Neural Abstract Tasks for Enhanced Planning in Latent Spaces

Michael Staud[1,2]

[1]StaudSoft UG, Ravensburg, Germany
[2]Ulm University, Institute of Artificial Intelligence, Ulm, Germany

Abstract

This paper introduces Neural Abstract Tasks (NATs), a novel framework that synergizes hierarchical planning techniques with the capabilities of neural networks to enhance problem-solving strategies in visually represented planning domains. By leveraging a projection function that transforms Planning Domain Definition Language (PDDL) domains into comprehensive 2D images, NATs facilitate an intuitive and efficient approach to learning action outcomes and preconditions.

NATs utilize a compressed latent representation of the world state, organized into hierarchical abstractions, to improve planning performance. Our evaluations across a range of domains validate our approach, illustrating that NATs enhance planning performance. The results underscore the potential of NATs to redefine problem-solving in planning.

Keywords

Planning, PDDL, HTN, Neural Networks, Hierarchical Planning, Latent Spaces

1. Introduction

This paper introduces Neural Abstract Tasks (NATs), a novel approach that combines traditional planning techniques with neural networks. They are similar to abstract tasks in Hierarchical Task Network (HTN) planners but are automatically generated. NATs aim to provide efficient and adaptive problem-solving in complex 2D environments by learning effects and precondition satisfaction through examples from a base planning domain. It uses the projection function defined by Staud to create a 2D projection of a PDDL world state [2].

NATs do not operate on atoms; instead, they operate on latent representations [3] of a 2D projection of the world state. This means our algorithm not only creates NATs but also generates an abstract representation of the world state. The NATs use neural networks to apply their effects in the latent representation of the world state.

Our contributions are:

- A new planning framework that uses neural abstract tasks (NATs) and can plan in PDDL domains for which a projection function exists [1].
- A deep learning-based planner that can create a plan consisting of NATs of fixed length. This planner is only used during training.
- A deep learning method for the automatic generation of neural abstract tasks.

CEUR Workshop Proceedings, 2024

michael.staud@uni-ulm.de (M. Staud)

© 2024 Copyright for this paper by its authors. Use permitted under Creative Commons License Attribution 4.0 International (CC BY 4.0).

The following sections describe the NATs, how they are trained, and used in the planning algorithm. Finally, we present the results of our research.

2. Background

We denote constants, variables, and atoms as C, V, and A respectively. A literal encompasses an atom or its negation, with an atom defined as a predicate applied to a tuple of terms. Terms can be either constants or variables and each predicate p is a member of the predicate set P. In this paper, a (classical) domain D_c consists of primitive tasks like in PDDL [2].

2.1. Hierarchical Task Planning Domain

Within a hierarchical task planning domain [4], represented as $D = (T_a, T_p, M)$, we have three distinct finite sets. The sets T_a and T_p consist of abstract and primitive tasks, respectively, whereas M contains methods. Tasks, whether primitive or abstract, are expressed as tuples $t(\overline{\tau}) = \langle prec_t(\overline{\tau}), \mathit{eff}_t(\overline{\tau})\rangle$, featuring a precondition $prec_t(\overline{\tau})$, an effect $\mathit{eff}_t(\overline{\tau})$, and parameters $\overline{\tau}$. The effects, a set of literals, alter atoms within the world state, while the precondition ensures the required atoms are present for the task's execution.

Decomposition methods are tuples $m = \langle t_a(\overline{\tau}_m), P_m\rangle$, with $t_a(\overline{\tau}_m)$ representing the abstract task to be decomposed, P_m the plan steps, and $\overline{\tau}_m$ the parameters. Methods substitute an abstract task with its corresponding plan steps P_m.

A plan is defined as a sequence of primitive tasks (plan steps) arranged in a total order. The domains used in this paper are total ordered. Problems are defined by the tuple $P = \langle init_P, goal_P, PS_P\rangle$, which includes the initial plan PS_P, the initial state $init_P$, and the goal state $goal_P$. The solution is a plan where every task is a primitive task with fulfilled preconditions, transforming the initial state into the goal state. The world state, $w \in W$, consists of a set of atoms and optional a finite number of fluents [5]. When fluents are used they can be altered via functions in task effects.

2.2. Projection Function

Deep Learning typically requires a fixed-size input representation, often a 2D image suitable. This is why we use a function that projects the world state to a fixed-size 2D RGB image (up to certain size of the world state). The use of 2D projections is common in reinforcement learning and algorithm selection [6].

The definition of the projection function $f_p : W \rightarrow I$, mapping the world state w into a 2D image space I, where the image $i \in I$ is a $M \times N \times 3$ matrix (see Figure 2a), is [1]:

- **Consistency:** For two equivalent planning states $w_1, w_2 \in W$, it holds $f_p(w_1) = f_p(w_2)$; if non-equivalent, $f_p(w_1) \neq f_p(w_2)$.
- **Reversibility:** Any image i produced by f_p enables the reconstruction of a world state w that is functionally equivalent for planning purposes, even though specific identifiers like names are not recoverable from i.
- **Pattern Recognition:** The spatial organization within i aids deep learning in pattern learning, maintaining spatial locality and minimizing output complexity.

For other dimensionalities, adaptations are feasible.

3. Neural Abstract Tasks

A NAT is formally defined as a tuple $t_{nat} = \langle prec_{nat}, eff_{nat} \rangle$, where $prec_{nat}$ and eff_{nat} are neural networks representing the preconditions and the effects, respectively.

Note that we treat NATs as grounded abstract tasks when applied by the planner, meaning they are black boxes that can either be applied to a state or not, without any parameters. This necessitates the creation of a diverse set of abstract tasks to ensure a meaningful variety in order to be able to use them to achieve a goal.

4. Generating Training Data

In this section, we discuss the generation of training data, which is then used to train the Neural Abstract Task (NAT) using examples from the classical planning domain D. The quality and diversity of the training data significantly impact the NAT's ability to learn the desired effects and preconditions. The training data is generated from a classical domain and the NATs are specific to that classical domain.

The first step involves preparing a set of classical planning problems. We create a set of n_p planning problems. A problem generator must be provided by the domain designer. Once the planning problems P_p are created, we generate for every problem $p \in P_p$ the tuples (s_i, s_g, p, d) containing a start state s_i and an end state s_g along with their corresponding minimum (planning) distances d. This distance is calculated by running a PDDL planner and counting the resulting actions. The world states are encoded as 2D images.

5. Training Process for Neural Abstract Tasks

In this section, the objective is to learn the NAT's effects and precondition while ensuring valid state transitions. The main training process is depicted in Figure 1.

5.1. Pretraining

We pretrain three networks, used as proxy functions in the main training phase, to facilitate the differentiation required for backpropagation [7, 8]:

- **Step Distance Network (d_{step}):** A convolutional regression network predicting the minimum steps (d) between states s_i and s_g. Architecture: 2 convolutional layers (4 and 8 features), ReLU, two dense layers (64 units), linear activation.
- **Problem Similarity Network (p_{sim}):** A convolutional classifier assessing whether two states s_i, s_g belong to the same problem. Similar architecture to d_{step}, with sigmoid activation in the final layer. Uses binary cross-entropy loss.
- **Local Change Network (δ_{step}):** A convolutional regression network identifying changes between state pairs s_i, s_g by computing the bounding box area of changed pixels. Similar

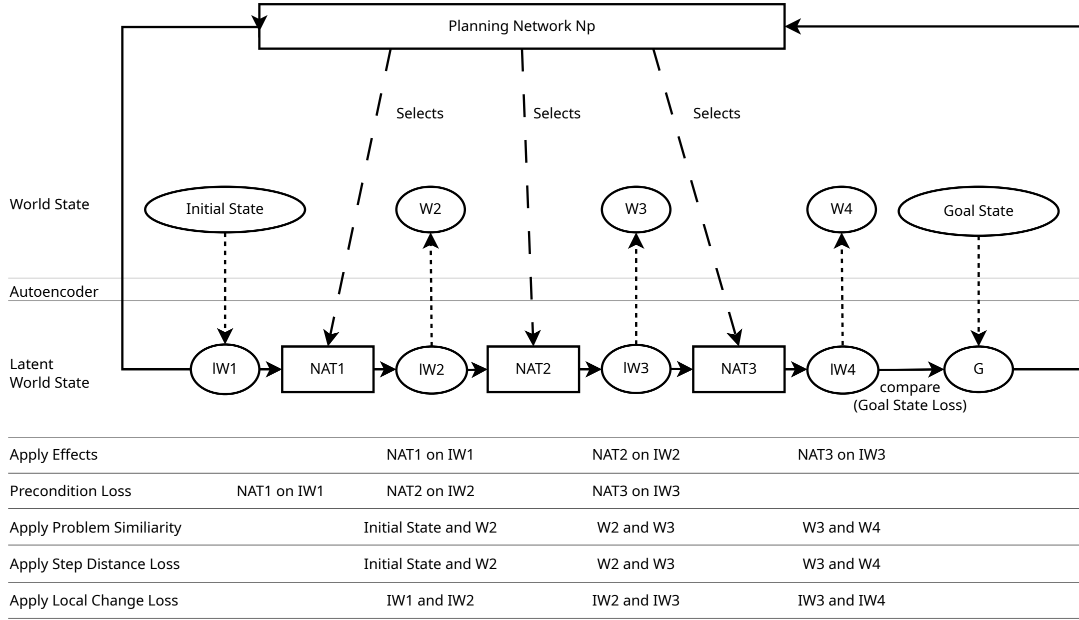


Figure 1: Overview of one step of the training process. First, the planning network N_p creates the plan matrix M_{plan}. Then, the NATs are 'executed' on the world states. IWx denote latent world states, Wx are normal world states (as 2D projections). The output of N_p, the matrix M_{plan} is only used during the training process. Each column of the matrix M_{plan} contains one entry for each NAT. The number of columns is equal to the number of plan steps used during training.

architecture to d_{step}. Trained on image pairs with random pixel alterations to ensure NATs cause localized changes.

We also pretrain a convolutional autoencoder to generate robust latent spaces, starting with 4 features and doubling at each subsequent layer. MSE and SSIM loss is used in combination [9].

5.2. Main Training

In the main training process, we simulate a planning process. Instead of an algorithm, we purely use neural networks and loss functions instead of constraints. The tasks are selected by the neural network N_p (2 conv layers, 4 and 8 features, ReLU, two dense layers) which acts as a planner. All networks will learn their role in the planning process during training, including the preconditions and effects of the NATs. Given a latent space representation of the initial state $init_P = s_i$ and the goal state $goal_P = s_G$, the network N_p outputs a matrix M_{plan}. This matrix contains, for each planning step, a vector (one column) indicating which NAT should be applied to the latent state. Note that the number of planning steps t_{tasks} to reach a goal is fixed, and we filter the dataset to only include state tuples with a minimum number of steps that satisfy $|d - d_t| \leq t_l$. The target number of steps d_t and the tolerance t_l are meta-parameters. The training process is repeated until the network N_p and the neural networks for each NAT have converged.

In the next step, we determine if our plan M_{plan} is indeed a valid plan in the latent space. For

each step, we have a column $m_{plan,i}$. We determine if the chosen NATs $t_{nat} = \langle prec_{nat}, \mathit{eff}_{nat} \rangle$ fulfill their preconditions $prec_{nat}$, which means that $prec_{nat} = 1$ (the precondition network $prec_{nat}$ uses the same architecture as $p_{sim}(s_1, s_2)$). We use binary cross-entropy loss in this case. Note that we usually do not have a column in M_{plan} where only one element is one and all others are zero, especially in the early training process. So, we check the precondition for every NAT and only penalize with the binary cross-entropy loss proportionally to the corresponding value in the column $m_{plan,i}$. We also test if the number of NATs whose preconditions are fulfilled is above a certain threshold given as a meta-parameter. If this is the case, we use MSE loss to penalize this because we do not want every NAT to be executable in a state.

Then, depending on the values in the column $m_{plan,i}$, we apply the effect of the neural networks eff_{nat} to the latent space (eff_{nat} is a U-Net with $(4, 8, 16, 32)$ features and ReLU). This latent space (IW2 in Figure 1, for example) is then transformed back into a 2D representation of the world state (W2 in Figure 1, for example) with the autoencoder. We then apply the Problem Similarity Network $p_{sim}(s_1, s_2)$ to determine if the new world state still belongs to the same problem after applying the effects. The binary cross-entropy loss function is used to enforce this. We also use the Step Distance Network $d_{step}(s_i, s_g)$ on the two non-latent states ($init_P$ and W2 in Figure 1) to determine the minimum number of steps t_{NAT} needed to fulfill this NAT. We then apply an MSE loss to $|t_{NAT} - \frac{d_t}{t_{tasks}}|$ to ensure that every NAT, on average, covers the same number of steps in the final plan. To keep the effect of one NAT on the latent space limited, we also apply a penalty when the area of change in it is too large and not localized. This is measured by using the $\delta_{step}(s_1, s_2)$ function. If this is the last planning step, the non-latent space is also compared to the 2D projection of $goal_P$ using the MSE loss.

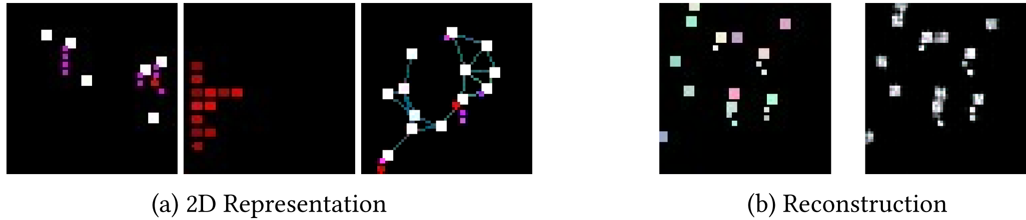
The autoencoder A is used in this process to transform the 2D projection of the initial state $init_P$ and the goal state $goal_P$ into a 2D latent space and back (for example IW1 or G in Figure 1). To ensure that only one NAT is applied at each time step, we apply the entropy function $H(\hat{y}) = -\sum_i \hat{y}_i \log(\hat{y}_i)$ as a loss function to each column of the matrix, as described by Shannon [10]. Additionally, we use the softmax function in the last layer of N_p [11].

Note that all steps presented above form, in fact, one function that is optimized. So, the steps are only presented here sequentially for illustrative purposes. It is also possible to stack on top of the latent representation of the world state another latent representation with additional NATs.

6. The Planning Algorithm

In this section, we describe a simplified version of our planning algorithm that uses Neural Abstract Tasks (NATs) within a Hierarchical Task Network (HTN) framework to solve problems defined in the Planning Domain Definition Language (PDDL). While NATs support multiple layers, for simplicity, we describe a two-layer approach in this paper. The full implementation can support more complex hierarchical planning, which we briefly touch upon later.

Our system takes a PDDL domain as input but uses an HTN planner to solve the problem. The abstract tasks (NATs) are automatically generated from the non-HTN domain and are then executed within the HTN framework. We represent the latent space as fluents within the planning problem. Fluents are variables that change over time, capturing the state of the world



(a) 2D Representation (b) Reconstruction

Figure 2: (a) The 2D representations (RGB images) displayed, arranged from left to right, represent the following domains: Airport, Blocksworld, and Transport. (b) The left image is the original 2D representation of an airport world state, and the right image is the reconstructed version from the autoencoder. This was trained on 10 epochs.

in a compressed form [5]. Each NAT has associated preconditions and effects that transform the latent space.

Each NAT can be decomposed into a method within the HTN framework with an end task. This threshold accounts for the inherent imprecision of neural networks.

1. **Method Decomposition**: A NAT is decomposed via a method that contains an end task. This end task checks if the current latent space matches the expected latent space after the effect eff_{NAT} of the NAT is applied, within a specified threshold. A start task will test for the preconditions of the NAT (with a threshold).

2. **Precondition and Effect Verification**: The method allows the execution of all primitive tasks available in the domain before reaching the end task. Each primitive task updates both the world state and the latent space.

NATs can also be decomposed via another method (the special decomposition method) that allows the arbitrary execution of primitive tasks from the PDDL domain. This ensures the completeness of the planning algorithm, as it effectively reduces the problem to a normal PDDL planner without HTN, capable of selecting any task. A penalty is applied to the plan when such a method is used. This penalty is designed to prioritize the use of methods that have a start and end task, which can be optimized further.

6.1. Optimization

We use the Hierarchical World State Planning (HWSP) algorithm to optimize our planning system [12, 1][1]. The general idea of this algorithm is to divide the world state into different layers that are more and more abstract. In our approach, we have two different world states: the latent representation and the world state consisting of atoms. As these are independent, we can first plan in the latent representation and create a series of plan steps containing NATs and then create the actual plan by using the effects of the NATs as goals. In the context of the HWSP algorithm NATs are separable abstract tasks. Note that splitting a planning problem into

[1]We omitted a full presentation of the planning algorithm instead we describe here only the necessary parts for our research.

smaller ones can increase the performance by an exponential factor [13]. For clarity, we present a simplified version with two layers.

1. **Plan in Latent Space**: The algorithm creates a plan in the latent space using NATs. This abstracts the problem and reduces complexity by working with a compressed representation of the world state.
2. **Generate Concrete Plan via PDDL Planner**: For every plan step containing a NAT, an HTN planner is invoked to decompose the NAT into primitive tasks. This is done in a forward manner. So we have divided the planning problem into smaller problems which can be solved independently. The HWSP algorithm describes this as independent planning processes.
3. **Update Latent Space**: A planner ensures the latent space is updated correctly as primitive tasks are executed, maintaining consistency with the effects of the NATs.
4. **Handling Infeasible Plans**: If a NAT cannot be achieved with primitive tasks, the algorithm replans. Ultimately, if everything fails, the special decomposition methods are used to ensure completeness.
5. **Soundness**: The algorithm is sound because only primitive tasks are selected whose preconditions are fulfilled.

This can be optimized even further. Like the HWSP algorithm, we can use more layers in the world state on top of each other. This would create another latent representation from the original latent representation. Another way is to use the concept of the horizon of the HWSP algorithm. Note that we trained the effects of the NATs to be as small as possible in the latent space via the Local Change Network. This means when invoking the HTN planner to decompose the NATs we do not need to provide them with the full world state. We cannot do this like in the HWSP algorithm by inspecting the domain because of the latent space. Instead, we can determine which pixels were changed in the 2D representation of the world state from the latent representation and estimate the affected atoms. Further research is needed to fully explore and implement this optimization.

Error Propagation Using an autoencoder for latent space generation introduces reconstruction errors, especially with multi-layer architectures:

- **Latent Representation Updates:** We continuously update all latent representations upon the addition of each primitive task to the plan, starting from the initial world state. This prevents the cumulative build-up of errors across planning steps.
- **Training Multi-Layer Autoencoders:** Multi-layer setups increase the risk of error accumulation. We refine our loss function to include both the reconstruction loss of the current latent state and the cumulative loss across all underlying layers (see Table 1).

Latent Size	MSE Loss
4×4	0.0501
8×8	0.0316
16×16	0.016

Table 1
Performance of the autoencoder across different latent sizes in the airport domain. The feature size is always 8. Note that when using multiple autoencoders on top of each other in the multi-layered approach, the total error is equal to the error obtained when using a single autoencoder with the latent space size of the most abstracted autoencoder.

Number of NATs	Precond.	Goal	Entropy
2 NATs (airport)	0.0341	0.252	0.00018
3 NATs (airport)	0.0010	0.083	0.00044
4 NATs (airport)	0.0011	0.011	0.00013
2 NATs (transport)	0.0013	0.0017	0.00022
3 NATs (transport)	0.0005	0.046	0.00032
4 NATs (transport)	0.0018	0.505	0.00057
2 NATs (blocksworld)	0.0011	0.029	0.00014
3 NATs (blocksworld)	0.0038	0.049	0.00025
4 NATs (blocksworld)	0.0033	0.031	0.00032

Table 2
Performance Metrics for Training Neural Abstract Tasks (NATs). This table presents the average loss in preconditions, goal achievement, and entropy in N_p. Training was conducted with a total of $n_{NAT} = 10$ NATs and a primitive tasks amount of $d_t = 10$. It is important to note that the convergence and successful goal attainment are significantly influenced by the number of primitive tasks assigned per NAT, which varies across different domains. A latent space of $16 \times 16 \times 8$ was used.

Test Domain	Ratio	NATs	H0
airport	17.3	102.8	1778.4
blocksworld	19.1	96.8	1848.9
transport	19.8	113.0	2237.4

Table 3
Comparison of Search Space Sizes: This Figure illustrates the ratio of the search space size of a Classical Planner to that of an NATs-based planner. The evaluation was conducted on planning problems with an average of 12 plan steps. The NATs training configuration consisted of a sequence of $t_{tasks} = 2$ plan steps, with a total of $n_{NAT} = 10$ and a number of primitive tasks of $d_t = 10$. A latent space of $16 \times 16 \times 8$ was used. The classical planner was developed internally and uses A* and the H0 heuristic.

7. Related Work

The research in automatic planning has led to different approaches on how to automate the creation of HTN tasks [14]. Hayes and Scassellati explored the autonomous creation of HTNs through their new type of HTN, which they called Clique/Chain HTN (CC-HTN). They created a graphical task representation and derived abstract tasks by analyzing topological properties [15]. This approach only works in small domains, as the generation of abstract tasks uses

NP-hard algorithms. It uses learning from demonstration (LfD) to construct the original task graph. Another approach was explored by Knoblock, which allows the automatic generation of abstractions in planning by dropping literals from the original problem so that the abstractions satisfy the ordered monotonicity property. This is different from HTN planning where this property in general does not hold [16].

Most methods do not create complete abstraction hierarchies. For example, preconditions of HTN decomposition methods can be learned by analyzing the hierarchical plan traces of an expert solver [17]. This can also be expanded with the help of version spaces so that complete methods are learned with no prior information [18]. It is also possible to learn methods from partial observations via the use of a MAX-SAT solver [19]. Hogg et al. explored an algorithm that generates methods of an HTN domain from a classical domain. The input is a series of plans and a collection of annotated tasks [20, 21]. The algorithm works by identifying which portions of the plans can fulfill which annotated tasks. Based on this, methods are created. The quality of this process is strongly dependent on the number of input plans received. The aim of these methods is to help a domain designer in creating a domain.

Chen et al. simplified HTN creation by likening it to finding reductions in a resistor network, a concept from electrical engineering [22]. Nejati et al. tackled HTN domain creation through observation, though struggling with complex task decompositions [23].

Machine learning techniques can be employed to develop abstractions. For instance, Yang et al. applied clustering to demonstrations to generate abstract tasks. Similarly, Neural Task Programming (NTP) leverages execution traces to establish a hierarchical organization of tasks [25]. However, unlike our approach, NTP functions as a model-free method. Additionally, deep learning can be utilized for Action Model Acquisition, which involves generating a planning problem or domain from model-free situations [26].

Similar to the HWSP algorithm [12], the method by Botea et al. constructs an abstract planning problem and generates macro actions for subsequent concrete planning. Unlike HWSP, which treats actions as certain, Botea et al.'s approach considers these actions uncertain and lacks the direct communication of the abstractions [27].

8. Results

In this section, we present the results from our evaluation of the planning algorithm using Neural Abstract Tasks (NATs). We tested the convergence of the precondition and effect networks of the NATs. For this purpose, we conducted the main training step with 2, 3 and 4 NATs. We present the loss in latent space between the goal and the state reached via the NATs. We also present the average loss of the precondition $prec_{NAT}$ networks. To further measure quality, we analyzed how well the deep learning-based planner can select unique actions. A high entropy loss indicates that the system was not able to select only a single NAT in a plan step during training, rendering the NATs not usable when implemented in the planning algorithm. This is presented in Table 2.

During training, 80% of the dataset was used for training and 20% for validation. Each domain's training dataset consisted of 100000 entries. We trained for 10 epochs. We utilized the airport domain [28], the blocksworld domain and the transport domain [29]. Example world

states are presented in Figure 2a. A reconstruction example via autoencoder is presented in Figure 2b.

To measure planning performance, we evaluated the search space size by calculating the average number of actions executable in a given world state. The plan length was then used to estimate the search space size using an exponential function. This process was repeated for planning with and without the use of NATs, allowing us to compare the efficiency of both approaches (see Table 3). The reduction in search space size when using NATs indicates a more efficient planning process, as the planner has to consider fewer potential actions at each step. This efficiency could translate into faster planning times and reduced computational resources required for complex planning tasks.

Based on our findings, automatically generated NATs have significant potential to enhance planning performance. However, there is a drawback compared to the HWSP algorithm. Its separable abstract tasks were previously the sole method for controlling which tasks should be executed in a lower layer, contributing to the algorithm's speed. Since we cannot ensure that NATs encompass the full range of necessary tasks to achieve a goal, it is essential to allow the upper layer to execute tasks directly in the lower layer if NATs are not applicable via the special decomposition method. This approach can potentially impact performance. Another problem which arose in our tests is the debuggability. As the latent space is automatically generated it is not obvious by human inspection which features were correctly compressed in the latent space and what the NATs change in the world state. A loss function, as low as it might get, is not very informative.

9. Conclusions

We introduced Neural Abstract Tasks (NATs), a novel approach to hierarchical planning that combines the strengths of traditional planning techniques and neural networks. NATs enable efficient and adaptive problem-solving in complex 2D environments by learning effects and precondition satisfaction through examples from a base planning domain. The NATs approach can also be expanded to 1D or 3D domains [1].

Further research is necessary to determine how well they perform in more complex domains like the ones used in our tests. Compared to the HWSP algorithm there could be bottlenecks because every planning instance can add primitive tasks to the final plan. This break of the hierarchical world state can decrease its performance.

References

[1] M. Staud, Integrating Deep Learning Techniques into Hierarchical Task Planning for Effect and Heuristic Predictions in 2D Domains, Workshop on Hierarchical Planning (ICAPS) (2023) 19.

[2] M. Fox, D. Long, The Automatic Inference of State Invariants in TIM, Journal of Artificial Intelligence Research 9 (1998) 367–421. doi:10.1613/jair.544.

[3] I. Goodfellow, Y. Bengio, A. Courville, Deep Learning, MIT Press, 2016.

[4] P. Bercher, S. Keen, S. Biundo, Hybrid Planning Heuristics Based on Task Decomposition Graphs, in: SoCS 2014, AAAI Press, 2014, p. 35–43.
[5] M. Fox, D. Long, PDDL 2.1: An Extension to PDDL for Expressing Temporal Planning Domains, Journal of Artificial Intelligence Research (2003).
[6] V. Mnih, K. Kavukcuoglu, D. Silver, A. Graves, I. Antonoglou, D. Wierstra, M. Riedmiller, Playing Atari with Deep Reinforcement Learning, 2013.
[7] Z. He, M. Ciocarlie, MORPH: Design Co-optimization with Reinforcement Learning via a Differentiable Hardware Model Proxy, arXiv preprint arXiv:2309.17227 (2023).
[8] P. H. Winston, Artificial Intelligence (3nd Ed.), Addison-Wesley Longman Publishing Co., Inc., USA, 1992.
[9] Z. Wang, A. C. Bovik, H. R. Sheikh, E. P. Simoncelli, Image Quality Assessment: From Error Visibility to Structural Similarity, IEEE Transactions on Image Processing 13 (2004) 600–612.
[10] C. E. Shannon, A Mathematical Theory of Communication, The Bell System Technical Journal 27 (1948) 379–423.
[11] C. M. Bishop, Pattern Recognition and Machine Learning, Springer-Verlag, Berlin, Heidelberg, 2006.
[12] M. Staud, Urban Modeling via Hierarchical Task Network Planning, Workshop on Hierarchical Planning (ICAPS) (2022) 73.
[13] R. E. Korf, Planning as Search: A Quantitative Approach, AI 87 33 (1987) 65–88.
[14] K. Erol, J. A. Hendler, D. S. Nau, HTN Planning: Complexity and Expressivity, in: B. Hayes-Roth, R. E. Korf (Eds.), Proceedings of the 12th National Conference on Artificial Intelligence, Volume 2, AAAI Press / The MIT Press, 1994, pp. 1123–1128.
[15] B. Hayes, B. Scassellati, Autonomously Constructing Hierarchical Task Networks for Planning and Human-Robot Collaboration, in: 2016 IEEE International Conference on Robotics and Automation (ICRA), IEEE, 2016, pp. 5469–5476.
[16] C. A. Knoblock, Automatically Generating Abstractions for Planning, Artificial Intelligence 68 (1994) 243–302.
[17] O. Ilghami, D. S. Nau, H. Munoz-Avila, D. W. Aha, CaMeL: Learning Method Preconditions for HTN Planning, in: AIPS, 2002, pp. 131–142.
[18] O. Ilghami, D. S. Nau, H. Munoz-Avila, Learning to Do HTN Planning, in: ICAPS, 2006, pp. 390–393.
[19] H. H. Zhuo, D. H. Hu, C. Hogg, Q. Yang, H. Munoz-Avila, Learning HTN Method Preconditions and Action Models from Partial Observations, in: Twenty-First International Joint Conference on Artificial Intelligence, 2009.
[20] C. Hogg, H. Munoz-Avila, U. Kuter, HTN-MAKER: Learning HTNs with Minimal Additional Knowledge Engineering Required, in: AAAI, 2008, pp. 950–956.
[21] C. Hogg, From Task Definitions and Plan Traces to HTN Methods, in: Workshop of International Conference on Automated Planning and Scheduling (ICAPS), 2007.
[22] K. Chen, N. S. Srikanth, D. Kent, H. Ravichandar, S. Chernova, Learning Hierarchical Task Networks with Preferences from Unannotated Demonstrations, in: Conference on Robot Learning, PMLR, 2021, pp. 1572–1581.
[23] N. Nejati, P. Langley, T. Konik, Learning Hierarchical Task Networks by Observation, in: Proceedings of the 23rd International Conference on Machine Learning, 2006, pp. 665–672.

[24] Q. Yang, R. Pan, S. J. Pan, Learning recursive htn-method structures for planning, in: Workshop on AI Planning and Learning (ICAPS), 2007.
[25] D. Xu, S. Nair, Y. Zhu, J. Gao, A. Garg, L. Fei-Fei, S. Savarese, Neural Task Programming: Learning to Generalize Across Hierarchical Tasks, in: 2018 IEEE International Conference on Robotics and Automation (ICRA), IEEE, 2018, pp. 3795–3802.
[26] M. Asai, A. Fukunaga, Classical Planning in Deep Latent Space: Bridging the Subsymbolic-Symbolic Boundary, in: Proceedings of the AAAI Conference on Artificial Intelligence, volume 32, 2018.
[27] A. Botea, M. Müller, J. Schaeffer, Extending PDDL for Hierarchical Planning and Topological Abstraction, in: Workshop on PDDL, International Conference on Planning and Scheduling (ICAPS 03), Trento (Italy), 2003.
[28] A. Anders, Planning Exercises, 2015. URL: https://github.com/arii/planning_exercises, https://github.com/arii/planning_exercises, Accessed: 20.3.2023.
[29] J. Seipp, Á. Torralba, J. Hoffmann, PDDL Generators, 2022. URL: https://github.com/AI-Planning/pddl-generators,Accessed:20.3.2023.