y]_{free},\ [X:Y]_{box},\ [X:Y]_{goal},\ [X:Y]_{player},\ box_{at-goal}\}$
- $I = \{[1:1]_{free} \mapsto T,\ [2:1]_{free} \mapsto T,\ [3:1]_{free} \mapsto T,$ $[1:1]_{player} \mapsto T,\ [2:1]_{box} \mapsto T,\ [3:1]_{goal} \mapsto T,$ all others $\mapsto F\}$
- $O_{[X:Y]-right}\ \forall [X:Y]_{free}$; moves the player one step right, if possible
  - precondition: $[X:Y]_{player} \wedge [X+1:Y]_{free} \wedge$ $(\neg [X+1:Y]_{box} \vee ([X+2:Y]_{free} \wedge \neg [X+2:Y]_{box}))$

  - effect: $\neg[X:Y]_{player} \wedge [X+1:Y]_{player} \wedge$
    $([X+1:Y]_{box} \triangleright (\neg[X+1:Y]_{box} \wedge [X+2:Y]_{box})) \wedge$
    $(([X+1:Y]_{box} \wedge [X+2:Y]_{target}) \triangleright box_{at-goal}) \wedge$
    $(([X+1:Y]_{box} \wedge \neg[X+2:Y]_{target}) \triangleright \neg box_{at-goal})$
- $O_{[X:Y]-left}$, $O_{[X:Y]-up}$, $O_{[X:Y]-down}$ correspondingly
- $\gamma = box_{at-goal}$

$O_{[X:Y]-right}$ describes the logic operation for moving the player, currently positioned at X:Y one step to the right, possibly moving a box. The precondition for applying a specific right operator, say $O_{[1:1]-right}$, is that the player is currently positioned at location 1:1, the position on the right (position 2:1) is not a wall and if a box is at this position, that 3:1 is free (not a wall) and has no box. With this precondition fulfilled, the operator can be applied to the current state, updating the players position to 2:1. Optionally moving a box, currently located at 2:1, one step to the right.

Learning a neuronal network based policy to predict the best next action, rises different questions. The state variables A have to be encoded for network input, additionally handling the changing variable count for different instances. With different problem instances also come a different amount of grounded actions and goal variables.

Succeeding, the necessary steps to close the gap between planning and machine learning are discussed, presenting different publications and their proposed solutions. In detail looking into action schema networks (ASNet, [19]) and generalized reactive policies (GRP, [2]).

4.1 State encoding

With the state variables as atomic propositions (as $T[rue]$ or $F[alse]$), the state can be directly expressed using a binary input array. When a categorical state representation is used, it can be either one-hot encoded or reformulated into a corresponding atomic representation.

Over different problem instances from the same domain, the number of state variables can change. In case of Sokoban, increasing the game size adds 4 state variables per added grid-cell (when using the plan described earlier). The changing size of the state representation is an issue for machine learning as classic neuronal networks have a fixed input size.

To overcome this problem, the action schema network (ASNet) approach trains so called modules. These modules have comparable properties to a dense layer. They are defined as:

$$\phi^l = f(W_f^l \cdot u^l + b_f^l) \tag{8}$$

where the modules output ϕ^l is a feature vector (the authors used 16 features). Each feature in a module can be seen as a node of a dense layer, having a set of weights W_f^l and a bias b_f^l. The modules output is then calculated by applying a non-linear activation function on its feature outputs. The authors use the ELU (Exponential Linear Unit) activation function, described in formula (9).

$$f(x) = \begin{cases} \alpha \cdot (e^x - 1), & \text{If } x < 0 \\ x, & \text{otherwise} \end{cases} \tag{9}$$

For action modules, the input u^l is constructed by concatenating the feature vectors of related proposition modules in the previous layer. A proposition module is related to an action module, when the proposition occurs as precondition or in an effect in the underlying action of the action module.

The proposition modules, having the same internal structure, represent state variables. They get the output of all related action modules in the preceding layer as input. An action module is related when it has the proposition modules underlying state variable in precondition or effect. In contrast to the input generation for action modules, the input for the proposition modules have to be pooled (the authors use max-pooling). This is necessary as related action modules can vary in number. For example, looking at the Sokoban problem predicate $[X : Y]_{box}$ described in the planning task earlier, shows that such a variable located in a corner can have 4 related ground actions (two for each free adjacent location)[1], while another has possibly 8 related actions (when all adjacent cells are free).
Finally, for training and action prediction, the modules are build up to a whole network with a predefined depth. This is done by alternating for every layer between action and proposition modules, starting with action modules in the first layer and finishing the network with a final action module layer. Each action module in the final layer only outputs a scalar, used for the action probability calculation with softmax.
In each layer, action and proposition modules (one for each grounded action and state variable) share their weights and biases when they stem from the same lifted action or predicate.

The generalized reactive policy (GRP) approach, predicts actions on Sokoban state images. As they are using a convolutional neuronal network (CNN), the width and height of the input image does not matter for the convolutional layers. When transitioning to fully connected layers, the output of the last convolutional layer is cropped around the players position. While this allows input of various sizes, the network needs to have a sufficient depth, to be able to grasp the whole Sokoban problem. Due to the fixed depth of the trained network, the problem size is implicitly limited.

[1]The described operators in the abstract Sokoban task check for an existing box in the next location and in the location thereafter.

As this work also uses the Sokoban domain, a CNN is applied. This improves the prediction performance, as it allows to exploit the spatial distribution of the grid-like states. As generalization solution, the network is fixed to an input of a 19x19 Sokoban problem, allowing the action prediction for all problems up to that size. This is possible, as every smaller problem instance can be padded with walls, without changing the problem itself.

4.2 Action encoding

Having the state as input, the policy network should output the goal leading probability of every applicable action in the current state.
To be able to use the same trained network for all states the policy has to be learned on all grounded actions (irrelevant of applicability in a specific state). When a state is evaluated the non-applicable actions have to be masked out before the softmax calculation, to get a valid probability distribution over all applicable actions.

The changing amount of grounded actions for different instances is no problem for ASNet, as the network is rebuild for every instance using the corresponding amount of action modules.

In the Sokoban domain, the grounded actions for all instances can be generalized to at most four applicable actions in every state. This is possible, as each grounded action has a specific player-location as precondition.

4.3 Goal encoding

To be able to learn a goal-leading policy, the goal has to be given as input to the network.

The ASNet approach adds an additional binary dimension to the input, stating whether the grounded proposition appears in the goal or not.[2] Optionally, as the first layer of an ASNet represents the grounded actions, additional action related heuristics can be added. In the Paper, disjunctive action landmarks[3] from LM-cut are used. Each action module in the first layer then gets additional information whether the corresponding action is the only action in at least one landmark, an action in a landmark with two or more actions, or does not appear in one landmark.

GRP, in contrast, directly works on images and thus adds the goal state image to the policy network's input. Furthermore GRP uses skip connections, adding the network's input to every following convolutional layers input.

As Sokoban is a puzzle game designed for humans, the goal is already encoded into the input, as every box target is marked on the grid. This makes the goal static with respect to the current state, making additional input as used in GRP, redundant.

[2]This limits the problem representation to goals which are conjunctions of positive literals.

[3]Given a planning task $\sqcap$. A disjunctive action landmark is a set of actions, from which each plan for $\sqcap$ contains at least one action.

5 Imitation learning

One possible approach to learn a policy is to learn by imitation. For that, the problem solutions have to be given for training. The downside of this, is that the performance of the learned policy is generally upper-bound to the solution quality of the planner algorithm.

5.1 Generating labels using the Fast Forward planner

To generate the needed problem solutions the Fast-Forward (FF v2.3, [3]) planner is used. Important to note is that the solutions from FF are inadmissible (not necessarily optimal). While this allows to solve more problems in the same time (resulting in more training data), it can have a negative impact on the training performance (as shown in section 5.2.2).

The Fast Forward planner FF won the planning systems competition (AIPS: Artificial Intelligence Planning and Scheduling) in the year 2000 and is still a widely used planner today. FF solves planning problems using a guided state-space search.
As guidance, a special H^{FF} heuristic is used. This heuristic is calculated using the relaxed version of a given problem, which ignores all delete effects in the operators. Then, a relaxed planning graph is built using forward chaining (comparable to Graphplan). Finally, the heuristic value for a state is calculated as its relaxed plan length,

extracted from the graph by chaining backwards.
Beginning in the start state s, FF runs a pruned breadth-first search (BFS) until a successor state s' with better heuristic is found, restarting the search in s'. This strategy, also called enforced hill climbing (EHC), is guaranteed to find a solution, unless the planning problem contains dead-ends. When EHC fails to find a solution, FF falls back to best-first search. This happens when a chosen state s' with a lower heuristic than s inevitable leads to a dead-end.

To solve a Sokoban problem with the FF planner, a problem and domain description is needed. For that, the Sokoban PDDL (Planning Domain Definition Language) description from PDDLGym [18] is used. This domain definition uses 3 lifted actions, move, push to goal and push to non-goal. Given a Gym Sokoban problem, its state is translated into a PDDL task. This task is then solved with the FF planner, parsing its response and returning a solution, consisting of left, right, up and down actions.

5.2 Imitation learning

Using the 9k Sokoban problems in the train set, 178k state-action pairs for the policy training are created by collecting states and their one-hot encoded actions, following the FF given solutions. For the heuristic network, training state and goal-distance pairs are collected in the same way, additionally applying oversampling to balance the goal-distance value distribution, resulting in 360k training pairs. For the network validation, 19k datapoints are collected from the Sokoban validation set, centering problem instances which are smaller than the input size.

The training data is additionally augmented. While the final training uses a data sequence generator, which allows to train the networks on a differently augmented dataset in each epoch, the augmentation for the network search is pre-computed for better comparison.

This is done by iterating 4 times over the dataset, collecting randomly rotated, mirrored and placed (when smaller than the input size) training states.

5.2.1 Searching a network with HpBandSter

HpBandSter is a python framework for different hyperparameter optimization algorithms. Hyperparameters, like neural network shape, learning rate, batch-size and so on, need to be searched to find a good setup which can generalize well on given training data. Basic optimization algorithms are for example grid-search and random-search. While random-search shows a better performance than grid-search (due to higher amount of tested parameter values), an even better approach has been published lately. This approach called BOHB (Bayesian optimization HyperBandSter [1]), uses a Bayesian approach to further improve the guidance of the optimization search.

Activation function Neuronal networks use an activation function after each layer. While the activation function used for the output layer is fixed by the desired output, those used in the preceding layers can be chosen freely. Today, the rectified linear unit (ReLU) function is commonly used (shown in figure 7 in red), as other functions, such as the logistic function, have the vanishing gradient problem when used with deep neuronal networks.

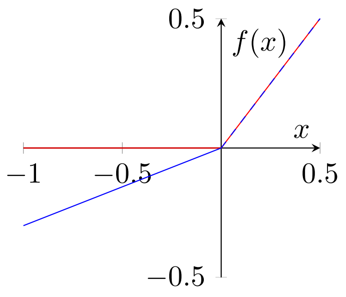


Figure 7: ReLU and Leaky ReLU ($\alpha = 0.3$)

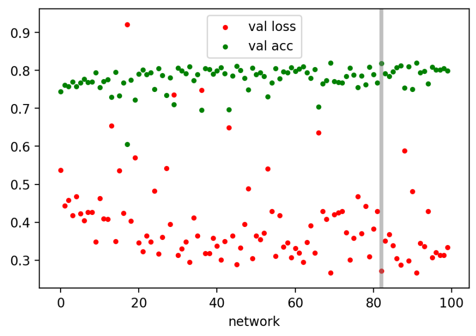
$$f(x) = \begin{cases} \alpha \cdot x, & \text{If } x < 0 \\ x, & \text{otherwise} \end{cases} \tag{10}$$

A modified version of ReLU is the Leaky ReLU activation function. This function, shown in figure 7 in blue, with the corresponding formula shown in (10), has an additional α variable changing the slope for values smaller than 0. Comparing ReLU and Leaky ReLU($\alpha = 0.3$) using a deep convolutional neuronal network (CNN) without skip connections on the Sokoban training set, showed an improved learning performance when using Leaky ReLU, in terms of validation loss and validation accuracy.
This observation has also been made in "Empirical evaluation of rectified activations in convolution network" [23]. The authors compared ReLU, Leaky ReLU and PReLU (Parametric ReLU, where α is learned) activation in CNNs on the CIFAR-10 and CIFAR-100 dataset. Those datasets are common benchmark datasets, containing 32×32 RGB images (50k training and 10k test images) of 10 respectively 100 classes. Using no pre-processing or augmentation, they showed an improved classification performance using Leaky ReLU with $\alpha = 0.01$ and further improvement using $\alpha = 0.18$. Additionally showing that PReLU performed even better.
Training on Sokoban data using PReLU and learning one α value for each filter, did not show any further improvement in comparison to Leaky ReLU with $\alpha = 0.3$.

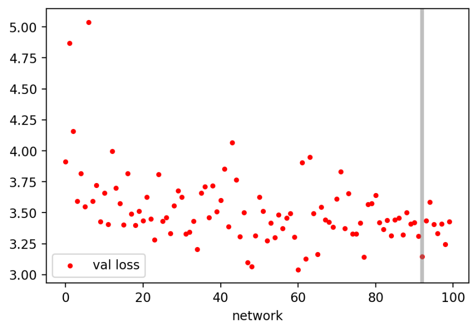
Having figured out a good activation function (which also could have been searched), BOHB was run for 100 iterations with a budget of 10 epochs on the following hyperparameter search domain:

- number of convolutional layer: 4 - 10
- number of convolutional filter: 1 - 32
- convolutional filter size (using 'same' padding): 3 - 9
- max pooling: yes (size 2×2, stride 2×2) / no
- additional dense layer: yes / no
- nodes in the dense layer: 10 - 100

To reduce the search dimension, the learning rate and batch-size were fixed to Adam optimization with a learning rate of 1e-3 and a batch-size of 256. These are common values, which also showed a stable learning performance while training networks manually. Additionally as explored previously, the Leaky ReLU activation function with $\alpha = 0.3$ is used.



Policy network search



Heuristic network search

Figure 8: BOHB network search

Figure 8 shows the found accuracy and loss values during the BOHB search, sorted by the underlying networks capacity, expressed as the trainable network parameters (the used loss and accuracy functions are presented in detail later). It can be observed that deeper networks are better at capturing the underlying logic, which has also been shown by the authors of the generalized reactive policy paper [2].
Marked in gray are the chosen networks, showing the lowest training loss. The networks with the lowest validation loss showed worse results when trained on more state-action pairs.

The chosen policy network has 6 convolutional layers and an additional dense layer with 14 nodes, finishing the network with 4 fully connected nodes for the action output. The heuristic network has 7 convolutional layers with one fully connected node for the scalar output. Table 10 in the appendix describes the networks in more detail.

While BOHB was run searching for network architectures using a 13×13 state input, the training on the final policy and heuristic network is done using a 19×19 input. This change only has an effect on the number of weights in the first dense layer. Comparing the training performance between both input sizes showed no negative effect on the reached validation loss and accuracy.

5.2.2 Policy learning

Figure 9 shows the training progress over 30 epochs (after the iterative improvement, described later in 5.2.2). Every epoch trained on randomly augmented state-action pairs from the Sokoban train set, minimizing the cross-entropy loss (11, for one datapoint) on the one-hot encoded actions (the training loss is shown in red).

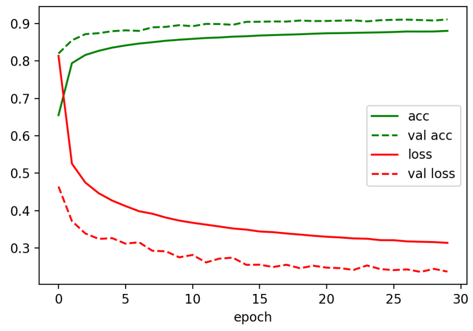


Figure 9: Policy network training

$$L(y, \hat{y}) = -\sum_{a \in A} y_a \cdot \log(\hat{y}_a) \tag{11}$$

At the end of each epoch, the networks validation loss (shown as red dashed line) was calculated on the evaluation state-action pairs, reaching a final loss value of 0.23. Additionally, the categorical accuracy was evaluated, measuring the percentage of data-points where the best predicted action (having the highest probability) matches the labeled action (shown in green and dashed green). After training for 30 epochs, the network reaches a validation accuracy of 91%.

Analysing the learned policy on the validation environments, shows that the network is able to find better solutions of up to half the length, compared to some quite poor FF solutions.

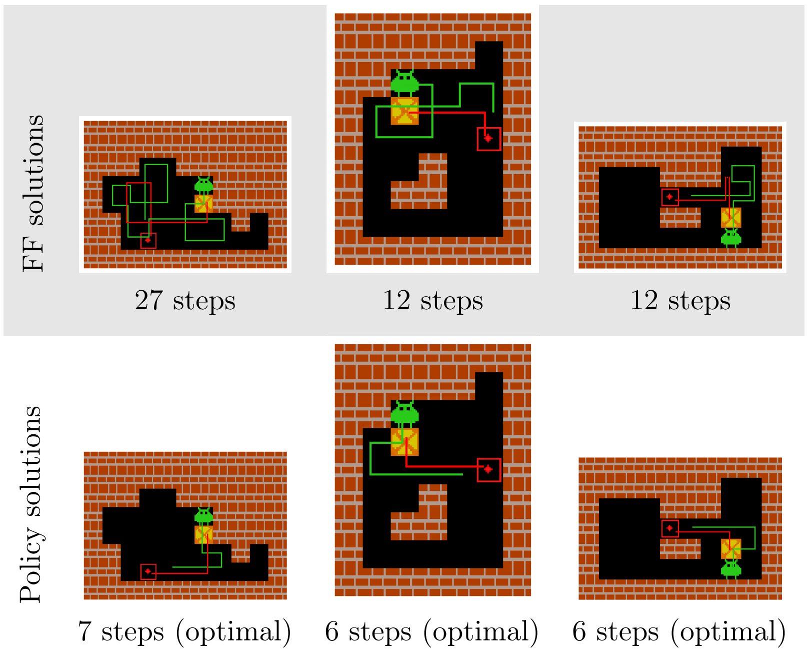


Figure 10: FF solutions vs policy solutions

Figure 10 shows 3 exemplary validation Sokoban problems and their solutions, marking the player path in green and the box path in red. The FF solution pathes are shown in the first row (highlighted in gray) and the better policy solutions in the second row. There are also better found solutions with more boxes which are not shown here, as their pathes are hard to visualize.

Iterative improvement Due to the observation made above, the Sokoban train and validation solutions were iteratively improved using the learned policy. By repeatedly improving the solutions and retraining on the improved data, the network was able to step-wise increase the validation accuracy from 79% to 91% accuracy. Improving 38% of the FF solutions (37.2% in validation and 38.9% in train set) by shortening them 20% on average, during 4 iterations.

Evaluating the trained policy network greedily (taking the action with the highest probability) on the validation problem set, shows that the network is able to solve 93% of the one-box problems, 69% of the two-box problems and 46% of the three-box problems.

5.2.3 Heuristic learning

Figure 11 shows the training progress of the heuristic network over 30 epochs. The network trained on equally weighted goal-distance values between 1 and 40, using a modified mean-absolute-error loss function (12) with $k = 0.3$. The red line shows the training loss, reaching a value of 2.6 and the dashed line shows the validation loss reaching 2.4.

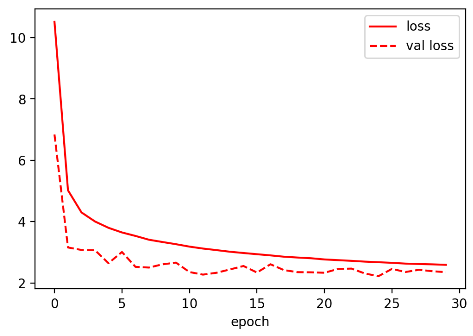


Figure 11: Heuristic network training

$$L(y, \hat{y}) = \frac{1}{N} \sum_{i=1}^{N} \left(\, |\hat{y}_i - y_i| + k \cdot (\hat{y}_i - y_i) \, \right) \tag{12}$$

As a heuristic for A* should rather underestimate the needed steps to solve a problem than overestimate them, possibly leading to longer solutions, a custom asymmetric loss function is introduced.

This custom loss function, shown in formula (12), is based on the mean-absolute-error (MAE) loss. By adding $k \cdot (\hat{y}_i - y_i)$ to the loss value, goal-distance underestimations are only punished by a factor of $1 - k$ while overestimations are punished $1 + k$ times, in contrast to an undistinguished factor of 1 in MAE ($k = 0$).

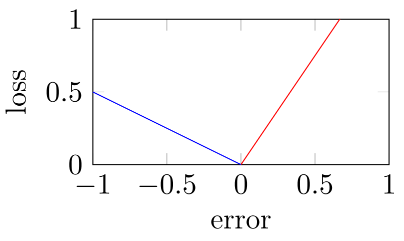


Figure 12: Increased admissibility loss function ($k = 0.5$)

Figure 12 plots the custom loss function for $k = 0.5$, showing the overestimation loss in red and the underestimation loss in blue.

Heuristic	Boxes	Coverage	Average time	Average length
None	1	0.96	1.54 (1.51)	0.99 (0.06)
(BFS)	2	0.26	2.4 (1.87)	0.99 (0.05)
	3	0.05	1.87 (1.64)	1.0 (0.0)
Manhattan	1	0.99	0.57 (0.81)	0.99 (0.06)
	2	0.46	2.63 (2.3)	0.97 (0.08)
	3	0.13	2.89 (2.39)	0.98 (0.07)
MAE	1	**1.0**	0.39 (0.46)	0.99 (0.06)
	2	0.99	1.8 (2.55)	0.96 (0.11)
	3	0.89	3.03 (4.09)	0.93 (0.14)
Increased	1	**1.0**	0.36 (0.4)	**0.99 (0.06)**
admissibility	2	0.99	1.61 (2.31)	**0.95 (0.1)**
$k = 0.3$	3	0.91	2.65 (3.55)	**0.91 (0.13)**
MAE	1	**1.0**	0.31 (0.32)	1.0 (0.08)
$w = 3$	2	**1.0**	1.12 (1.86)	1.01 (0.14)
	3	0.94	2.2 (3.66)	0.99 (0.18)
Increased	1	**1.0**	**0.29 (0.3)**	1.0 (0.07)
admissibility	2	**1.0**	**1.04 (1.44)**	1.0 (0.14)
$k = 0.3, w = 3$	3	**0.95**	**2.06 (3.29)**	0.98 (0.16)

Table 2: Heuristic comparison using A*

Table 2 shows the evaluation results, evaluating the learned heuristic on the validation Sokoban problem set using A*. The best values in each category, 1 to 3 boxes are written in bold. The shown results were measured using a cutoff at 1k iterations. Having solved an environment under this threshold, the elapsed time and the solution length were taken into account else the problem was counted as unsolved. The solution lengths were measured relative to the policy improved FF solutions.
For comparison, the A* performance was also tested using a custom Manhattan heuristic[1] and a second heuristic network trained with the standard mean-absolute-error loss.

[1]The heuristic value is calculated by summing up the Manhattan distances between each box and its closest target position, ignoring all walls. This heuristic is consistent as its value for a goal state is zero and with a fixed action cost of 1 the heuristic never improves by more than 1 going from state s to a child-state s'.

The measurements show that all trained heuristic networks clearly outperform the Manhattan heuristic or no heuristic (breadth-first search) in terms of coverage and average time.
Comparing MAE and the custom loss function (denoted as increased admissibility) shows that the custom loss function slightly outperforms in terms of coverage, time and solution length. This out-performance also holds when searching with weighted A* using $w = 3$ (described in section 3.1).

5.3 Imitation learning with exploration

As the trained policy network still shows potential for improvement, further learning methods are explored.

ASNet presents a new imitation learning algorithm using exploration. This algorithm can be seen as an extension to the DAgger (Data aggregation, [10]) algorithm. DAgger lets the learned policy explore for a number of steps, collecting the visited states. Those states then get labeled by an expert (in this case the planner) and added to the training dataset. After improving the policy on the collected dataset, a new iteration is started, repeating all previous steps.
ASNet extends this algorithm by not only adding the encountered states visited by the policy to the dataset, but also the states on the planners solution path. In the best case, this would be done using an admissible planner. As this is not feasible in most cases, the authors of ASNet investigated the influence of possibly sub-optimal solutions. Examining the problem domains Triangle Tire World, CosaNostra Pizza and Probabilistic Blocks World with the SSiPP [20] planner they came to the conclusion that sub-optimality is sufficient, as it allows to run more algorithm iterations and thus collecting more state-action pairs, instead of using the time on solving the sub-problems optimally.
Another difference to the original DAgger algorithm is that ASNet trains the policy by sampling mini-batches from the collected dataset, while DAgger uses full epochs (training the policy on all datapoints in each iteration).

Using ASNet's proposed algorithm with the FF planner on the Sokoban domain led to a worsening of the previously imitation learned policy. This problem is possibly caused by FF's unexpected behaviour described in section 5.3.1.

As previously observed, the imitation learned policy is already able to solve some problems better than the FF planner. To reinforce the policy on better solutions, the ASNet exploration algorithm is modified for the use with classical planning problems.

Shown in figure 13 is an exemplary policy exploration trace, starting in s_0 and terminating in s_3, missing the goal state s_G.

ASNet's exploration algorithm adds all state-action pairs of the policy to the dataset, including s_2 with its action leading to s_3 from where the goal state is not reachable anymore (dead-end). Exploring beyond s_3 leads the algorithm to save even more state-action pairs, yielding no information to learn. Additionally, all planner solutions from $s_{0|1|2}$ to s_G, from which only s_1 to s_G with respect to s_0 is optimal, are saved and trained on.

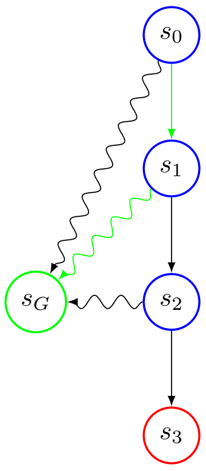


Figure 13: Exploration schemata

Now, with the new modified exploration algorithm the potential solution length for every state is tracked. After the exploration, the data collection is started with s_3, checking whether the state is a goal-state or an FF solvable state (proving that the state is not a dead-end). As s_3 is a dead-end state, the state is dropped (not saved to the dataset). Continuing with s_2, the state is again checked with FF. When the planner finds a solution, its length is compared to the potential policy solution. As the policy didn't solve the problem, all planner state-action pairs s_2 to s_G are saved. Evaluating s_1 next, shows that the FF plan is again shorter compared to $s_1, s_2..s_G$, adding all data-points from s_1 to s_G to the dataset and dropping the policies s_1 to s_2 state-action pair. Finishing with s_0, the planner returns a longer solution to the goal compared to $s_0, s_1...s_G$.

By then dropping the FF solution and only adding the s_0 to s_1 state-action pair, the policy network gets reinforced.

Algorithm 3 Modified imitation learning with exploration (exploration learning)

```
memory ← initialize FIFO memory
$\theta$ ← initialize policy network
for iterations do
    $p \leftarrow p \in P_{train}$                              ▷ Select train problem
    $s_0, a_0, ..., s_n, a_n$ ← sample n step trajectory from $p$ following $\pi_\theta$
    distance ← 1 if $p$ solved else $\infty$
    for $s_n, ..., s_0$ do                                       ▷ Collect state-action pairs
        $s_n, a_{n_{planner}}, s_{n+1_{planner}} \cdots$ ← solve $s_n$ with planner
        if distance is $\infty$ and no planner solution then
            continue
        end if
        if distance < planner solution then                     ▷ Policy is better
            memory ← memory $\cup$ $s_n, a_n$
        else                                                     ▷ Planner is better
            memory ← memory $\cup$ $s_n, a_{n_{planner}}, s_{n+1_{planner}} \cdots$
            distance ← planner solution length
        end if
        distance ← distance + 1
    end for
    for minibatches do                                           ▷ Imitation learning
        $\beta$ ← sample minibatch from memory
        $\theta$ ← update $\theta$ towards $\beta$
    end for
end for
```

As shown in the algorithm 3, the goal distance is initialized to 1 when the policy network solved the problem else all policy state-action pairs are dropped until the planner is able to solve a state, setting the distance to its solution length. From then on, only the shortest solution pathes are added to the memory. When the FF solution for the current state is shorter than the current distance, the whole FF solution path is added to the memory and the distance is set to the FF solution length, else the policies state-action pair is added to the memory. This enables the algorithm to reinforce the policy in case it finds better solutions, allowing the network to possibly converge to optimal solutions in the long run.

5.3.1 Unexpected Fast Forward behaviour

While ASNet executes the "learning with exploration" algorithm using the SSiPP planner (as the authors use probabilistic problems), employing the FF planner might have a negative effect on its convergence. This is due to an unexpected behaviour where the FF solution diverges from the original solution, when the solution is executed on the problem environment and the encountered sub-states are re-solved.

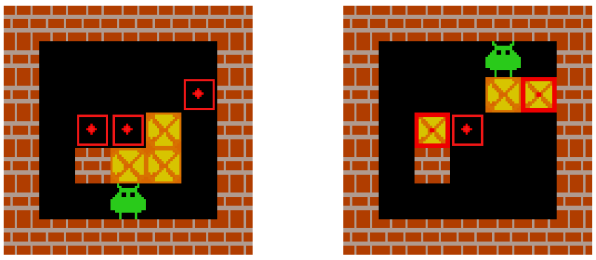

Initial state with 23 steps   Sub-state 8 vs 13 steps

Figure 14: FF solution and sub-solution

Figure 14 shows one exemplary Sokoban problem. The left image shows the initial Sokoban problem solved with 23 steps by the FF planner. The right image is a sub-state encountered following the original FF solution for 15 steps. Solving this sub-state gives a new solution with 13 steps while the remaining original solution is only 8 steps long.

5.3.2 Training

The policy training and parallel heuristic training was run iterating 4 times over the randomly ordered training problem set. As memory, a FiFo buffer with up to 500k states was used. The training was started after filling the buffer by a quarter, sampling augmented mini-batches with 256 states from it. The networks were trained using the Adam optimizer with a learning rate of 1e-3.

Due to policy exploration, an additional sensible exploration depth must be chosen. As the training environments show a wide range of FF solution lengths, beginning at 1 step problems and going up to solution lengths with 228 steps, a different exploration step-count for every problem is needed.

$$\text{steps}_{\text{max}} = \begin{cases} \lfloor 2.0 \cdot |FF_{solution}| \rfloor, & \text{if evaluation} \\ \lfloor 1.5 \cdot |FF_{solution}| \rfloor, & \text{else} \end{cases} \tag{13}$$

The maximum step-count for every problem in evaluation and test set is defined as shown in formula (13). As the upper bound for an optimal solution length is given by the FF planner, the maximum step-count can be set differently for each environment. This allows to focus the exploration around interesting problem states, while allowing long problems to be solved.[2] Furthermore, not taking the FF solution length at face-value, allows to learn a possibly more congruent policy which needs more steps to solve some problems.

Policy exploration learning Figure 15 shows the training results going over all training problems four times and evaluating the learned policy every 200 iterations on the validation set. In particular, the figure shows the reached average reward, the coverage[3] and the average solution length (divided by the improved FF solution length) over all validation problems. The measurements are separated by the box count, as a categorisation of the problems difficulty level.
The first evaluation data-point, marked with a blue horizontal line as baseline, shows the initial network performance starting the training with the imitation learned policy. This policy has a coverage of 93%, 69% and 46% for the one- to three-box validation Sokoban problems respectively (see section 5.2.2).

[2]Granting 100 exploration steps for 2 step problems is unnecessary, while there is a possible need for 200 steps to solve other problems.
[3]The percentage of solved problems taking the best predicted action up to the maximum step-count.

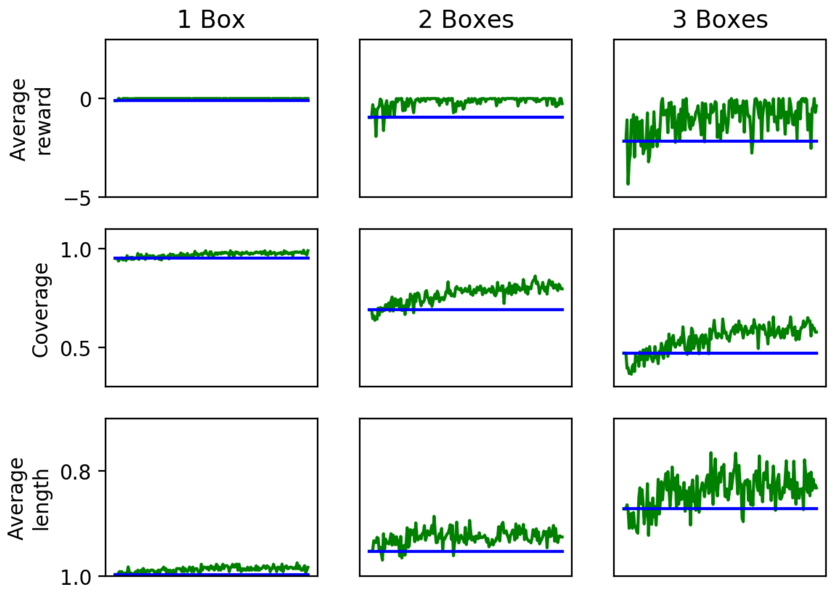


Figure 15: Modified exploration learning performance

As shown in the figure, the algorithm step-wise improved the problem coverage to 99%, 86% and 65% at peak (achieved by different network iterations).

The final policy network for this training method is built by averaging the weights from the three networks reaching the best coverage in each category. Each chosen network reaches the following coverage values, for one-, two- and three-box environments:

Best network for 1 box problems: 99%, 78% and 61%
Best network for 2 box problems: 98%, 86% and 60%
Best network for 3 box problems: 97%, 82% and 65%

By averaging the weights, the final network reaches a coverage of 98%, 87% and 71%. Surprisingly, this network beats all single policy networks for two- and three-box environments.

Measuring the solution length in relation to the already improved FF solution length, shows that the final network is able to further reduce the needed step-count. Especially solution length for three-box problems are improved, achieving in average up to 11% shorter pathes compared to the baseline.

Heuristic exploration learning As the new modified exploration algorithm already keeps track of the solution length, small changes can be made to additionally train the heuristic network. Starting with the imitation learned heuristic network, it gets further improved alongside the policy network. The heuristic was trained using the normal mean-absolute-error loss, as the sampled mini-batches are not equally distributed. Due to their skewed goal-distance distribution towards lower heuristic values, the underestimation effect is already accomplished implicitly.

Heuristic	Boxes	Coverage	Average time	Average length
Increased	1	**1.0**	0.36 (0.4)	**0.99 (0.06)**
admissibility	2	0.99	1.61 (2.31)	**0.95 (0.1)**
$k = 0.3$	3	0.91	2.65 (3.55)	**0.91 (0.13)**
Exploration	1	**1.0**	**0.25 (0.25)**	**0.99 (0.06)**
MAE	2	**1.0**	1.4 (2.08)	0.97 (0.11)
	3	**0.98**	2.46 (3.48)	0.92 (0.15)
Increased	1	**1.0**	0.29 (0.3)	1.0 (0.07)
admissibility	2	**1.0**	1.04 (1.44)	1.0 (0.14)
$k = 0.3, w = 3$	3	0.95	2.06 (3.29)	0.98 (0.16)
Exploration	1	**1.0**	0.3 (0.23)	1.0 (0.07)
MAE	2	0.99	**1.0 (1.38)**	1.01 (0.14)
$w = 3$	3	0.93	**1.74 (2.76)**	0.99 (0.18)

Table 3: Exploration heuristic comparison using weighted A*

Table 3 compares the A* search performance between the imitation and exploration learned heuristic. The search was run on the validation problems using a 1k iteration cutoff and a heuristic weighting of 1 and 3. The measurements show that the exploration learned heuristic is able to further improve the coverage for a heuristic weighting of 1, while its coverage suffers with higher weighting.

6 Reinforcement learning

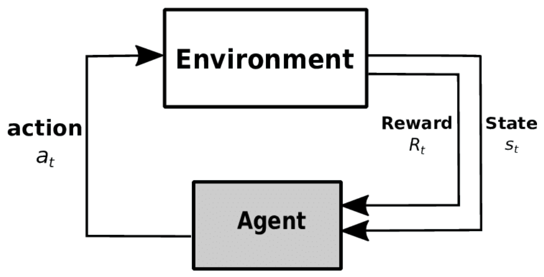


Figure 16: [Source] Reinforcement learning schema

To further improve the learned policy, reinforcement learning is used. In reinforcement learning, an agent is connected to an environment via perception and action (depicted in figure 16). In each step, the agent executes an action. The environment then changes according to its internal logic, returning a reward and the new state to the agent. The goal of the agent is to maximize the accumulated rewards during an episode. In the case of classical planning, the environment is a deterministic Markov decision process.
A Markov decision process (MDP) is defined with:

- A discrete set of n states $S = s_1, s_2, ..., s_n$, with $s_t \in S$ describing the state in timestep t.
- A discrete set of m actions $A = a_1, a_2, ..., a_m$, with $a_t \in A$ denoting the action taken by the agent in timestep t.
- A transition function $T(s_t, a, s_{t+1})$, mapping state-action pairs to the resulting new states.

- A reward function $R(s_t, a, s_{t+1})$, defining the received reward when taking action a in state s_t and transitioning to state s_{t+1}.

The reward function

As the reward function is the sole goal-leading signal the reinforcement agent gets, it needs to be properly designed. To ease the learning process, a non-sparse reward function is used. With the goal to have all boxes on target positions, moving a box on such a target is rewarded positively. Removing a box from a target position must be punished with a value, so that the reward gotten from moving it onto the target is at least neutralised. Failing to do so, possibly leads to the "cobra effect".[1] As the additional goal is to minimize the solution length, a general step punishment is introduced. Another important property is, that the reward for entering a good terminal state is bigger than the expected discounted reward for continuing the episode. This is given with the general step punishment and the neutralisation of box on-off target movements. Adding an additionally punishment for actions without effect, for example walking against a wall in Sokoban, leads to the following reward function:

$$r(s,a) = \begin{cases} +1.0, & \text{if box pushed on target} \\ -1.0, & \text{if box pushed off target} \\ -1.0, & \text{if nothing changed} \\ -0.1, & \text{otherwise} \end{cases} \tag{14}$$

[1]The term "cobra effect" refers to a historic incident that occurred in British India. To fight the cobra plague the governor offered a reward for every dead cobra. The approach worked quite well, until people started to breed cobras to further capitalize on the reward. Applied to Sokoban, using a bad reward function could incentivize the agent to repeatedly push a box on and off target to maximize the accumulated reward.

Given a classical planning problem, with the goal condition expressed as conjunction of positive literals, the reward function can be generalized to:

$$r(s,a) = \begin{cases} +1.0, & \text{per goal literal made true} \\ -1.0, & \text{per goal literal made false} \\ -1.0, & \text{operator has no effect} \\ -0.1, & \text{otherwise} \end{cases} \tag{15}$$

6.1 DDQN

Deep q-network (DQN, [21]) is a reinforcement learning algorithm, with the goal to learn the q-values of each action in each state using a neuronal network. Q-values are the expected sum of future rewards if the specified action is taken and the optimal policy is followed thereafter. Given a policy π the q-value of an action a in state s is defined as:

$$Q_\pi(s,a) = \mathbb{E}\left[\sum_{n=0}^{N} \gamma^n R(s_n, a_n) | s_0 = s, a_0 = a, \pi\right] \tag{16}$$

Where γ is a specified discount value between 0 and 1. γ takes into account the uncertainty of the future and attaches greater importance to rewards that are closer in time if a value less than 1 is chosen. As the algorithm is applied to a deterministic MDP, having deterministic transitions and finite episodes, the discount variable γ can be set to 1.

DQN learns the $Q(s,a)$ values using temporal difference (TD). Taking an action a_t in state s_t, receiving a reward r_t and transitioning to state s_{t+1}, the q-value is updated following formula (17), omitting the learning rate for simplicity.

$$Q(s_t, a_t) \leftarrow r_t + \gamma \cdot \underset{a_{t+1}}{max}\ Q(s_{t+1}, a_{t+1}) \tag{17}$$

By bootstrapping the maximal action value from the resulting state s_{t+1}, the learned q-values converge to the optimal values.

For use with neuronal networks, two important improvements to the basic q-learning algorithm are introduced.

Algorithm 4 Double q-learning

```
$\theta \leftarrow$ initialize q-network
$\theta_{target} \leftarrow$ copy $\theta$
replay $\leftarrow$ FiFo buffer
**for** iterations **do**
    **for** max exploration steps **do**                         ▷ Collect datapoints
        $s_t, a_t, s_{t+1}, r_t \leftarrow$ execute step using some policy; add to replay
        **for** minibatches **do**                               ▷ Improve q-network
            $s, a, s', r \leftarrow$ sample transitions from replay
            $L(\theta) = (r + \gamma\theta_{target}(s', \underset{a'}{argmax}\ \theta(s', a')) - \theta(s, a))^2$
            $\theta_{target} \leftarrow \tau\theta_{target} + (1 - \tau)\theta$          ▷ Update target network
        **end for**
    **end for**
**end for**
```

One is the application of an additional target network, the algorithm is then called double deep q-network (DDQN). The target network is a lagging copy of the q-network (e.g. achieved with Polyak averaging), where τ defines the networks "distance". The target network is used to estimate the q-value of the best predicted action in state s_{t+1} (as shown in algorithm 4). This solves the moving target problem (updating the estimator using its own estimates) and reduces q-value overestimation, caused by taking the maximum predicted value.
The other important improvement is the use of experience replay. To break the correlation between episode transitions, a replay buffer is introduced, further stabilizing the learning performance. This buffer stores the transition tuples (s_t, a_t, r_t, s_{t+1}) observed by the agent, commonly in a first in - first out (FiFo) manner. These transitions are then uniformly sampled creating mini-batches for network training.

Augmentation While augmentation is common practice for supervised learning, it is not a big topic in reinforcement learning. To add augmentation to q-learning, the authors of the paper "Towards more sample efficiency in reinforcement learning with data augmentation" [6] propose further improvements to the replay buffer. The authors introduce two new techniques, KER (Kaleidoscope Experience Replay) and GER (Goal-augmented Experience Replay), to improve the learning efficiency on the observed transition tuples.

KER exploits domain symmetries. Observing a transition tuple in the Sokoban domain, the state observations s_t and s_{t+1} can be augmented (rotated, reflected and randomly placed as described in imitation learning). By additionally applying the changes correspondingly to the action a_t, the result is again a valid transition on which the q-network can be trained on.

On the other hand, GER, which is not used in this project (but could be), dynamically changes the goal properties of a transition tuple. In Sokoban, the observed states could be changed so that the goal is achieved through a tuples action by moving the goal position of the box and also changing the reward value accordingly.

While the paper proposes KER to take place before adding the observed tuples to the replay buffer, possibly flushing the buffer with many augmented transitions, this project applies the augmentation on batch sampling.

6.1.1 Exploration / exploitation strategies

A key problem in reinforcement learning is the balance of exploration and exploitation. The policy should exploit learned good behaviour but also should not miss out on possibly even better actions to take. This issue is especially hard in the Sokoban domain, as one bad move can unknowingly create a dead-end.
Shown in figure 17 is an exemplary Sokoban problem, in which the agent has to move the box down two times. Pushing the box down once too much (against the lower wall) the given problem can no longer be solved as no pull action is available.

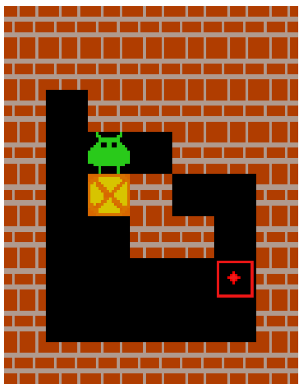


Figure 17: Possible dead-end

As DDQN is an off-policy algorithm a greedy policy can be learned, while executing a different exploration policy. Following, the most common exploration policies are presented and their performance on the Sokoban domain is evaluated.

ϵ-greedy

Commonly used is the ϵ-greedy policy. Given an ϵ value in $[0, 1]$ a random action is chosen with probability ϵ, otherwise the best (maximum predicted q-value) action is executed. Additionally, the ϵ value can be annealed, minimizing the agent's regret over time.

Boltzmann

While ϵ-greedy randomly selects actions, choosing the worst action with the same probability as others, Boltzmann exploration samples the actions from a probability distribution. This distribution is calculated from the predicted q-values using the softmax formula (18).

$$\pi(a|s) = \frac{e^{Q(s,a)/\tau}}{\sum_b e^{Q(s,b)/\tau}} \tag{18}$$

The exploration grade can be controlled by setting the τ parameter. With $\tau \mapsto 0$ the agent always takes the greedy action, while bigger τ flattens the probability distribution, encouraging exploration.

UCB1

The UCB1 (upper confidence bound) exploration strategy counts the number of times an action has been executed in a given state. With $N(s, a)$ denoting the action count in state s, UCB1 selects the next action according to formula (19). Where c is an exploration parameter commonly set to a value of 2.

$$\underset{a}{\operatorname{argmax}}\; Q(s,a) + \sqrt{\frac{c \cdot \ln \sum_b N(s,b)}{N(s,a)}} \tag{19}$$

Having chosen an action, the corresponding action counter $N(s, a)$ is increased by 1. With an increasing count of $N(s, a)$ the exploration pressure vanishes, leading to greedy q-value selection on extensively explored states. As the predicted q-values for a state can be quite different in value, they are normalized to a range of $[0, 1]$ using min-max normalization.

Evaluation

The exploration strategies are evaluated on 6 7×7 problem instances (2 instances for each box count). Each strategy was run 5 times on every problem, using the maximum step-count defined in formula (13). The training on an environment was terminated when the learned q-network was able to solve the problem in the evaluation step (taking the best predicted action in each state), measuring the needed iterations and the evaluation solution length.

The exploration parameters ϵ in ϵ-greedy and τ in Boltzmann exploration were fixed during training, as the problems show a wide range of difficulty levels. Furthermore, a replay buffer of size 50k was used from which mini-batches with 64 non-augmented transitions were sampled.

The q-network was trained using Adam with a learning-rate of 1e-4 and the target network was lagging behind with $\tau = 0.01$.

By adding the FF solution transitions every 10 iterations to the replay buffer (this is possible due to the off-policy property of DDQN), the iterations needed to solve a problem were strongly reduced.

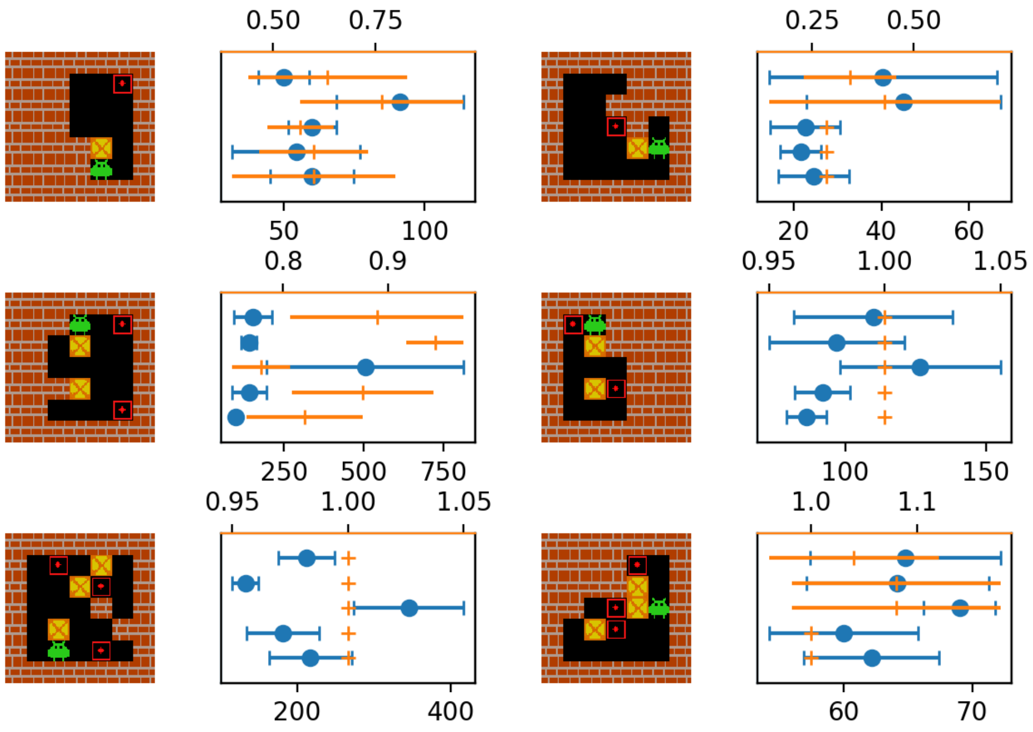


Figure 18: Exploration / exploitation strategy comparison

Figure 18 shows the problem instances with their evaluation results. The exploration strategies from top to bottom are: ϵ-greedy with $\epsilon = 0.5$ and 0.25, Boltzmann with $\tau = 0.5$ and 0.25 and UCB1 with $c = 2$. Marked in orange are the evaluation solution lengths and shown in blue are the needed iterations to solve the problem. The orange vertical lines and the blue dots mark the average values and the horizontal lines show the standard deviation between the 5 evaluation runs. With the solution length values divided by the FF solution length, a value lower than 1 indicates a shorter found solution.

		ϵ / $\tau = 0.5$		ϵ / $\tau = 0.25$	
Algorithm	Boxes	Iterations	Length	Iterations	Length
ϵ-greedy	1	45.1 (20.07)	0.49 (0.22)	68.2 (32.22)	0.6 (0.3)
	2	132.0 (51.49)	0.94 (0.08)	119.0 (33.04)	0.97 (0.03)
	3	138.4 (78.25)	1.02 (0.06)	98.5 (36.64)	1.04 (0.08)
Boltzmann	1	41.3 (20.44)	0.43 (0.15)	38.0 (23.19)	0.44 (0.18)
	2	315.2 (289.29)	0.89 (0.11)	117.7 (46.45)	0.94 (0.08)
	3	207.2 (147.15)	1.04 (0.08)	120.8 (69.54)	1.0 (0.0)

Algorithm	Boxes	Iterations	Length
UCB1 $c = 2$	1	42.3 (21.35)	0.44 (0.21)
	2	92.8 (10.51)	0.91 (0.1)
	3	139.8 (86.37)	1.0 (0.0)

Table 4: Exploration / exploitation strategy comparison

Table 4 shows the exact numbers with their standard deviation in brackets, combining the runs of problems with the same box number.

As all average solution-length values of one- and two-box instances are lower than 1, supplying the replay buffer with sub-optimal FF solution transitions is shown not to have a negative effect on the found solution length.

From the figure and table it can be seen that the ϵ-greedy method is comparably noisy, resulting in longer evaluation pathes. Boltzmann exploration shows more stability when used with a lower τ value. This depends on the used reward function, with a "flat" reward function τ has to be smaller to get good action samples. UCB1 also shows a good performance but is not practical in this case. Since the number of actions are stored explicitly, the multiple repetition of 9k training problems takes up too much memory space.

6.1.2 Training

The DDQN training was started with a pre-trained network, having the same structure as the policy network. This network was trained on q-values generated from FF state solutions, additionally using augmentation. The training dataset was build by step-wise executing the FF solution actions on each environment and collecting the encountered states and their action q-values. These values were calculated by executing each action in a state and using the new FF planners solution to calculate the corresponding action q-value, as shown in formula (16), using $\gamma = 0.98$.

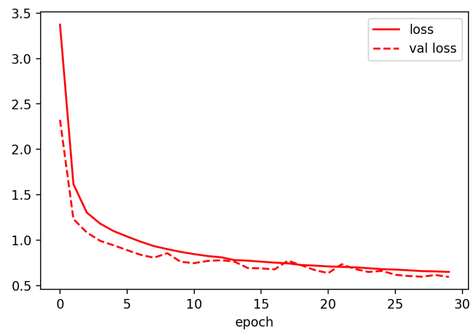


Figure 19: Imitation q-value learning

After training for 30 epochs (shown in figure 19) the network is initially able to solve 10% of the one-box problems, 0.5% of the two-box problems and 0.3% of the three-box problems. A better initial performance could be achieved by using a different planner, as the FF planner shows the unexpected behaviour discussed in section 5.3.1, which distorts the q-values.

Figure 20 shows the training progress, going over all training problems four times and evaluating the networks performance greedily every 200 iterations on the validation problems. During each iteration the network explored an environment 5 times. When the environment was not solved the FF solution transitions were added to the buffer. The maximum step-count for training and evaluation was set as defined in formula (13).

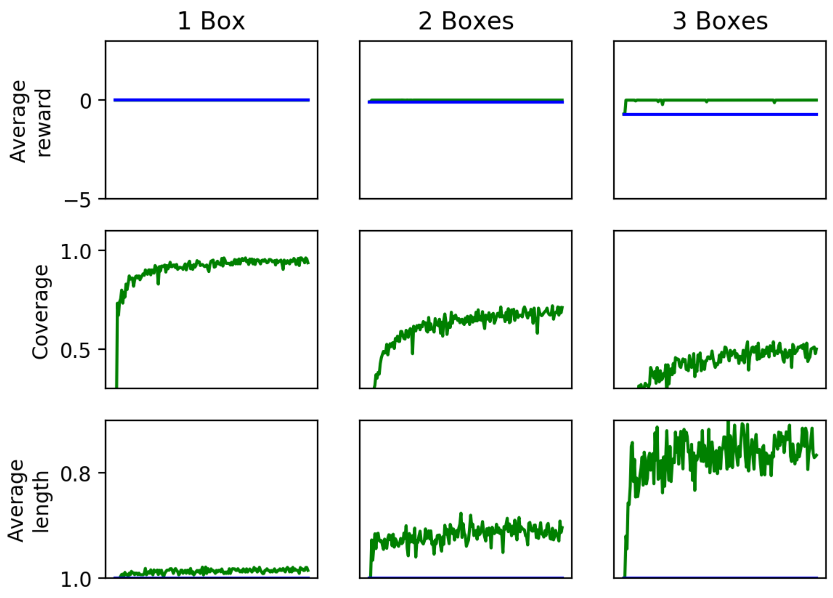


Figure 20: DDQN training performance

As exploration strategy the Boltzmann exploration with $\tau = 0.25$ was used. The training used a replay size of 50k, from which mini-batches with 64 augmented transitions were sampled. As the problems have a finite horizon, using a discount of $\gamma = 1$ would be theoretically possible, but a value of 0.98 improved learning. The network was trained on the mean-squared-error loss with a reduced learning rate of 1e-4 using Adam optimization.

DDQN shows a very stable learning performance, reaching a peak coverage of 96% on one-box problems, 71% on two-box problems and 53% on three-box problems. As those values are reached in different iterations, the final network is again an average. This final network achieves a coverage of 96%, 71% and 54% respectively, with the resulting three-box coverage value again (as with exploration learning) better than any single network. The final network shortens the improved FF solutions by 2% for one-, 9% for two- and 26% for three-box problems on average.

6.2 PPO

In general, there are two main approaches to solve a MDP through reinforcement learning, value methods and policy methods. Value methods try to learn the q-value of actions in a state, the most common algorithm for this is the already presented DQN algorithm. From such learned value functions the final policy can be inferred, such as the argmax policy. Another possibility for solving MDPs, is to learn the policy directly. Current state-of-the-art reinforcement learning algorithm are so called Actor-Critic methods, in which both approaches, value and policy learning, are combined. One such method is the proximal policy optimization (PPO, [14]) algorithm.

Algorithm 5 Proximal policy optimization

```
$\theta \leftarrow$ initialize policy network
$\theta_{old} \leftarrow \theta$                                    ▷ Duplicate policy network
$V \leftarrow$ initialize value network
for iterations do
    for trajectories do
        $s_0, a_0, r_0, ..., s_n, a_n, r_n \leftarrow$ sample n step trajectory following $\pi_\theta$
    end for
    for minibatches do                                 ▷ Improve $\theta$ and $V$
        $L(\theta) = -\hat{\mathbb{E}}_t \left[ min(r_t(\theta)\hat{A}_t, clip(r_t(\theta), 1-\epsilon, 1+\epsilon)\hat{A}_t) + \beta H(\pi_\theta) \right]$
        $L(V) = \hat{\mathbb{E}}_t \left[ (Q_{\pi_\theta}(s_t, a_t) - V(s_t))^2 \right]$
    end for
    $\theta_{old} \leftarrow \theta$                                ▷ Update old policy to current
end for
```

PPO, the pseudo code is shown in algorithm 5, learns a policy by repeatedly sampling trajectories using the current policy.

$$Q_t(s_t, a_t) = r_t + \gamma r_{t+1} + \gamma^2 r_{t+2} + \gamma^3 r_{t+3} + ... + \gamma^n r_n \tag{20}$$

The on-policy q-value for each encountered state-action pair is then calculated in a full-backup manner following formula (20), with a discount value γ between $[0, 1]$.

$$L(\theta) = -\hat{\mathbb{E}}_t \left[min(r_t(\theta)\hat{A}_t, clip(r_t(\theta), 1 - \epsilon, 1 + \epsilon)\hat{A}_t) + \beta H(\pi_\theta) \right] \tag{21}$$

To improve the current policy, PPO then trains the policy network to maximize the expected episode rewards in future trajectories. This is done by minimizing $L(\theta)$ (shown in formula 21), using the following equations.

$$r_t(\theta) = \frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_{old}}(a_t|s_t)} \tag{22}$$

By calculating the probability ratio $r_t(\theta)$, also called importance sampling, as shown in formula (22), the policy can be optimized several times on the same collected data-points. Where $\pi_{\theta_{old}}$ denotes the distribution from which the state-action pair was sampled and π_θ the current policy.

$$\hat{A}_t(s_t, a_t) = Q_t(s_t, a_t) - V(s_t) \tag{23}$$

By additionally using an advantage term (shown in formula 23), instead of the plain on-policy q-values, the policy learning can be stabilized due to variance reduction. With $Q_t(s_t, a_t)$ as the current trajectories q-value and $V(s_t)$ as the value achieved by previous policy iterations from state s_t onwards, the expected advantage of taking action a_t in state s_t and following the policy thereafter is expressed with $\hat{A}_t(s_t, a_t)$. $V(s_t)$ is then updated towards the current $Q_t(s_t, a_t)$ value, minimizing $L(V)$ calculated with the mean-squared-error (MSE) loss function.

$$min(r_t(\theta)\hat{A}_t, clip(r_t(\theta), 1 - \epsilon, 1 + \epsilon)\hat{A}_t) \tag{24}$$

PPO improves the TRPO (Trust region policy optimization, [13]) method by introducing the clipped surrogate objective (shown in formula 24). This addition controls the policy training, clipping large updates to avoid too much deviation from the current policy. The authors of the paper evaluate different ϵ values on 7 robotics tasks, showing that $\epsilon = 0.2$ yields the best results.

$$H(\pi_\theta) = -\pi_\theta(a_t|s_t) \cdot \ln(\pi_\theta(a_t|s_t)) \tag{25}$$

By adding an entropy based regularization term (shown in formula 25) training can be further stabilized, especially when learning a hierarchical behavior as in planning problems. First described in "Function optimization using connectionist reinforcement learning algorithms" [22], the regularizer improves exploration by discouraging premature convergence to a possibly sub-optimal policy. This is achieved by the regularizer rewarding flat action probability distributions and punishing focused distributions.

Training

The PPO algorithm was run starting from the already improved exploration learned policy network, going over all training problems 17 times and evaluating the performance on the validation environments every 500 iterations. After collecting 5 trajectories from 10 different problems, the policy was optimized on 5 mini-batches containing 256 randomly sampled datapoints. The learning rate was set to 1e-4 using stochastic gradient descent and the entropy regularizer was weighted with β = 5e-3.

Instead of letting PPO learn the state value on its own, the already learned q-network was used. As this network predicts each action value in a state, the average of these values was taken as state value, reducing possible overestimation.
Due to PPO's on-policy property, the augmentation was applied once to the current environment before sampling a trajectory.

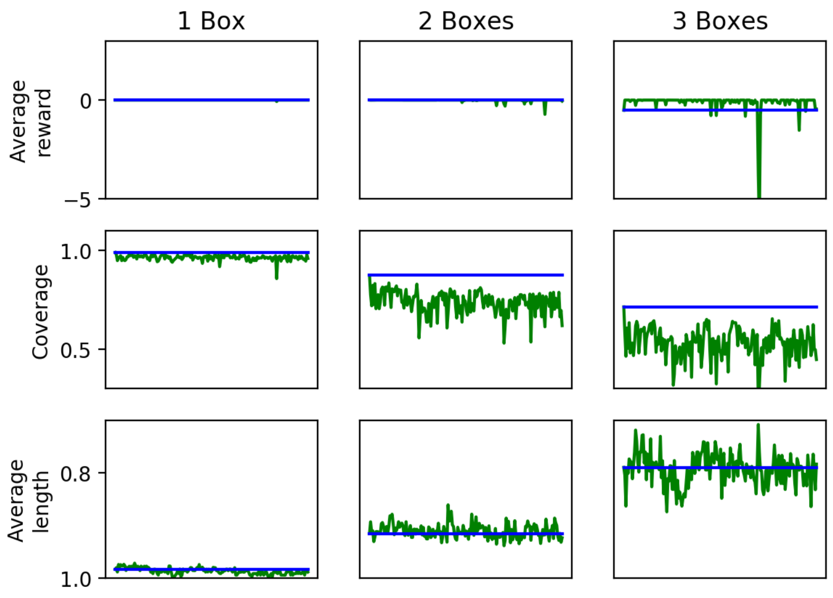


Figure 21: PPO training performance

Figure 21 shows the PPO training performance, starting with the exploration trained network. The PPO training shows no improvement trend after consolidating on lower coverage values. As already observed by the ASNet authors, policy gradient algorithms do not seem to be the best method for behavioral learning in plannable domains.

6.3 Deepmind's AlphaGo

With the huge success of AlphaGo [16] and its successor AlphaGo Zero [17], beating professional Go players and reaching super-human skill, the applications have become very popular.
In the past, two-player zero-sum games like Go (shown in figure 22) were classically solved using Alpha-Beta pruning. Although this approach can theoretically return optimal game-play, it is too slow to explore the full depth of Go.

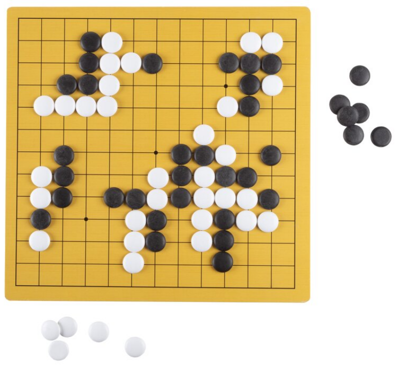


Figure 22: [Source] A 13×13 Go board

AlphaGo uses imitation and reinforcement learning and by combining the learned networks in a novel way with MCTS, the algorithm is able to reach super-human performance. As Go can be seen as a non-deterministic planning problem, the methods used can also be applied to other planning domains.

6.3.1 Board encoding

For network input, the Go board is encoded into a 19 × 19 binary array. In AlphaGo, the board is represented using 3 layers, the players stone, the opponents stone and the empty positions. Later, in AlphaGo Zero, the empty positions are encoded implicitly, so that only 2 layers are needed. As Go is not fully observable, due to the game rule based restriction of repetitive moves, the historic moves have to be supplied to the network. While AlphaGo uses 8 layers to encode the number of turns since a played move, AlphaGo Zero simply concatenates the board encodings of the last 8 board states.

AlphaGo then adds 35 more layers to the input, consisting of binary encoded, game rule based information, like liberties and capture size. Since AlphaGo Zero outperforms AlphaGo without using these handmade features, they prove to be unnecessary.
In both versions, an additional final layer is added, indicating the current player. This is necessary for estimating the state value.

6.3.2 Network

AlphaGo uses a policy and a value network. The policy network consists of 12 convolutional layers using 192 filters with a filter size of 5×5 in the first layer and later 3×3 filters finishing every layer with a ReLU activation. By using the 'same' padding method for the internal hidden layers, sizes are kept at 19×19. The final output layer is again a convolutional layer using 1 filter with a size of 1×1 and a different bias for each position, resulting in an output of 19×19. By finally masking out illegal actions and applying the softmax function, the action probabilities are predicted. The value network has the same convolutional structure, adding a dense layer with 256 nodes and ending with a single output node using tanh activation.

In the later AlphaGo Zero version, the used network is deeper. The network has up to 79 convolutional layers with 256 3×3 filters each, additionally using batch normalization and skip connections. As AlphaGo Zero combines value and policy network into one unit, additional dense layers are used, outputting the value as a scalar from one head and the action probabilities as a flat vector from the other.

6.3.3 Training

AlphaGo first trains the policy network on 30 million game states with their corresponding expert moves from the KGS Go server. The Go boards are additionally augmented in all possible 8 directions using rotation and mirroring.

Algorithm 6 REINFORCE (Monte-Carlo policy gradient) in AlphaGo

```
$\theta \leftarrow$ initialize policy network
for 10k iterations do
    for $n \in [1..128]$ do                                        ▷ Create minibatch
        $s_0, a_0, r_0, ..., s_t, a_t, r_t \leftarrow$ sample full trajectory following $\pi_\theta$
    end for
    $\theta \leftarrow \theta + \alpha \frac{1}{N} \sum_{n=1}^{N} \sum_t \nabla_\theta \log \pi_\theta(a_t|s_t) z_t$      ▷ Apply policy gradient
end for
```

In the second step, the imitation trained policy network is further improved using policy gradient reinforcement learning. For that the REINFORCE algorithm is used, with parameters as shown in the pseudo code 6 were z_t is the result of a trajectory. The algorithm is run with no discount and a sparse reward function, i.e. 0 for all non terminal states, $+1$ for a winning state and -1 for a loosing state (from the view of the current player). To stabilise learning and reduce overfitting, the network is trained by playing against a random iteration of itself.

Using the reinforcement learned policy, the value network is trained to predict the outcome of a state, $+1$ for winning states and -1 for loosing states, minimizing the mean squared error. The training dataset, consisting of 30 million states, is build by sampling each state from a different Go game. This solved the problem of overfitting, which occurred by training on consecutive states.
In detail, a training game-state is generated by executing between 1 to 450 actions sampled from the imitation learned policy and then executing one random legal action. The resulting states value is then calculated by finishing the game using the reinforcement learned policy.

In contrast, the successor AlphaGo Zero ditches the imitation and reinforcement learning approach. Instead, the action probabilities and state values are directly learned from the build-up Monte Carlo tree. This is done by collecting a training dataset containing the trees root action probabilities, from which the next action is drawn, and the resulting game outcome.

6.3.4 Search

Finally the learned networks are used with Monte Carlo tree search. To incorporate the learned policy and value network, novel modifications to the classic MCTS algorithm are made.
Following, only the modified steps are described, omitting the details for distributed computation.

Selection

In addition to the $Q(s,a)$ values and visit counts $N(s,a)$ the nodes also hold the prior probability $P(s,a)$. To incorporate the prior probability into the action selection, the polynomial upper confidence tree algorithm (PUCT, [9]) is used.

$$\underset{a}{\operatorname{argmax}}\ Q(s,a) + c_{puct} P(s,a) \frac{\sqrt{\sum_b N(s,b)}}{1 + N(s,a)} \tag{26}$$

Formula (26) shows the PUCT algorithm, the main difference to UCT (3) is the incorporated prior probability in the uncertainty expression. For c_{puct} AlphaGo and AlphaGo Zero use a value of 5.

The successor, AlphaGo Zero, adds Dirichlet noise to the roots prior probabilities using $\alpha = 0.03$.

$$P'(s_0,a) = (1-\epsilon)P(s_0,a) + \epsilon Dir(\alpha) \tag{27}$$

As the noise itself, represents a legal probability distribution (summing up to 1) it is added with a weighting of $\epsilon = 0.25$ to the prior probability, resulting in a new probability distribution $P'(s_0,a)$, as shown in formula (27).

Expansion / simulation

Expanding a new leaf node, its prior probabilities $P(s,a)$ are initialized. While AlphaGo Zero uses its single network, AlphaGo uses the imitation trained policy network with a softmax temperature of $\tau = 0.67$ (increasing the focus). The authors observed, that using the imitation trained network shows a better performance due to its less focused action distribution.
AlphaGo then does a rollout until game termination, sampling each players action from the policy network. The states value $V(s_l)$ for backpropagation is then calculated as follows:

$$V(s_l) = (1-\lambda)v_\omega(s_l) + \lambda z_l \tag{28}$$

Where $v_\omega(s_l)$ is the state value predicted by the value network and z_l the result of the rollout. AlphaGo equally weights both values, i.e using a λ value of 0.5, as it showed the best performance.

In contrast, AlphaGo Zero does no rollout, only evaluating the leaf-node with its value network using a random rotation/reflection of the Go board.

Action selection

Finally, AlphaGo selects the most visited root action for execution, which showed more stability compared to the max q-action.

$$\pi(a|s_0) = \frac{N(s_0,a)^{\frac{1}{\tau}}}{\sum_b N(s_0,b)^{\frac{1}{\tau}}} \tag{29}$$

AlphaGo Zero, in comparison, uses an exponentially weighted visit count (shown in formula 29) to sample the next action from. During a game τ is set to 1 for the first 30 moves and to an infinitesimal number for the rest of it, leading to the selection of the most visited action.

The tree is kept during the whole game, changing the tree root to the corresponding child state and discarding the other former root children.

7 Putting everything together

In this chapter, the presented search algorithms and the trained networks are combined to a new planning algorithm. Finally this algorithm is tested on the test set, containing same size and bigger size Sokoban problems.

7.1 Evaluating different search and network combinations

Having learned a policy, q-value and heuristic network (in chapter 5 and 6), they are now evaluated using different possible combinations with the best found search algorithms in chapter 3, i.e. A* and UCT* showing the best solo performance. Additionally, ideas from AlphaGo and AlphaGo Zero presented in 6.3 are evaluated.

One obvious application of the policy network, is to use it as rollout policy. Having a guided rollout and a deterministic MDP as basis, allows UCT* to terminate when the rollout reaches a goal state. Returning the in-tree path with the rollout path as a solution, spares the time to build up the whole tree until a leaf node is a goal state. Another basic change with no drawback is the use of prioritised expansion. As UCT* expands all children in a new state, the expansion can be prioritised with regards to the policy action probability. This possibly spares the time to rollout a bad child with no goal or with a worse solution path than a child with high policy probability. Following, using these two modifications, additional changes are evaluated.

As observed earlier (in section 3.3), backpropagating the rollout-value from an unguided rollout, worsened the results of MCTS based algorithms. This raises the question whether this is also the case with a guided rollout. Comparing UCT* using a 20 step policy rollout (taking the best predicted action on the validation problems), with and without rollout-value backpropagation, showed no mentionable difference in percentage of solved problems, average time or solution length.

Having learned a goal-distance heuristic, the needed policy rollout depth can be predicted. This has the advantage that the policy has a higher chance to solve the problem greedily without early rollout termination.

Strategy	Boxes	Coverage	Average time	Average length
Greedy rollout	1	**1.0**	1.22 (2.46)	**1.0 (0.08)**
20 steps	2	0.94	5.05 (24.65)	**1.03 (0.16)**
	3	0.85	9.19 (35.17)	**1.01 (0.21)**
Greedy rollout	1	**1.0**	1.19 (2.45)	**1.0 (0.08)**
Predicted steps	2	**1.0**	**3.41 (7.74)**	1.04 (0.18)
	3	**0.98**	**4.74 (11.11)**	1.07 (0.23)
Greedy rollout	1	**1.0**	**1.12 (0.94)**	1.01 (0.09)
+ 4 sampled rollouts	2	**1.0**	5.53 (11.21)	1.05 (0.16)
Predicted steps	3	**0.98**	8.81 (31.37)	1.07 (0.24)

Table 5: UCT* with policy rollout

Table 5 compares different policy rollout strategies. The experiments were conducted on the validation problem set using UCT* with $c = \sqrt{2}$, a 1k iteration cutoff and no rollout-value backpropagation. The predicted rollout steps were calculated as $\lfloor 1.5 \cdot h(s) \rfloor$ to compensate for the heuristics underestimation. The data shows an improved performance when using a predicted rollout depth on harder problems in terms of coverage and average time, without increasing the average solution length.[1] Additionally running 4 sampled rollouts had a negative effect on the average time and length without improving the coverage.

[1] All length values were divided by the improved FF solution lengths.

As shown, the policy rollout is expensive in terms of time. This has also been observed by the authors of AlphaGo, resulting in the removal of the policy rollout in AlphaGo Zero. Instead, AlphaGo Zero only uses the learned policy with PUCT and evaluates the leafs only using the value network.

Strategy	Boxes	Coverage	Average time	Average length
UCT*	1	**1.0**	1.09 (1.43)	**1.0 (0.08)**
$c_{puct} = 5$	2	0.8	2.37 (2.55)	**1.04 (0.17)**
	3	0.65	3.59 (4.54)	**1.03 (0.18)**
UCT*	1	**1.0**	1.13 (0.71)	**1.0 (0.08)**
Leaf value	2	0.81	2.42 (4.34)	**1.04 (0.18)**
$c_{puct} = 5$	3	0.66	5.14 (7.03)	**1.03 (0.18)**
GreedyUCT*	1	**1.0**	1.12 (0.62)	**1.0 (0.08)**
Leaf value	2	0.95	2.78 (5.7)	1.05 (0.17)
$c_{puct} = 5$	3	0.7	5.07 (6.43)	1.05 (0.2)
GreedyUCT*	1	**1.0**	**0.9 (0.2)**	**1.0 (0.08)**
1 / Leaf heuristic	2	**1.0**	**1.72 (1.5)**	**1.04 (0.17)**
$c_{puct} = 5$	3	**0.97**	**2.48 (3.28)**	1.04 (0.21)

Table 6: UCT* and GreedyUCT* without rollout

Table 6 shows different network combinations, using PUCT with $c_{puct} = 5$ (as in AlphaGo Zero) with UCT* and GreedyUCT* (ignoring in-tree action rewards). The search algorithms were run with a 1k iteration cutoff, trying to solve the validation Sokoban problems. While UCT* with PUCT shows a lower average time compared to the previous rollout measurements, the coverage is way lower. Using the q-value network to predict the leaf-value as $\max_a Q(s, a)$ with UCT* and GreedyUCT* also shows no big performance improvement. The best MCTS based performance was reached by GreedyUCT* using PUCT combined with the inverse heuristic value (due to maximization tree implementation). This approach beats all previous experiments in terms of time, additionally reaching high coverage values and comparable solution length.

Comparing the measurements between GreedyUCT* and A* using the heuristic network (shown in table 3), shows that A* needs less time to solve problems.

As a search algorithm should leverage the greedy solving capability of the policy, different A* modifications employing the policy network are explored.

Strategy	Boxes	Coverage	Average time	Average length
No rollout	1	**1.0**	0.3 (0.23)	**1.0 (0.07)**
$w = 3$	2	0.99	1.0 (1.38)	**1.01 (0.14)**
(Transcribed from tab. 3)	3	0.93	1.74 (2.76)	**0.99 (0.18)**
Greedy root rollout	1	**1.0**	**0.06 (0.17)**	**1.0 (0.08)**
Predicted steps	2	**1.0**	**0.36 (0.99)**	1.04 (0.16)
$w = 3$	3	0.95	**0.84 (2.1)**	1.05 (0.23)
Greedy rollout	1	**1.0**	0.08 (0.55)	**1.0 (0.08)**
Predicted steps	2	**1.0**	1.38 (7.2)	1.04 (0.17)
$w = 3$	3	0.96	3.43 (14.43)	1.06 (0.22)
Greedy rollout to memory	1	**1.0**	0.08 (0.1)	**1.0 (0.08)**
Predicted steps	2	**1.0**	0.37 (1.05)	1.04 (0.17)
$w = 3$	3	**0.97**	0.96 (2.04)	1.05 (0.23)

Table 7: A* with policy and heuristic

Table 7 shows the performance combining weighted A* with a policy rollout. "Greedy root rollout" tried to solve the problems using the greedy policy and falling back to A* search when no solution was found. As a high amount of problems in the validation set can be solved greedily, the average time is noticeable decreased. Instead of only executing a rollout on the root state, "Greedy rollout" runs a policy rollout on all states selected from the queue. While the measured coverage increases slightly, the approach needed more time especially for two and three box problems.

As observed in the previous experiments, policy rollouts are too expensive to be thrown away if the problem is not solved. To capture the encountered states during rollout and giving A* the possibility to continue the search from a rollout state, "Greedy rollout to memory" saves the encountered states to the memory and queue. The greedy exploration is cancelled when the predicted step count is reached or an already known state is encountered. With that modification, the positive aspect from a root rollout is captured, while continuing to leverage the policy during search, when no greedy policy solution is found. The approach shows a better coverage than "Greedy rollout", while needing considerably less time.

Policy correlation evaluation As the final search algorithm (Greedy rollout to memory) only uses the heuristic and policy network, a possible performance improvement when additionally using the inferred policy from the q-network is evaluated.

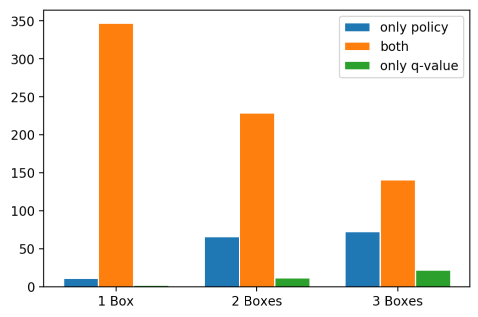


Figure 23: Policy correlation

Figure 23 shows the correlation between the policy and q-value network. Executing both policies greedily on the validation set, the solved problems were separated by their box count. Most problems for each box count were solved by both policies (shown in orange). As expected, more problems were exclusively solved by the policy network (blue) compared to the q-network (green).

Combining both policies equally weighted using softmax with $\tau = 0.1$ for the q-values, showed the same greedy coverage (compared to the plain policy network) and worse average solution length.

As foreseen, using the combined policy with the modified A* search algorithm shows no improvement in coverage or solution length, while worsening the average time.

7.2 The final planning algorithm

The final planning algorithm is a modified weighted A* search algorithm, using the exploration trained policy and heuristic network with $w = 3$. A* is modified so that each selected state from the queue is first simulated using the policy network before it is expanded, adding its children to the memory and queue. This simulation is run for up to $\lfloor 1.5 \cdot h(s) \rfloor$ steps, adding each encountered state with its predicted heuristic value to the memory and queue. When an encountered state is already present in memory the simulation is aborted. After the simulation, A* continues as usual with the expansion.

As a final planner performance test, the developed algorithm is executed on the till now untouched Sokoban test set. This set contains 1k same size problems and 1k bigger problems, as described in section 2.2.1.

First, the generalization performance of the trained policy network is tested.

Problem size	Boxes	Coverage	Average length	Accuracy
Training size	1	0.98	0.91 (0.13)	0.92
(7 - 13)	2	0.86	0.93 (0.14)	0.88
	3	0.71	0.92 (0.16)	0.85
Bigger size	1	0.86	0.88 (0.12)	0.87
(14 - 19)	2	0.68	0.89 (0.17)	0.83
	3	0.55	0.79 (0.21)	0.79

Table 8: Policy generalization to bigger instances

Table 8 shows the greedy policy evaluation results. The test is split between same size and bigger size problems, measuring coverage, average length and accuracy. The coverage of the training sizes is as expected while it drops for bigger sizes, still solving more than half of the problems greedily. The average length (divided by the FF solution length), improves with growing problem size. The accuracy measures the state-action mapping match between the policy and the Fast Forward planner over all FF solution pathes. Important to note is that this measurement might not capture the real performance as some FF solutions are shown to be worse than the policy solutions.

Problem size	Boxes	Average time	Average length
Training size	1	0.09 (0.15)	0.92 (0.19)
(7 - 13)	2	0.45 (1.22)	0.93 (0.2)
	3	1.68 (8.12)	0.92 (0.21)
Bigger size	1	0.61 (2.7)	0.88 (0.24)
(14 - 19)	2	4.17 (15.27)	0.89 (0.2)
	3	6.6 (21.37)	0.81 (0.24)

Table 9: Modified A* search on the test set

Table 9 shows the planner performance on the test set, with the solution lengths divided by the FF solution lengths. The algorithm was run without cutoff, resulting in good run-times and shorter solutions compared to Fast Forward.

8 Conclusion

This thesis explored different policy and heuristic learning methods using deep convolutional networks. By combining the trained networks with A* a new planning algorithm was created. This algorithm showed good solving times and shorter solutions than Fast Forward.

The results in the Sokoban domain suggest that imitation learning a policy for classical planning is the best choice as policy gradients are too noisy. The policy training can be further leveraged by incorporating exploration, which allows the policy to overcome the used planners sub-optimality. Furthermore, the exploration property of MCTS algorithms seems to be unnecessary in a classical planning domain when a good heuristic and policy is available.

While this work focused on Sokoban and thus used a CNN based approach, all presented methods can also be applied to ASNets. The needed heuristic for A* can be learned as additional output. As ASNet uses action modules for output, one could learn a heuristic value for each action (training one module at a time), taking the minimum predicted value as state heuristic.
When learning a policy for probabilistic planning domains, the AlphaGo Zero approach, which learns the policy directly from the Monte Carlo tree, could be promising.

Bibliography

[1] Stefan Falkner, Aaron Klein, and Frank Hutter. BOHB: Robust and efficient hyperparameter optimization at scale. `http://proceedings.mlr.press/v80/falkner18a.html`, 2018.

[2] Edward Groshev, Aviv Tamar, Siddharth Srivastava, and Pieter Abbeel. Learning generalized reactive policies using deep neural networks. `https://arxiv.org/abs/1708.07280v1`, 2017.

[3] Jörg Hoffmann. FF the fast-forward planning system. `http://www.aistudy.com/paper/aaai_journal/AIMag22-03-005.pdf`, 2006.

[4] Steven James, George Konidaris, and Benjamin Rosman. An analysis of monte carlo tree search. `https://aaai.org/ocs/index.php/AAAI/AAAI17/paper/view/14886`, 2017.

[5] Levente Kocsis and Csaba Szepesvári. Bandit based monte-carlo planning. `http://ggp.stanford.edu/readings/uct.pdf`, 2006.

[6] Yijiong Lin, Jiancong Huang, Matthieu Zimmer, Yisheng Guan, Juan Rojas, and Paul Weng. Towards more sample efficiency in reinforcement learning with data augmentation. `https://arxiv.org/abs/1910.09959v3`, 2019.

[7] Thomas M. Moerland, Joost Broekens, Aske Plaat, and Catholijn M. Jonker. Monte carlo tree search for asymmetric trees. `https://arxiv.org/abs/1805.09218v1`, 2018.

[8] Thomas M. Moerland, Joost Broekens, Aske Plaat, and Catholijn M. Jonker. Monte carlo tree search for asymmetric trees. `https://github.com/tmoer/MCTS-T`, 2018, commit 0760426.

[9] Christopher D. Rosin. Multi-armed bandits with episode context. `https://link.springer.com/article/10.1007/s10472-011-9258-6`, 2011.

[10] Stéphane Ross, Geoffrey J. Gordon, and J. Andrew Bagnell. A reduction of imitation learning and structured prediction to no-regret online learning. `https://www.cs.cmu.edu/~sross1/publications/Ross-AIStats11-NoRegret.pdf`, 2011.

[11] Stuart J Russell and Peter Norvig. *Artificial intelligence: a modern approach (3th ed.)*. Boston: Pearson, 2010.

[12] Max-Philipp B. Schrader. gym-sokoban. `https://github.com/mpSchrader/gym-sokoban`, 2018, commit a70efca.

[13] John Schulman, Sergey Levine, Philipp Moritz, Michael I. Jordan, and Pieter Abbeel. Trust region policy optimization. `https://arxiv.org/abs/1502.05477`, 2017.

[14] John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. Proximal policy optimization algorithms. `https://arxiv.org/abs/1707.06347v2`, 2017.

[15] Tim Schulte and Thomas Keller. Balancing exploration and exploitation in classical planning. `http://gki.informatik.uni-freiburg.de/papers/schulte-keller-socs2014.pdf`, 2014.

[16] David Silver and Aja Huang. Mastering the game of go with deep neural networks and tree search. `https://www.nature.com/articles/nature16961`, 2016.

[17] David Silver, Julian Schrittwieser, and Karen Simonyan. Mastering the game of go without human knowledge. `https://www.nature.com/articles/nature24270`, 2017.

[18] Tom Silver and Rohan Chitnis. PDDLGym: Gym environments from pddl problems. `https://github.com/tomsilver/pddlgym`, 2020, commit 838e36e.

[19] Sam Toyer, Felipe Trevizan, Sylvie Thiébaux, and Lexing Xie. Action schema networks: Generalised policies with deep learning. `https://arxiv.org/abs/1709.04271v2`, 2017.

[20] Felipe W. Trevizan and Manuela M. Veloso. Depth-based short-sighted stochastic shortest path problems. `https://felipe.trevizan.org/papers/trevizan14:depth.pdf`, 2014.

[21] Hado van Hasselt, Arthur Guez, and David Silver. Deep reinforcement learning with double q-learning. `https://arxiv.org/abs/1509.06461v3`, 2015.

[22] Ronald J. Williams and Jing Peng. Function optimization using connectionist reinforcement learning algorithms. `http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.54.3433&rep=rep1&type=pdf`, 1991.

[23] Bing Xu, Naiyan Wang, Tianqi Chen, and Mu Li. Empirical evaluation of rectified activations in convolution network. `https://arxiv.org/abs/1505.00853v2`, 2015.

Appendix

Layer	Policy network	Heuristic network
Input	$19 \times 19 \times 4$	$19 \times 19 \times 4$
Convolutional	$29 @ 6 \times 6$	$26 @ 3 \times 3$
	$24 @ 4 \times 4$	$23 @ 6 \times 6$
	$26 @ 8 \times 8$	$20 @ 7 \times 7$
	$16 @ 5 \times 5$	$29 @ 8 \times 8$
	$22 @ 4 \times 4$	max pooling 2×2
	max pooling 2×2	$30 @ 5 \times 5$
	$23 @ 7 \times 7$	max pooling 2×2
		$31 @ 7 \times 7$
		$10 @ 5 \times 5$
Dense	14	
Output	4	1
Activation	Softmax	Linear

Table 10: Convolutional networks