Rheinische
Friedrich-Wilhelms-Universität Bonn

Master Thesis

Robot Motion Planning with Value Iteration Networks on Multiple Levels of Abstraction

Author:
Daniel Schleich

First Examiner:
Prof. Dr. Sven Behnke

Second Examiner:
Prof. Dr. Maren Bennewitz

Date: January 11, 2019

Declaration

I hereby declare that I am the sole author of this thesis and that none other than the specified sources and aids have been used. Passages and figures quoted from other works have been marked with appropriate mention of the source.

Place, Date

Signature

Abstract

While traditional motion planning approaches tend to extensive searches in complex scenes, learning-based methods are promising to speed up the planning process due to an increased scene understanding. Value Iteration Networks (VINs) learn goal-directed behavior and generalize well to unseen domains. However, they do not scale well to larger state spaces. This thesis aims to improve the scalability of VINs by extending them to incorporate multiple levels of abstraction. In particular, only the vicinity of the robot is represented in full detail, while more distant areas are represented in a coarser resolution. Information loss due to coarser representations is compensated by the introduction of additional descriptive features.

We apply our method to 2D grid world planning tasks. Different design choices are evaluated and our approach is compared against VINs and Hierarchical VINs. The results show that our method improves the performance on large 2D grid worlds.

To evaluate the applicability of our approach to real robot motion planning tasks, we extend our method to 3D locomotion planning. We consider a robot that can perform omnidirectional driving and has a certain footprint. Different methods how to adapt our approach to this task are evaluated and the best network architecture is integrated into a planning pipeline for the Centauro robot. Experiments show that our method enables VINs to solve 3D locomotion planning tasks.

Contents

1. Introduction

Robot motion planning is a challenging problem for large or high-dimensional configuration spaces. Traditional planning approaches, such as A* or RRT, become computationally expensive for such queries since they do not understand the scene and tend to extensive searches. Using learning-based planners in such cases is a promising idea. They extract and process relevant information to assess a given scene and derive a suitable action instead of searching through a large amount of states which may result in shorter planning times.

In recent years, many works applied standard CNN architectures to robot motion planning tasks by directly mapping a given scene to actions. However, these networks simply react to their input by using convolutional layers to extract certain features from the input map, and map these features via fully connected layers to action probabilities. Thus, they have difficulties in understanding the goal-directed behavior and generalizing to unseen domains.

Value Iteration Networks (VIN) address this issue by embedding an explicit planning module into a neural network. This planning module is based on Value Iterations and enables VINs to generalize better to unseen environments than standard CNN architectures. However, VINs do not scale well to larger state spaces since the number of required iterations within the planning module depends on the state space size. Training time and memory requirement is increased for larger state spaces. Hence, VINs are only suitable for small or low-dimensional scenarios which does mostly not account to real world applications.

A common approach to deal with large and high-dimensional configuration spaces is abstraction. It has been successfully applied to many different planning approaches, such as planners based on A*-search or reinforcement learning. This thesis aims to improve the performance of VINs to solve more complex problems by extending it to employ an environment representation on multiple levels of abstraction. In particular, a detailed representation of the environment is only used in the vicinity of the robot while more distant areas are described in a more abstract representation. Here, abstraction is achieved by encoding the environment map at a coarser resolution, while each cell is equipped with additional features to compensate the information loss due to the coarser representation.

Describing distant areas of the environments at a more abstract level is an intuitive approach, since detailed information about the environment is usually only available in the vicinity of the robot and less detailed information is provided for more distant areas. Additionally, using neural networks as a planning system may decrease the system runtime by using parallel GPU computations. Furthermore, this thesis investigates, whether VINs can be adapted to be used for real robot motion planning tasks.

After giving a definition of the motion planning problem in Chapter 2 we discus related work, and give a detailed explanation of VINs and the underlying fundamentals in Chapter 3.

In Chapter 4 we propose the new network architecture which combines VINs with multiple environment representations on different abstraction levels and apply it to 2D grid world planning tasks to compare against original VINs.

We offer different solutions to adapt our method to a 3D robot locomotion planning task in Chapter 5. We consider a robot that can perform omnidirectional driving and has a certain footprint, while possible actions are to move to one of the eight adjacent neighbor states with fixed orientation, or to turn to the next discrete orientation with fixed position.

Our proposed architecture and the different design options for applying the method to 2D grid world and 3D robot locomotion planning are evaluated in Chapter 6. Furthermore, we integrate the network architecture which achieves the best performance for the 3D locomotion planning task into a planning pipeline for the Centauro robot.

Finally, Chapter 7 gives a conclusion and points out possible future work.

2. The Motion Planning Problem

In this chapter, we give a definition of the motion planning problem and state simplifications considered in this work. Furthermore, we give a definition of the metrics used to evaluate the performance of the considered planning systems.

2.1. Definition

The general problem of motion planning can be posed as follows: Given the initial robot pose s and the goal pose g, as well as geometric descriptions of the robot and the environment, the planning system should output a sequence of consecutive actions $a_0, \dots, a_T$ which move the robot from s to g on a collision-free path. As there may be multiple solutions, we are interested in those which minimize a given cost function.

In this thesis, we consider a locomotion task with discrete state and action spaces, where the environment information is given as an occupancy map. Furthermore, we do not provide the system explicit information about the start pose but define that the robot is initially located at the center of the map. This is no constraint to the result as different start positions can be realized by shifting the map accordingly. However, this enables us to reduce the resolution of the map dependent on the distance from the robot as described in Section 4.2.

The planning system only outputs the next optimal action. A whole optimal path towards the goal can be generated by iteratively calling the planner, each time updating the input map according to the previously predicted action.

Figure 2.1 shows the general structure of the planning system.

2.2. Evaluation Metrics

To compare the performance of our approach against other learning-based planners, we consider different evaluation measures:

Most important is the *success* rate, which describes whether the planner is able to generate a successful path to the goal. Here, a path is considered successful if it

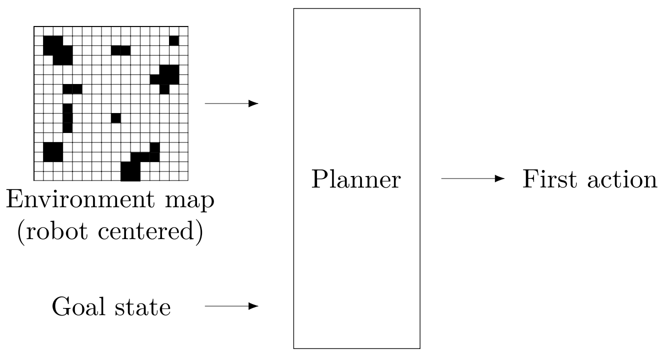


Figure 2.1: Setting of the planning system for a 2D grid world task.

reaches the goal without hitting any obstacles and within no more than twice the number of actions compared to an optimal path, as determined by an A* planner.

We further evaluate the *accuracy* as a measure for the individual decision quality. Since our system is designed to learn form an expert planner, the *accuracy* measures, how often the system chooses the same next action as the expert. Mind that in many cases there is more than one optimal next action. Hence, there occur cases in which the output of the network is different to the output of the expert planner but the network still unrolls an optimal path although the *accuracy* measures a mistake. Nevertheless, we consider the *accuracy* to increase the comparability to other works, which also consider this performance measure.

To evaluate the quality of the generated paths, we consider the difference between its length and the optimal path length. The *path difference* describes how much longer the generated paths are in average compared to optimal paths. To be able to compare between different environment sizes, the *path difference* is taken relative to the path length. Here, we only consider successful paths.

Furthermore, we compare the runtime and graphics memory consumption for training the planning systems.

3. Related Work

In this section, we discuss related planning approaches. We start by giving an overview over traditional planners. Since these tend to extensive searches for large and high-dimensional configuration spaces, we subsequently discuss possible approaches to address this problem, i.e., learning-based planners and abstraction. Value Iteration Networks, which are the starting point for the method proposed in this work, are described in more detail.

3.1. Traditional Planning Approaches

Robot motion planning is often done using search-based or sampling-based approaches. A common method is to discretize the configuration space into a grid and use graph-based search algorithms to obtain an optimal path. The A* algorithm (Hart, Nilsson, and Raphael 1968) is an informed search algorithm commonly used for such tasks. It is an extension of Dijkstra's algorithm (Dijkstra 1959) which uses a heuristic function to expand the most promising nodes first, thus guiding the search towards the goal. While discretizing the configuration space into a fine-resolution grid results in large search spaces and thus in huge memory consumption and runtime, using grids of coarse resolution may result in suboptimal solutions or existing paths may not even be found due to discretization errors. Cell decomposition methods (as in Kambhampati and Davis 1986) and multi-resolution approaches (Behnke 2003) discretize the environment into cells of multiple sizes. This results in smaller search spaces and speeds up the planning process.

Sampling-based methods quickly explore large and complex configuration spaces by connecting randomly sampled configurations to form graphs representing free paths. Probabilistic road maps (PRM) (Kavraki et al. 1996) generate this graph, which is called a road map, by sampling nodes in the configuration space and connect them, if possible, to nearby nodes of the road map. Optimal paths are found by searching this road map, e.g., using the A* algorithm. Rapidly exploring random trees (RRTs) (LaValle 1998) connect the samples using a tree structure. Therefore, no additional graph-based search algorithm is necessary since paths can directly be extracted from the tree. In contrast to PRMs, RRTs do not require to completely connect already existing nodes to the sampled configuration. Instead, they only grow the tree from the nearest node slightly towards the sample. Thus, RRTs can be directly applied to nonholonomic and kinodynamic planning tasks, while finding a complete connection between to configurations, as required for PRMs, is challenging for such tasks. However, the paths obtained by RRTs usually are not optimal. Karaman et al. introduce RRT* (Karaman and Frazzoli 2011) which addresses this problem by locally rebuilding the tree every time new nodes are inserted. Thus, optimal paths can be extracted.

Potential Field Methods (Khatib 1985) model the current robot configuration as a particle which is influenced by a potential field. This potential field is a combination of forces which attract the particle towards the goal and forces repulsing it from obstacles. Planning can be done by performing gradient descent. However, this may lead to sub-optimal local minima.

3.2. Learning-based Planners

Traditional planning approaches tend to extensive searches due to a lack of scene understanding. One approach to increase the scalability is to increase this scene understanding. Convolutional neural networks (CNN) have shown promising results for tasks such as image classification (Krizhevsky, Sutskever, and Hinton 2012) and robot perception (Schwarz et al. 2018). In recent years, CNNs have also been applied to motion planning tasks. They have been used to directly derive actions from state observations. In Bojarski et al. 2016, a CNN is trained to map raw images to steering commands for an autonomous driving car.

Reinforcement learning (Sutton and Barto 1998) offers frameworks to handle inaccuracies in motion execution and uncertainties in the environment. Furthermore, it enables robots to learn complex motions which are difficult to define manually, i.e., learning robust controllers for an autonomous helicopter (Bagnell and Schneider 2001). Reinforcement learning methods aim at generating a policy, i.e., a mapping from the current state to the next action, which achieves a high long-term reward. Similar to search-based and sampling-based planning approaches, reinforcement learning methods suffer from the curse of dimensionality and do not scale well to larger state spaces and higher-dimensional problems. Therefore, CNNs have also been used in the context of reinforcement learning. Deep Q-networks (DQN) (Mnih et al. 2015) directly map state observations to the expected long-term rewards for each action and thus learn to play Atari games from visual input. In Levine et al. 2016, the policy is represented by a CNN, which is trained to map raw images to robot motor torques for real-world manipulation tasks.

Although the results of these applications are impressive, such approaches simply react to their input by using convolutional layers to extract certain features from the input map and map these features via fully connected layers to action probabilities. Thus, they lack the capability of including long-term goal directed behavior and generalization to unseen domains. To address this problem, explicit planning modules can be embedded into the network.

Universal Planning Networks (UPN) (Srinivas et al. 2018) learn useful latent state representations from images of the current scene and the desired goal scene. They infer motion trajectories by performing gradient descent planning and iterating over action sequences in the learned internal representations. UPNs are capable of solving robot motion planning tasks, even with more than two dimensions. However, considered environments are rather small since the gradient descent planner, which is embedded in the forward pass of the network, is time consuming. This causes long training times and hinders scaling to larger environments. Impressively, UPNs are able to generalize to modified robot morphologies.

Tamar et al. propose Value Iteration Networks (VIN) (Tamar et al. 2016) which embed an explicit planning module based on the Value Iteration algorithm into the network. In the supplementary material of their work, Tamar et al. propose Hierarchical Value Iteration Networks (HVIN) which perform Value Iteration on different resolution levels to increase the performance on larger map sizes. Since VINs are the starting point of the network architecture discussed in this work, we will give a more detailed description of the underlying fundamentals and of VINs and HVINs in Section 3.2.1 to 3.2.3.

VINs have also been applied in other domains. In Niu et al. 2017, Generalized Value Iteration Networks are proposed, which work on arbitrary irregular graph structures and can be applied to real world data like street maps. Gupta et al. propose a Cognitive Mapper and Planner (CMP) (Gupta et al. 2017) to plan actions from first person views. They combine a neural network, which processes first person images to build up a latent representation map of the environment, with an hierarchical planning module based on VINs, which plans on multiple spatial scales. Khan et al. propose Memory Augmented Control Networks (Khan et al. 2017) which apply VINs to partially observable environments. VINs are applied to local observations to generate local policies. A global controller network learns a history representation that is stored in external memory and combined with the local policies to predict actions. QMDP-nets (Karkus, Hsu, and W. S. Lee 2017) also handle partially observable environments. Similar to VINs, they express value iteration through a CNN. Additionally, they combine it with a second neural network which implements a Bayesian filter to update the current belief of the agent state.

3.2.1. Markov Decision Processes and Value Iteration

In this section we describe the underlying fundamentals for VINs. Markov Decision Processes (MDPs) are standard models for decision making and planning in stochastic environments (Bellman 2013 [1957]). An MDP is a tuple $(\mathcal{S}, \mathcal{A}, \mathcal{R}, \mathcal{P}, \gamma)$, consisting of the following components:

- A set of states $\mathcal{S}$,
- a set of actions $\mathcal{A}$,
- a reward function $\mathcal{R} : \mathcal{S} \times \mathcal{A} \to \mathbb{R}$ which assigns each state-action pair a real-valued reward,
- transition probabilities $\mathcal{P}$, where $\mathcal{P}(s'|s,a)$ denotes the probability of the next state s' given the current state s and action a, and
- a discount parameter $\gamma \in [0,1]$.

The goal is to find a policy π, i.e., a mapping from current state to next action, which results in high long-term rewards. One common algorithm to find such an optimal policy is Value Iteration (VI) (Bellman 1957). For each state s a state value $V^\pi(s)$ is calculated which denotes the expected long-term reward when starting in state s and following the policy π:

$$V^\pi(s) := \mathbb{E}^\pi\left[\sum_{t=0}^{\infty} \gamma^t \mathcal{R}(s_t, a_t) | s_0 = s\right]. \tag{3.1}$$

Here, the use of the discount parameter γ enables to weight immediate obtained rewards stronger than future rewards. The VI algorithm approximates the optimal value function V^{π^*}, i.e., the value function corresponding to the policy π^* which results in the highest possible expected long-term rewards:

$$V^{\pi^*}(s) := \max_\pi V^\pi(s). \tag{3.2}$$

This is done by iteratively applying the Bellman equation: Regarding the optimal policy, the expected long-term reward for a state equals the expected long-term reward for taking the optimal action in this state. Following this idea, an arbitrary initial policy π is chosen and two steps are alternated: First, each action is evaluated by doing a one-step lookahead to compute the expected long-term reward Q when starting in state s with action a and afterwards following the policy π, which is

$$Q(s,a) = \mathcal{R}(s,a) + \gamma \sum_{s'} \mathcal{P}(s'|s,a) V^{\pi}(s'). \tag{3.3}$$

Here, the sum runs over all possible states s' to which the application of action a in state s may lead. Second, following the idea of the Bellman equation, the expected reward for the state s is set to the expected reward of the best action for this state, i.e.,

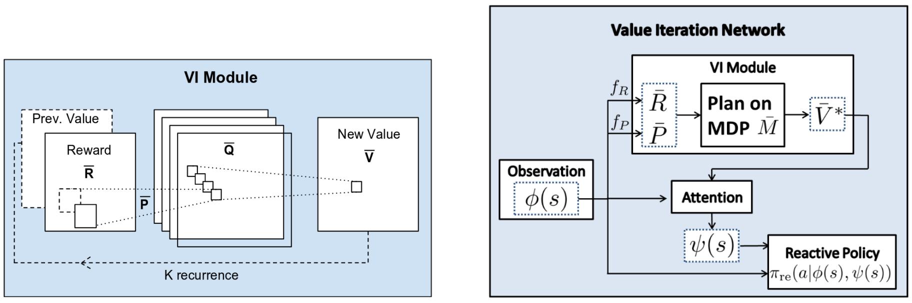
$$V^{\pi}(s) = \max_{a \in \mathcal{A}} Q(s,a). \tag{3.4}$$

In the following, we will refer to the application of Equation (3.3) followed by Equation (3.4) as one Bellman update. It is known that the state-value function V and the action-value function Q converge to the optimal state-value and action-value functions V^{π^*} and Q^* after several Bellman updates. The optimal policy π^* can then be obtained by always moving to the state with the highest state value, i.e., choosing

$$\pi^*(s) = \arg\max_{a \in \mathcal{A}} Q^*(s,a). \tag{3.5}$$

3.2.2. Value Iteration Networks

Value Iteration Networks (Tamar et al. 2016) mimic the VI algorithm by rewriting it as a CNN. The neighborship relation of the state space $\mathcal{S}$ is assumed to follow a grid structure. Hence, the state-value function V can be expressed as a map. Instead of generating rewards that depend on state and action, we compute rewards that are obtained for entering a state independent of the performed action. Therefore, we have one reward per state and can also write the reward function as a map $\mathcal{R}$. If we assume the transition probabilities $\mathcal{P}$ to be spatially invariant, we can regard Equation (3.3) as a convolution over the old state-value estimates V with the convolution kernel $\gamma\mathcal{P}$. This results in the action-values Q where we have one channel per action. Updating the state-value estimates according to Equation (3.4) then corresponds to a max pooling operation over the action channel. These two steps are iterated several times. Figure 3.1 (a) shows the VI Module, which is the VI algorithm depicted as a CNN: Reward and state-value images are stacked and fed through a convolutional layer resulting in action-values Q, from which updated state-values are obtained via a max pooling operation over the action channel. These are stacked with the reward image and the whole process is iterated several times to obtain an approximation of the optimal state-values.



(a) VI Module

(b) VIN structure

Figure 3.1: Value Iteration Networks. Left: The VI Module expresses the VI algorithm as a convolution and max pooling operation. Right: The complete VIN architecture. From Tamar et al. 2016.

Expressing the VI algorithm as a CNN has the advantage that the whole operation is fully differentiable. Hence, it can be embedded into a neural network architecture which can be trained end-to-end using standard backpropagation. This enables us to learn the parameters of the VI algorithm, such as reward function and transition probabilities.

Figure 3.1 (b) shows how the VI Module is embedded into a neural network resulting in the complete VIN architecture: Input is an observation of the environment which is fed into a neural network to obtain the reward map $\mathcal{R}$. This is used within the VI Module to generate approximate state-values V. The transition probabilities correspond to the convolution kernels of the VI Module. Subsequently, an attention mechanism is used to extract the state-values for the relevant states, e.g., the neighboring states of the start state. These relevant state-values are finally passed through another neural network which maps them to action probabilities.

VINs achieve good results for planning motion trajectories for simple environments of limited size like for the 2D grid navigation task presented in the original VIN paper (Tamar et al. 2016). Although VINs have been applied to several problems such as continuous control tasks or web page navigation, we only apply VINs to discrete grid navigation tasks in this thesis. The experiments of Tamar et al. show that, due to the embedded planning operation, VINs generalize well to unseen environments. However, they have only been evaluated on grid sizes up to 28×28 cells. For real robot motion planning tasks, the planning system usually has to deal with significantly larger and higher-dimensional state spaces. In order to successfully predict the next action, the reward information has to be conveyed from the goal to the start state. If the shortest motion trajectory between start and goal state consists of k actions, at least k Bellman updates are needed since

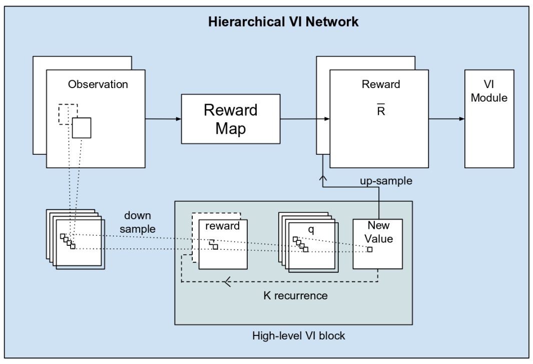


Figure 3.2: Hierarchical Value Iteration Networks with two levels. From the supplementary material of Tamar et al. 2016.

each update only passes reward information from one state to its next neighbors. For large and high-dimensional grids, this leads to large computation graphs for the gradients during backpropagation, resulting in long training times and high memory consumption. Therefore, the number of states within the VI module is limited.

3.2.3. Hierarchical Value Iteration Networks

To reduce the number of necessary Bellman updates, Tamar et al. propose Hierarchical Value Iteration Networks (HVIN) in the supplementary material of Tamar et al. 2016. Value iteration is first performed on a down-sampled copy of the input map to generate rough state-value estimates, which are up-sampled and used as initialization for another value iteration module working on the full resolution. Mind that only the state-value map but not the reward map is up-sampled. To be able to successfully avoid collisions, a different reward map for each resolution level is generated. For the highest resolution level, the used reward map is learned from the full resolution input map. Figure 3.2 shows the structure of HVINs for two levels, though this model can be extended to multiple hierarchical levels. The lowest resolution level uses the same number of Bellman updates as proposed for original VINs with a similar number of cells. Since no details for the number of Bellman updates for the higher resolution levels are given by Tamar et al. , we decided to choose it equal to the down-sampling factor. Thus, we ensure that the reward information from all neighbors of a lower level cell is conveyed to all its refined cells.

Especially for larger grid sizes, HVIN need significantly less Bellman updates compared to VINs. However, down-sampling the input image results in information loss. In contrast to the method proposed in this thesis, this is not compensated. Furthermore, all levels operate on the whole environment size resulting in only slightly decreasing memory requirements.

3.3. Abstraction

For complex and high-dimensional tasks, traditional planning approaches tend to extensive searches resulting in huge memory consumption and runtime. One possible method to increase the scalability of such approaches is to reduce the state space size. Abstraction is an established method to achieve this, while keeping the resulting information loss small. Abstract states unify multiple detailed states. This can be realized through coarser resolutions or lower-dimensional representations. In contrast to multi-resolution approaches, the loss of information is compensated by additional features which increase the representation's semantics. In Klamt and Behnke 2018 the search-based approach for the high-dimensional problem of hybrid driving-stepping locomotion planning (Klamt and Behnke 2017) is extended to plan on multiple levels of abstraction which results in significantly shorter planning times while the result quality stays comparable.

Abstraction is also used in the field of reinforcement learning. In Kulkarni et al. 2016, temporal abstraction is used to generate an efficient space to explore complicated environments.

4. Value Iteration Networks on Multiple Levels of Abstraction

In this chapter, we describe the general method of extending VINs to incorporate multiple levels of abstraction. We introduce the chosen network architecture and motivate the design decisions for applying the method to 2D grid world planning tasks.

4.1. Choice of Software Architecture

To decide which learning-based planning approach to use as a starting point for the method proposed in this thesis, we implemented VINs and Universal Planning Networks (UPNs) for 2D grid world path planning tasks. For this task, we do not consider the robot orientation or the footprint. Thus, a robot pose corresponds to a single grid cell.

We adapted VINs such that the start pose always is located at the center of the map. As in the original VIN paper, information about the goal pose is given by a one-hot goal map. Additionally, we employed VINs with three levels of abstraction consisting of one, two and six features. We stacked the three abstraction maps and corresponding goal maps and applied the VIN architecture to the resulting map. However, within the VI Module, the max pooling operation over the action channel was performed independently for each abstraction map.

We trained VIN and their extension to three abstraction levels on 35000 different training scenes which were generated by placing obstacles of random number, size and position into grid worlds of size 32×32. The test set consisted of 5000 different scenes. This experiment showed that VINs are capable of solving the tasks we address in this work and that they can profit from using multiple abstract environment representations.

UPNs are designed to learn from environment images and to solve continuous control tasks. Input for the original implementation are RGB-images of the initial scene and the goal scene, with a resolution of 84×84 each. For each, initial and goal scene, we used the occupancy map stacked with a one-hot map where the robot position is marked, and up-sampled them to a resolution of 64×64. We applied UPNs to the same task as VINs but they did not learn successful paths for our generated input data. Therefore, we also tested UPNs on grids of size 8×8, which is similar to the grid world task of the original UPN paper (Srinivas et al. 2018). However, they did not learn successful paths for this task, too. There are two possible reasons:

First, during training, the provided expert actions describe movements to one of the eight adjacent grid cells. Since UPNs are designed to solve continuous control tasks, it it possible that these movements are too large to be correctly predicted by our UPN implementation.

Second, the runtime per training epoch of UPNs is significantly higher compared to the runtime of VINs. This is due to the iterated gradient computation within the forward pass of UPNs. Furthermore, the structure of UPNs is more complex resulting in an increased number of trainable parameters. Therefore, a larger amount of training data and more training iterations may be necessary. It is possible that UPNs generate better results for significantly increased amounts of training data and training time.

However, as VINs showed promising results and outperform UPNs with respect to result quality, runtime and data efficiency, we decided to use VINs as a starting point for our work.

4.2. Multiple Levels of Abstraction

As already stated in Section 3.2.2, VINs do not scale well to larger state space sizes due to the increased number of necessary Bellman updates. To be able to plan for larger environment sizes while keeping the number of states small, we introduce additional, abstract environment representations. In the vicinity of the robot, the planner needs precise environment information at a high spatial resolution to avoid collisions when planning the next robot action. For more distant areas, less detailed information is sufficient: Instead of knowing the precise obstacle position within these distant areas, we only need to know whether we can traverse the area in a certain direction. Thus, we encode the environment information for more distant areas at a coarser and more abstract resolution.

We define three levels of abstraction with a constant number of cells but decreasing resolution. *Level-1* has the original input resolution but only covers the vicinity of the robot. For *Level-2*, the resolution is halved resulting in a four times larger covered area. This step is repeated to obtain *Level-3*. Hence, *Level-3* covers an area which is 16 times larger than the *Level-1* area. The spatial arrangement

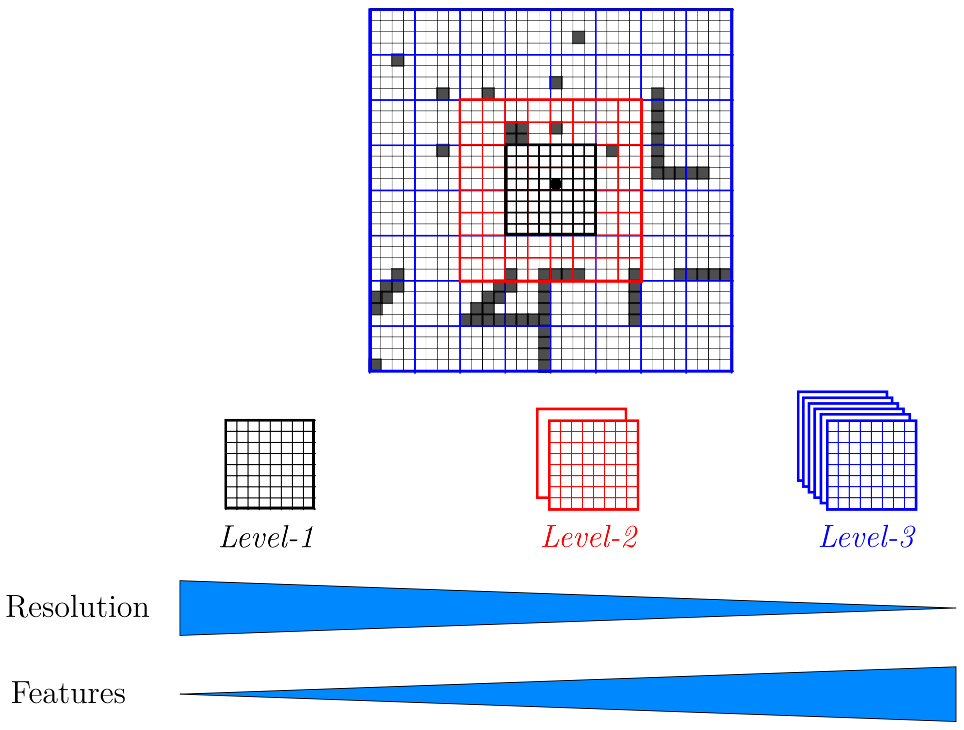


Figure 4.1: The areas covered by the different abstraction maps.

of the three representations is depicted in Figure 4.1.

In contrast to HVINs, the information loss due to coarser representations in higher abstraction levels is compensated by introducing additional features for each abstract cell. While each *Level-1* cell only carries the information whether the cell is occupied or not, we could encode additional features for each *Level-2* cell, describing the possibility to traverse the cell in different directions and even more features for traversing possibilities of each *Level-3* cell. However, instead of manually defining such features, we let the network learn them. Each cell of *Level-2* and *Level-3* can be represented at full detail using four and sixteen features, respectively. However, we are interested in coding the cells in a more compact way. Thus, we define the number of features for each cell of *Level-1*, *Level-2*, and *Level-3* to be one, two, and six, respectively.

4.3. Network Architecture

In this section, we apply our method to plan shortest paths for a point-like agent in 2D grid worlds. As actions, the agent can move to one of the eight adjacent neighbor cells. This planning task is similar to the one from the original VIN paper (Tamar et al. 2016) and is used to evaluate our approach against VINs and

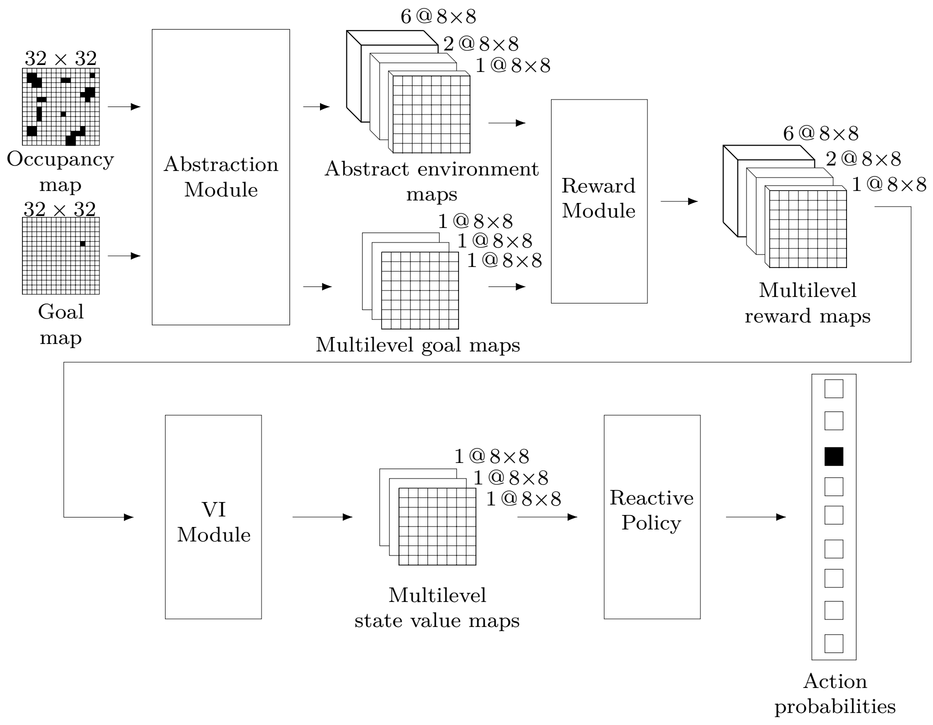


Figure 4.2: Network architecture for 2D grid world planning.

HVINs in Chapter 6.

Input to the network is an occupancy map of the environment and an equally sized one-hot goal map, which only contains zeros except for the goal cell. In contrast to original VINs, we do not provide the system explicit information about the start pose, but define that input maps are always robot centered. Thus, we are enabled to keep the spatial arrangement of the different abstraction levels constant. The network structure is depicted in Figure 4.2. In the following, we give a detailed explanation of the different modules.

4.3.1. Abstraction Module

In a first step, the Abstraction Module processes the input environment map to three, equally sized abstract environment maps. The *Level-1* map is extracted as a patch around the center of the occupancy map. A convolution generates the *Level-2* representation with halved resolution from the input map. While the *Level-2* map is again extracted from the map center, the whole *Level-2* representation is processed by another convolution to obtain the *Level-3* map. The goal map is processed similarly. However, we do not use convolutions but max pooling operations with the same kernel size and stride. For each abstraction level, we thus obtain a one-hot goal map with one channel, which has the same resolution

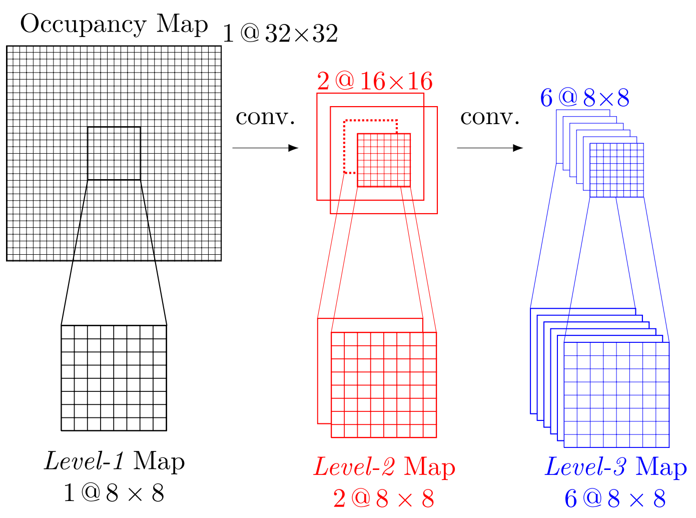


Figure 4.3: Abstraction Module. Both convolutions use kernels of size 2×2 with a stride of 2. The goal map is processed using max pooling operations with the same parameters instead of the convolutions.

and covers the same area as the corresponding abstraction map. The Abstraction Module is depicted in Figure 4.3.

4.3.2. Reward Module

Subsequently, the abstract environment maps and the multilevel goal maps are fed into the Reward Module (Figure 4.4) to generate rewards for each state. Since the transition probabilities learned by VINs are spatially invariant, the information about valid actions for a given state cannot be represented by these transition probabilities. Instead, VINs overcome this issue by learning suitable rewards which punish the choice of invalid actions, i.e., moving to occupied cells, and therefore, discourages the network to choose those actions. However, this relies on the fact that the validity of an action only depends on the target state. For the cells within our higher-level abstraction maps, this is not true. Since we have multiple features per abstract cell, it might, e.g., be possible that an abstract map cell can be entered from one direction but not from another which is encoded in the features. Same applies to the action start cell which may only allow actions to certain directions. Hence, the validity of an action depends on both, target and start state. Following the idea of punishing invalid actions, we need rewards that consider the multiple features of the start and target cell of an action. We model this by extending the reward map to the number of features of the corresponding abstraction map.

There are two different options how to process the abstraction maps to rewards: Processing each abstraction level independently or allowing information flow be-

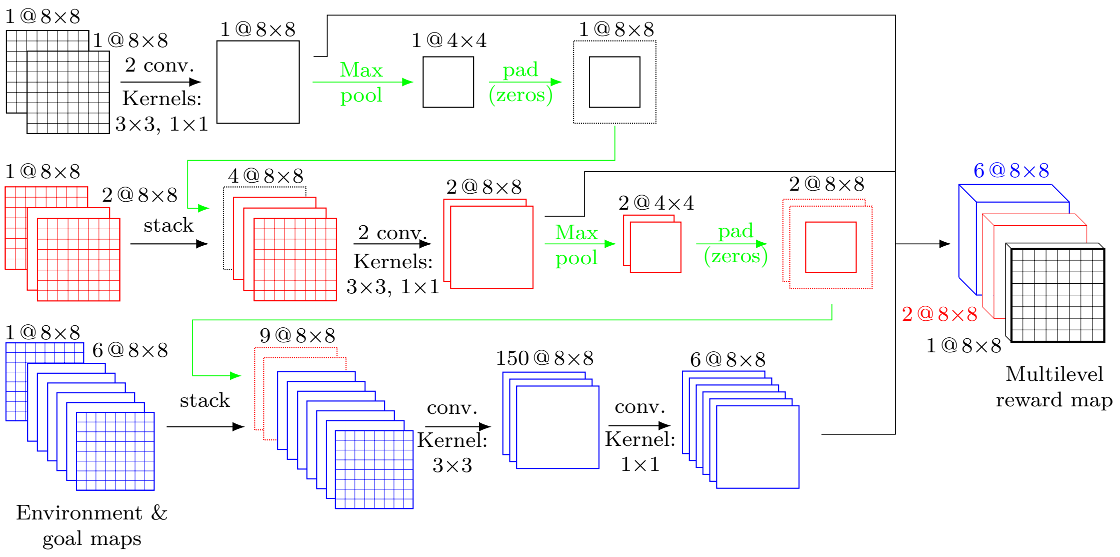


Figure 4.4: Reward Module. The green parts are used to enable information flow between the different abstraction levels. They are omitted for the independent processing variant. *Level-1* maps are shown in black, red parts belong to *Level-2* and blue parts to *Level-3*. Wherever two convolutions are depicted in one step, the first maps to 150 features and the second to the depicted number of channels, as shown for *Level-3*.

tween the different levels. Both approaches are described in the following and they are compared against each other in Section 6.1.1.

We first consider the **independent input processing** approach. For each abstraction level, we stack the respective environment and goal map and process them with two convolutions resulting in the reward map of the respective abstraction level. The advantage of this method is that the depth, i.e., the number of layers, for the reward module is kept small. This may facilitate the training process since the error information only has to be backpropagated through two layers.

However, not all available information is used to generate the reward maps when processing each abstraction level independently. The area covered by the center of higher abstraction maps is described in more detail by the lower abstraction maps. Thus, by enabling **information flow between the abstraction levels**, we can use the more detailed representation from lower abstraction levels when generating the reward maps for higher levels.

It is important to understand that information encoded at the same cell of different abstraction maps refers to different locations in the environment. We support the network in understanding this relation with the following method: The *Level-1* reward map is obtained by stacking the respective environment and goal maps and processing them with two convolutions. These convolutions use a

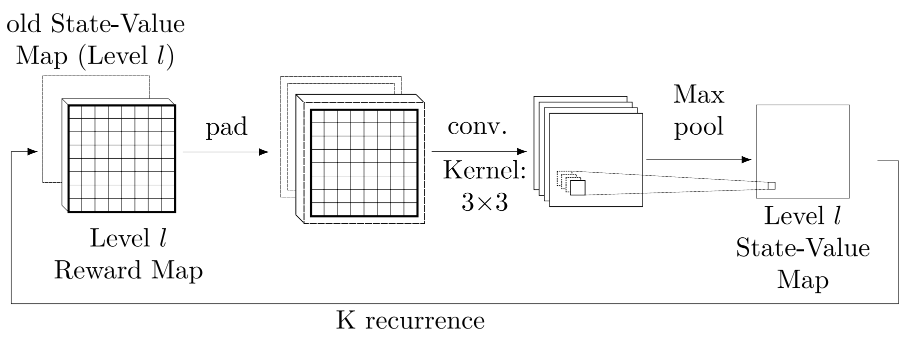


Figure 4.5: Value Iteration Module. The depicted operations are performed for each abstraction level in parallel. The padding operation is used to enable information flow between the levels and is described in Figure 4.6.

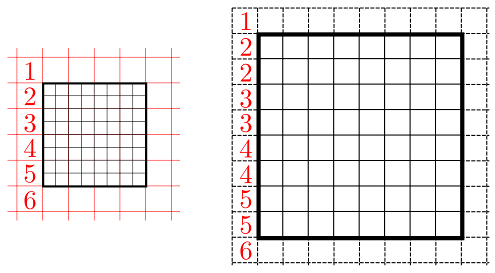


Figure 4.6: Padding the map of abstraction level l (right) to allow information flow from the map of level $l + 1$ (left) to level l. The numbers indicate which values are copied where.

padding to keep the map size constant. Thus, the relation between cell position and environment location stays constant, too.

To enable information flow between levels, the *Level-1* reward map is also used to generate the *Level-2* reward map. A max pooling operation matches the resolution of the *Level-1* map to the *Level-2* resolution. The result is padded with zeros to match the size of the *Level-2* map. This procedure ensures that information at the same cell position in both maps describe the same environment location.

Subsequently, the stacked *Level-1* and *Level-2* maps are processed as described above to obtain the *Level-2* reward map and provide the result to the *Level-3* reward map generation.

4.3.3. VI Module and Reactive Policy

Similar to original VINs, the reward maps are input to the VI Module (Figure 4.5) where they are processed to state-values. Each iteration of the Bellman update is represented through a convolution and subsequent max pooling operation. The kernel is chosen such that it covers the set of possible actions and thus can propagate state values through the map, respectively. Unlike the reward maps, state-value maps consist of only one channel as they describe the expected long-term reward for a pose. At the beginning of each iteration, we apply a padding to the input maps as shown in Figure 4.6 to enable information flow between the different abstraction levels. The padded area contains values of the neighboring cells of the next higher abstraction level. However, the reward maps consist of an increasing number of channels for higher abstraction levels. Therefore, we have to reduce the number of features for the cells of the higher-level reward map before we can use their values as a padding for the lower-level map. To achieve this, we consider two different methods.

First, one possibility is to use the average over all features of one cell of the higher level map as the padding value for all respective cells in the lower level map. However, this method does not pay attention to the meaning of the different features and leads to information loss.

The second option is to enable the network to learn a mapping from higher-level to lower-level features which learns the transformation of these features. We therefore extend the padding operation within the VI Module as follows: For all higher-level cells whose values shall be used as a padding, we feed the corresponding feature vector through a fully-connected layer mapping it to the feature vectors of the corresponding lower-level abstraction cells, which we subsequently use as a padding. Since this method is able to learn to understand the meaning of the features, it preserves more information than the mean-padding method. However, this comes at the cost of a more complex architecture and an increased number of parameters that have to be learned. Both methods are compared against each other in Section 6.1.1.

Output of the VI Module is the *Level-1* state-value map, which is processed by a Reactive Policy. First, the state-values of all neighbors of the start state are extracted. These are subsequently mapped to probabilities over actions through a fully-connected layer.

4.4. Training Details

The networks are trained using the RMSprop optimizer as proposed by Geoffrey Hinton in his lecture (Tieleman and Hinton 2012) which was also used in the original VIN publication. Trying different learning rates, we found that the best training performance was achieved using a learning rate of 0.001, independent of the grid world size. During training, we validate the network every 20 epochs. Subsequently, we choose for each network architecture the state which achieves the highest *success* rate on the validation set. The final results are obtained by evaluating these network states on the evaluation set.

Training data is generated by placing obstacles of random number, size and position into a grid world. Here, the maximum obstacle size is 3×3. However, larger occupied areas are possible if multiple obstacles are placed next to each other. To achieve a similar obstacle density across different grid world sizes, the number of obstacles scales with the number n of grid cells. For each grid, the number of obstacles is sampled between $0.03n$ and $0.1n$. In the original VIN paper, 50 obstacles of maximum size 2×2 are used for grids of size 28×28. Our sampling methods generates 24 to 78 obstacles of maximum size 3×3 for the same grid world size.

While the start state is defined to be in the map center, seven different goals are placed randomly into each grid. Subsequently, we use Dijkstra's algorithm on each environment to simultaneously generate optimal paths to all goal states.

Overall, we generated 5,000 environments, resulting in 35,000 different training scenes. The validation and evaluation sets both consist of 715 additionally generated environments with seven planning tasks each, resulting in 5,005 different scenes for each set. Training, validation and evaluation set are pairwise disjoint.

We consider two different methods how to label our training data: First, as described in the original VIN paper, each scene is assigned the optimal next action as generated by the expert planner. Since the problem can be regarded as a classification problem, Cross Entropy is used as a loss function. However, there may be multiple optimal actions. If the network predicts an optimal action which is not the one predicted from the expert planner, it will be considered as an error. This may affect the training performance.

Therefore, we introduce a second method, where each scene is assigned all optimal next actions. While the network architecture remains unchanged in this case, we have to use a different loss function during training. Since we want the network to output all optimal actions, we can regard the setting as a multi-label classification problem. The probabilities for predicting a certain action should be independent from the other action probabilities. This is achieved by compositing a sigmoid function with the Cross Entropy Loss. The resulting loss function, also known as Binary Cross Entropy Loss, is a standard choice for multi-label classification problems. Both methods are evaluated in Section 6.1.1.

To increase data efficiency during training, we do not only use the whole expert paths but randomly chosen sub-paths. Hence, the network is not only trained on the full paths but on many segments of every expert path which significantly increases the amount of training data. For single-action labels these segments are generated by randomly placing the start and goal poses on the expert path.

However, for multi-action labels, this method has to be adapted. We still keep one example optimal path from which we sample the start position. However, the goal position has to be kept constant. Otherwise, the set of optimal actions for a given state may change.

Having a look at the annotated training examples, we found that the action labels are not equally distributed. Therefore, we weight the losses for the different actions by the inverse action frequencies.

4.5. Path Generation

The network does not directly output the next action but confidence values describing which action is the optimal next one. We choose the predicted action by selecting the one with the highest confidence score. Thus, we only obtain the first action of an optimal path towards the goal. To generate the whole path, we iteratively predict the next action, move the robot accordingly and update the input maps. It may happen that the robot oscillates between multiple states and thus never reaches the goal. For example, the network may predict a wrong action which does not lead to a collision. If it realizes the error when predicting the next action, it will choose to return to the previous state. Since the network output is deterministic, the same wrong action is predicted again. Therefore, the robot oscillates between the two states. To address this problem, we introduce an additional method to generate whole paths. We remember all states from which we return to their direct predecessor and all states that are visited at least twice. When choosing the next action, we check whether the action with the highest confidence value leads to such a state. If this is not the case, we choose this action, otherwise we consider the action with the second highest value and iterate. The effect of this method is evaluated in Section 6.1.3.

5. Adaption to 3D Locomotion Planning

To investigate whether our approach is capable of handling problems of higher complexity and whether it can be applied to real robot motion planning tasks, we apply it to a 3D robot locomotion planning task. We consider a robot that can perform omnidirectional driving and has a certain footprint. Possible actions (Figure 5.1) for the agent are:

- Move to one of the eight adjacent neighbor states with fixed orientation, and
- turn to the next discrete orientation (16 equal orientation steps) with fixed position.

We represent the robot orientation in 16 discrete orientations of equal angular distance.

When generating training data and evaluating the network, collision checking is done by checking if any cell which is occupied by the robot footprint is also occupied by an obstacle. Hence, for robots with modular footprints, it is possible to, e.g., take obstacles between their legs.

5.1. Network Architecture

The architecture for the 3D locomotion planning tasks (Figure 5.2) is based on the one described in Section 4.3. In the following, we state different methods how

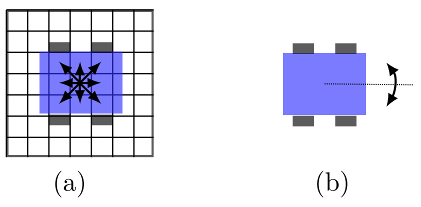


Figure 5.1: Possible actions for 3D robot locomotion planning. Left: Drive to an adjacent neighbor state with fixed orientation. Right: Turn to the next discrete orientation with fixed orientation.

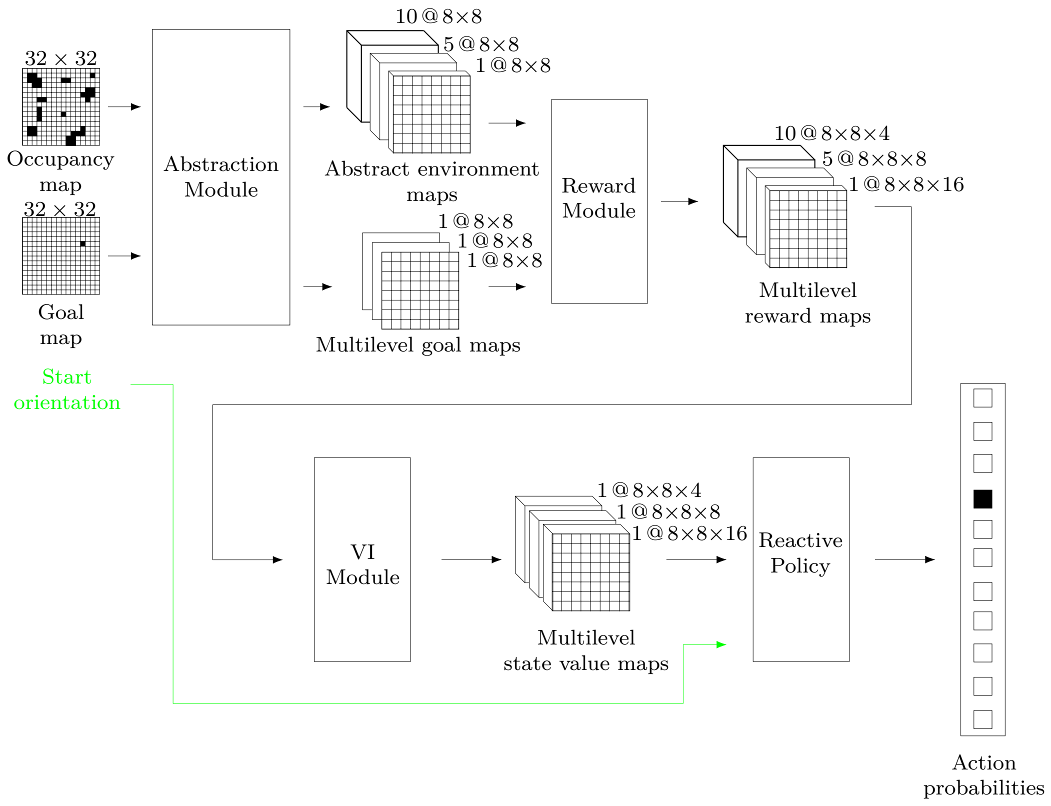


Figure 5.2: Network architecture for 3D locomotion planning. The general structure of the network is similar for each proposed method. However, the start orientation is only provided as explicit input for the approaches introduced in Section 5.1.3 and 5.1.4. The sizes and dimension of the maps differ depending on the method. As an example, we depict the map sizes for the approach presented in Section 5.1.4.

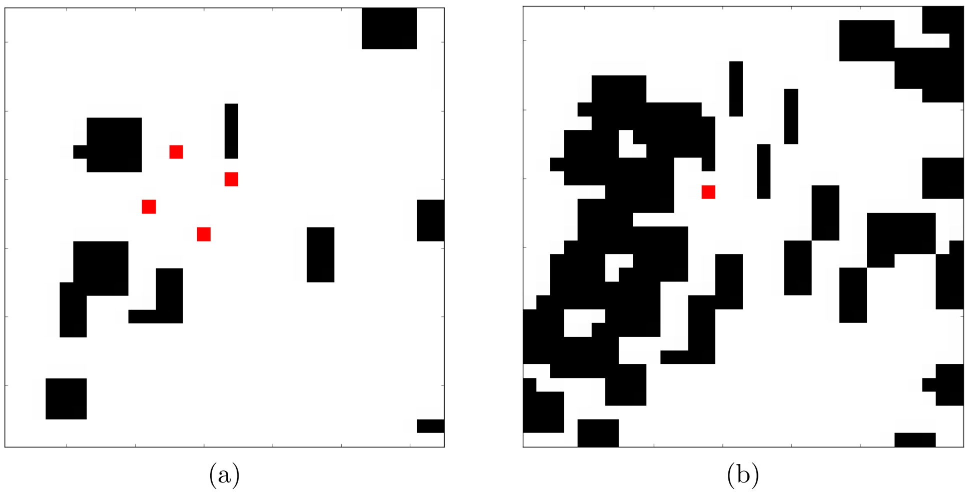


Figure 5.3: Preprocessing the occupancy map. Left: The original 2D occupancy map of the environment with one example robot footprint (red). Right: Slice along the depicted orientation of the corresponding preprocessed 3D occupancy map. The red cell corresponds to the footprint marked in the left picture.

to adapt the Abstraction and Reward Module to the 3D locomotion task. The different methods are evaluated in Section 6.2.1.

5.1.1. Full 3D Architecture

A first adaption extends the Abstraction and Reward Modules to handle 3D input maps. In a preprocessing step, we generate a 3D occupancy map (Figure 5.3) whose third dimension represents the orientations. Hence, for each orientation, we have a 2D map, which are stacked to generate the 3D occupancy map. Each cell of this map represents one possible robot configuration and describes whether this configuration is valid or results in a collision. The goal map is also extended to three dimensions. Again, only the goal configuration is marked.

Due to the cyclic neighborhood relation of orientations, we can cyclically permute the orientation dimension of our input maps. We do this, to ensure that the start configuration is located at the map center, not only with respect to the spatial dimensions but also with respect to the orientation dimension.

As in the 2D case, the input maps are processed by the Abstraction Module. However, we use 3D kernels for the convolution and max pooling operations. Thus, we do not only achieve abstraction for the robot position but also for the orientation. To generate the next higher abstraction level, not only the spatial resolution is halved but also the number of orientations. While we distinguish 16 orientations in the *Level-1* representation, only 8 and 4 abstract orientations are represented in

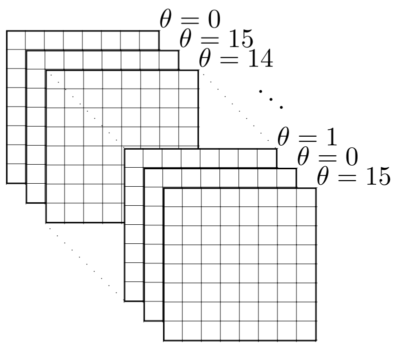


Figure 5.4: Orientation padding during 3D VIs to emphasize that the orientations $\theta = 15$ and $\theta = 0$ are neighbors.

Level-2 and *Level-3*, accordingly. To ensure that all three abstraction maps consist of the same number of cells, we crop a patch around the center of the *Level-2* and *Level-3* maps. Mind that we do not only crop along the spatial dimensions but also along the orientation dimension. Since we represent more lower-level states by a single higher-level state than in the 2D case (eight instead of four), we increase the number of features for *Level-2* to five and for *Level-3* to ten.

For the convolutions within the Reward Module and the VI Module, we also use 3D kernels. Same applies to the max pooling operations within the Reward Module. However, the one-dimensional max pooling operation within the VI Module is not adapted since it is applied to the action channel. As for the 2D grid world task, we pad the reward and state-value maps with values from neighboring cells of the next higher abstraction level to enable information flow between the different abstraction levels (see Figure 4.6). Mind that this padding method is used along all three dimensions of the reward and state-value maps. Since the neighborhood relation for the orientation is cyclic, we introduce an additional padding: The *Level-3* reward and state-value maps are padded on the orientation channel on each end with the values of the opposite end (Figure 5.4). Mind that this is not necessary for the *Level-1* and *Level-2* map since they do not contain all orientations but only the ones near to the start orientation.

Finally, the Reactive Policy extracts the state-values of the neighbor states of the start state and maps it to action probabilities similar to the 2D task. However, the network outputs probabilities for eleven actions instead of eight.

5.1.2. Independently Processed Orientation Channels

While the method described above is an intuitive generalization of our proposed method to 3D locomotion planning, it suffers from the disadvantage of a significantly increased amount of parameters that have to be learned. The 3D convolutions within the VI Module are necessary since the convolution kernel has to cover all possible actions. However, we do not need to process the input using three dimensional convolution and max pooling kernels. Instead, we introduce the idea to process each orientation channel of the input independently within the Abstraction and Reward Module.

As for the method described above, we provide the system with 3D occupancy and goal maps. First, we slice the input maps along the orientation dimension. For each orientation, the resulting 2D slice is processed with exactly the same Abstraction and Reward Module as for the 2D grid world task. Afterwards, the resulting 2D reward maps are stacked along a new dimension to generate a 3D reward map. Mind that this way, we only achieve abstraction for the spatial positions but not for the orientations. Therefore, within the VI Module, only the spatial dimensions of reward and state-value maps are padded as for the 2D grid world tasks. The cyclic padding is not only applied to the orientation channel of the *Level-3* maps but to all abstraction maps.

This method effectively decreases the amount of parameters for the Abstraction and Reward Module. Furthermore, it enables us to initialize the parameters of those two modules with the pretrained ones obtained from the 2D grid world task. This may facilitate and speed up the training progress, since the VI Module is provided with meaningful reward maps from the training start.

5.1.3. 2D Input

Above, we stated that we can process 3D input maps using the 2D architecture for the Abstraction and Reward Module. Therefore, the question arises whether we actually need three dimensional input. We propose a method which is provided with 2D occupancy maps of the environment and extends the state representation to three dimensions within the Reward Module by learning independent rewards for each orientation.

Since we provide our system 2D occupancy and goal maps, we need to find a new way to encode information about the start and goal orientation. The start orientation is fed into our system as an additional parameter. It is only used within the Reactive Policy to select those state-values which belong to neighbor poses of the start pose. The goal orientation is encoded in the goal map in which all cell entries are 0, except for the goal cell which carries the index of the discrete orientation $(1 - 16)$.

The Abstraction Module processes the input as in the 2D case. However, since the Reward Module shall learn orientation dependent rewards and therefore needs more information than in the 2D case, we use five and ten features for *Level-2* and *Level-3*, respectively.

Within the Reward Module, we increase the number of channels of the reward maps of each abstraction level by a factor of 16. Thus, by rearranging the channels, we obtain a 3D reward map. Furthermore, we increase the number of convolutions within the Reward Module by two additional convolutions for processing the *Level-1* map and one additional convolution for the *Level-2* map. To consider the robot footprint, we transform the reward map at the end of the Reward Module: For each possible robot base pose, we sum over the four cells corresponding to the wheel positions and assign the result to the cell corresponding to the robot base pose.

The VI Module and Reactive Policy are left unchanged, compared to the previous method.

5.1.4. 2D Input With Abstract Orientations

For the methods described in Section 5.1.2 and 5.1.3, we do not use any abstraction with respect to orientations. This leads to large state spaces since each abstraction map covers all 16 orientations. We address this issue by reducing the number of orientations for higher abstraction levels. We therefore adapt the above method by setting the number of orientation channels within the Reward Module for *Level-1*, *Level-2*, and *Level-3* to be sixteen, eight, and four, respectively.

5.2. Training Details

The network architectures presented above are trained on grid worlds of size 32×32, which are generated similar to the 2D planning task. However, we randomly place fewer but larger obstacles into these grids. The maximum obstacle size is 5×5 and the number of obstacles is sampled between 5 and 20.

As for the 2D grid world task, we generate three pairwise disjoint sets, one for training, validation end evaluation each. The training set consists of 5,000 environments with seven planning tasks each, resulting in 35,000 different training scenes. The validation and evaluation sets both consist of 715 additionally generated environments with seven planning tasks each, resulting in 5,005 different scenes for each set.

Expert paths are generated using Dijkstra's algorithm. We choose a footprint configuration with a longitudinal and lateral distance of four cell widths between the wheels. This footprint is chosen since it corresponds to a stable driving position for the Centauro robot (see Section 6.2), when using a grid resolution of 0.2 m.

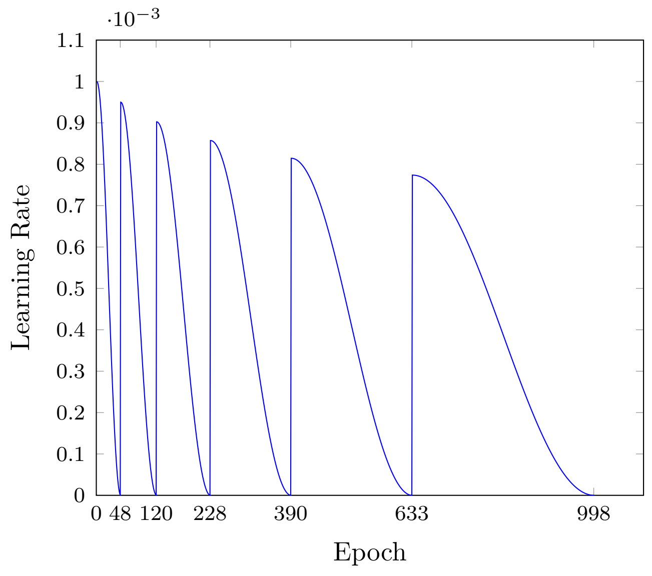


Figure 5.5: Cyclic learning rate.

Collision checking is done by checking if any cell which is occupied by the robot footprint is also occupied by an obstacle.

For the 2D grid world task, we do not use any learning rate scheduling since we evaluate our approach against original VINs which do not use it either. However, for the 3D locomotion planning task, we consider to use a learning rate scheduler.

To reduce the probability that the network converges to sub-optimal local minima, we combine RMSprop with the cyclic learning rate scheduler proposed in Loshchilov and Hutter 2016: The learning rate is decreased using a cosine annealing scheme. After several training epochs, we reset the learning rate to a higher value. We call the time between learning rate resets a learning rate cycle. Initially, the length of a learning rate cycle is set to 48 epochs and the learning rate is 0.001. After each cycle the cycle length increases to 150% of the previous length while the initial learning rate decreases to 95% of the previous one. The evolution of the learning rate is depicted in Figure 5.5.

6. Evaluation

In this chapter, we evaluate the methods proposed in Chapter 4 and 5. All experiments are done on a system equipped with an Intel Core i7-8700K@3.70 GHz, 64 GB RAM and an NVidia GeForce GTX 1080Ti with 11 GB memory.

VINs — and also their extensions considered in this work — are sensitive to weight initialization, which has been previously reported in L. Lee et al. 2018. Different training runs of the same network can lead to significantly different performances. This makes it difficult to evaluate different design decisions. To reduce the influence of random weight initialization, we train each network architecture several times and consider mean and standard deviation as well as the median, which is more robust to far outliers than the mean. Although more training runs allow a more accurate evaluation, we restrict ourselves to five training runs per network architecture due to time limitations. The detailed results of the single training runs can be found in Appendix A.

To furthermore reduce the influence of the random data sampling during training, we initialize the pseudo-random number generator between network initialization and training start with the same seed for each network architecture, while using different seeds for different training runs. Thus, we ensure that each network architecture is presented the same training examples in the same order.

The different network architectures are implemented using Python 2.7 and PyTorch 0.4.1. Our implementation is based on the work of Kent Sommer[1].

6.1. 2D Grid Worlds

In this section we evaluate our approach for the 2D grid world planning task (Chapter 4). First, we compare the different design decisions. The method achieving the best performance is subsequently compared against VINs and HVINs.

[1]`https://github.com/kentsommer/pytorch-value-iteration-networks`

Table 6.1: Evaluation of different implementations of the Reward Module. The configuration of the other components uses mean-padding, weighted class losses and single-action labels.

	Independent Processing			Information Flow		
	mean	median	std dev	mean	median	std dev
Success	94.44%	94.00%	1.96%	**96.94%**	**97.40%**	**1.43%**
Accuracy	81.64%	82.00%	**1.54%**	**84.96%**	**85.30%**	2.37%
Path difference	1.39%	1.41%	0.34%	**0.87%**	**0.73%**	**0.25%**
Graphics memory		**763 MB**			765 MB	
Time (per epoch)		**22 sec**			**22 sec**	

6.1.1. Design Choices

Subsequently, we evaluate the different design decisions concerning the network architecture (Section 4.3) and training method (Section 4.4) on medium-sized grid worlds with 64 $\times$ 64 cells. Since we do not have enough time to evaluate all possible combinations of the proposed designs, we follow the decision tree depicted in Figure 6.1. We start by evaluating the different designs for the first component while choosing the designs for the other components arbitrarily but fixed. For the first component, we then fix the design that achieves the best performance and continue to evaluate the different options for the next component.

Reward Module First, we evaluate the two different options for implementing the Reward Module (Section 4.3.2): Processing each abstraction level independently or allowing information flow between the levels. The configuration of the other components is chosen as follows: The VI Module uses the mean-padding method. During training, we use single-action labels and weight the losses corresponding to the label frequencies.

As expected, enabling information flow between abstraction levels achieves significantly better results than processing each level independently (see Table 6.1). It outperforms the other approach with respect to *success* rate, *accuracy* and path length. Runtime and memory consumption are similar for both methods. The generation of higher-level reward maps profits from using the already processed more detailed information about areas also encoded in the lower-level abstraction maps. All subsequent experiments therefore are done with enabled information flow within the Reward Module.

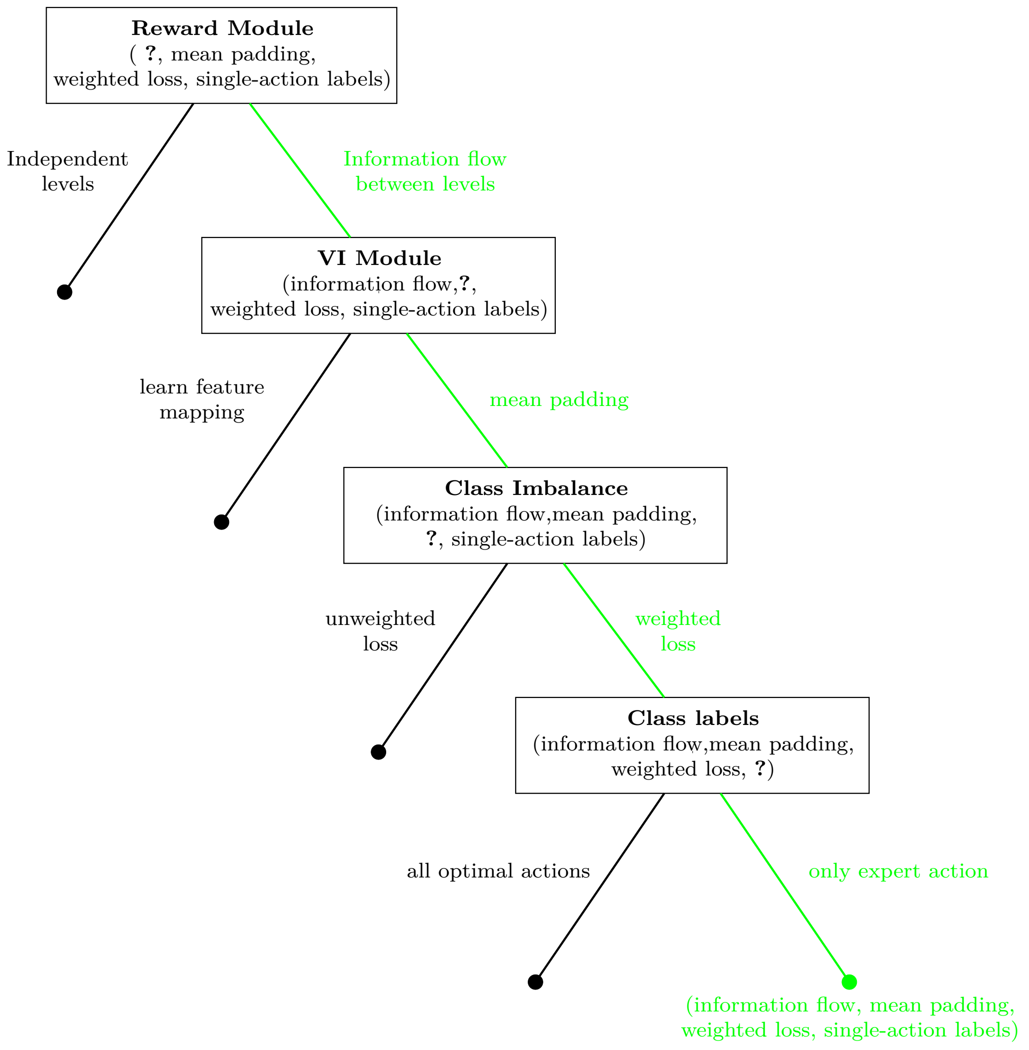

Reward Module
(?, mean padding, weighted loss, single-action labels)
Independent levels
Information flow between levels
VI Module
(information flow,?, weighted loss, single-action labels)
learn feature mapping
mean padding
Class Imbalance
(information flow,mean padding, ?, single-action labels)
unweighted loss
weighted loss
Class labels
(information flow,mean padding, weighted loss, ?)
all optimal actions
only expert action
(information flow, mean padding, weighted loss, single-action labels)

Figure 6.1: Evaluating the design decisions of our proposed method for the 2D grid world planning task. The green edges indicate which configuration achieves better results for the current decision. We start by evaluating the different designs for the first component while choosing the designs for the other components arbitrarily but fixed. For the first component, we then fix the design that achieves the best performance and continue to evaluate the different options for the next component.

Table 6.2: Evaluation of the padding method within the VI Module. The Reward Module uses information flow between the abstraction levels and during training single-action labels and weighted class losses are used.

	Mean Padding			Learned Padding Layer		
	mean	median	std dev	mean	median	std dev
Success	**96.94%**	**97.40%**	**1.43%**	92.88%	91.60%	2.22%
Accuracy	**84.96%**	**85.30%**	**2.37%**	81.44%	80.40%	2.89%
Path difference	**0.87%**	**0.73%**	**0.25%**	1.72%	1.67%	0.77%
Graphics memory		**765 MB**			767 MB	
Time (per epoch)		**22 sec**			23 sec	

VI Module Subsequently, we evaluate the padding method within the VI Module (Section 4.3.3). During training, we still use single-action labels and weighted class losses. On the one hand, introducing the padding layer increases the network capabilities to transfer useful information between the abstraction levels. The network may learn to average the features as in the mean-padding method, but also more useful feature mappings are possible. On the other hand, introducing the padding layer increases the network complexity and the number of adjustable parameters. This may affect the training performance.

The results depicted in Table 6.2 show that this negative influence dominates the benefits of the padding layer. Runtime and memory consumption is similar for both approaches. However, compared to the padding-layer method, the mean padding method achieves higher *success* rates and *accuracies*, and lower deviation between the optimal path lengths and the length of the predicted paths. Furthermore, the standard deviation between the five different training runs is smaller.

To give a possible explanation, we assume that the network encodes different kinds of cell traversals in the different reward features. We use the same fully-connected layer for determining the padding values for the left/ right borders as for the upper/ lower borders of the reward maps. Therefore, the padding layer has to learn a method mapping the two *Level-2* features of a given cell to its single *Level-1* feature without prior knowledge of the direction from which we enter the cell. For each cell, the best method is to take the expected reward for entering this cell. Without prior knowledge, the probabilities of the actions leading to a given cell are uniformly distributed. Hence, the expected reward corresponds to the mean over the *Level-2* features. Thus, the padding-layer should learn to average the *Level-2* features, which is exactly what the mean-padding method does. As a result, introducing the padding-layer increases the network complexity but does

Table 6.3: Evaluation of weighting the losses for predicting wrong actions with the inverse of the action frequencies. The Reward Module uses information flow between the abstraction levels, the mean-padding method is used within the VI Module and during training single-action labels are used.

	Unweighted			Weighted		
	mean	median	std dev	mean	median	std dev
Success	94.12%	94.70%	1.95%	**96.94%**	**97.40%**	**1.43%**
Accuracy	82.42%	83.20%	3.51%	**84.96%**	**85.30%**	**2.37%**
Path difference	1.67%	1.65%	0.36%	**0.87%**	**0.73%**	**0.25%**

not improve the information flow from *Level-2* to *Level-1*. In the following, we use the mean-padding method for all experiments.

Class Imbalance After evaluating the different network architectures above, we consider the different training methods (Section 4.4). Since these have no influence on the runtime and memory consumption, we do not state the resource consumption for these experiments. First, we address the problem of imbalanced action labels. We penalize wrong predictions of less frequent actions more than wrong predictions of more frequent actions by weighting the losses with the inverse of the label frequencies. Table 6.3 shows that this indeed improves the training performance. It outperforms the method with unweighted losses with regard to all performance measures. Thus, we use weighted action losses for all subsequent experiments.

Class labels Finally, we evaluate whether the training performance can be increased by providing the network not only with the optimal action chosen by the expert planner but with all optimal actions for each training example. Mind, that we use a different sampling method to extract sub-paths from expert paths in the case of multi-action labels. The adapted sampling method results in a reduced number of training examples compared to the training data used in the previous experiments. To ensure that performance differences between the different label types are not caused by this, we use the adapted sampling method in this experiment for both, single-action labels and multi-action labels. Table 6.4 shows the results.

Comparing the performance of the single-action labels with the adapted sampling method to the performance of the previous experiments, we see that changing the sampling method does not effect the performance significantly. The slight dif-

Table 6.4: Evaluation of different methods how to label the training data.

	Single-action			Multi-action		
	mean	median	std dev	mean	median	std dev
Success	**95.52%**	**94.20%**	**2.40%**	72.00%	86.50%	21.48%
Accuracy	86.44%	86.40%	**1.23%**	**92.08%**[2]	**94.30%**[2]	3.49%[2]
Path difference	0.73%	0.70%	**0.08%**	**0.60%**	**0.55%**	0.36%

ferences that can be observed are not larger than the variance between different training runs. However, using multi-action labels results in lower *success* rates and significantly larger standard deviation between different training runs. Mind that the increased *accuracy* of multi-action labels is due to the fact that, in this case, we measure whether the predicted action is one of the optimal next actions and not only whether it is the one predicted by the expert planner. A possible explanation for the lower *success* rate of multi-action labels is the following: The Reactive Policy should predict the action leading to the state with the highest state-value. In theory, if there are multiple optimal actions, the state-values of the corresponding target states are equal. Due to numerical inaccuracies and since we are using approximations of the optimal state-values, we cannot expect that this is true for the state-values generated by the VI Module. Therefore, it is difficult for the Reactive Policy to decide how many and which actions are optimal. A more complex Reactive Policy with multiple layers may perform better by learning a threshold for the state-value differences between target states of multiple optimal actions.

6.1.2. Final Results

In this section, we compare our method against original VINs and HVINs. Similar to our approach, we use three hierarchical levels for HVINs, each halving the resolution of the previous level. The lowest resolution level uses the same number of Bellman updates (K) as proposed for original VINs with a similar number of cells. This coarse state-value initialization is then refined twice by two Bellman updates on the map with medium resolution and two consecutive Bellman updates on the map with fine resolution.

[2]In the case of multi-action labels, the accuracy does not measure whether the action predicted by the network is the one predicted by the expert planner, but whether the predicted action is one of the optimal next actions.

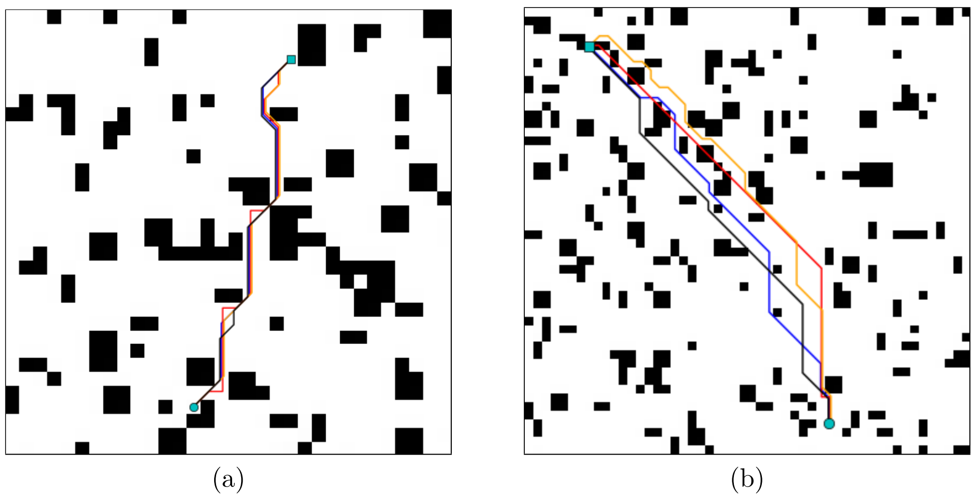


Figure 6.2: Patches of grid worlds including the optimal path (black) and the paths predicted by VIN (red), HVIN (orange), and our approach (blue). Left: Patch form a 64 × 64 grid world. Right: Patch form a 128 × 128 grid world.

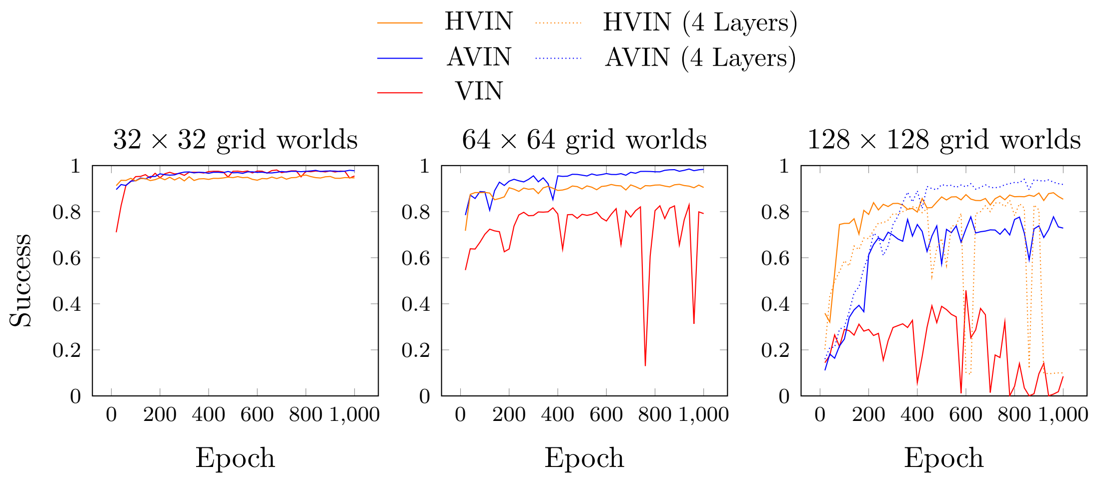


Figure 6.3: Training performance of original VINs, HVINs and our approach on the validation set. For each network architecture, the best performance out of five training runs is depicted.

Coarse 2D Grid Worlds In the original VIN paper (Tamar et al. 2016), tests were performed on 2D grid worlds of sizes from 8×8 to 28×28. Since we are interested in applying learning-based planning approaches to more complex problems, we evaluate on grid sizes from 32×32 to 128×128 cells. Some example paths are depicted in Figure 6.2.

Figure 6.3 shows the training performance of our approach and compares it to original VINs and HVINs. It can be seen that our approach obtains better *success* rates than VINs and HVINs on the validation set for map sizes of 32×32 and 64×64 cells. In addition, our approach converges faster and with a higher stability. Especially on larger maps, original VINs show large instabilities in their training behavior. On 128×128 maps, we achieve a better performance than VINs. However, HVINs even achieve a higher *success* rate. We will refer to this observation in more detail below.

The results over multiple training runs are shown in Table 6.5. On 32×32 grid worlds, our approach outperforms VINs and HVINs in terms of a better *success* rate. Furthermore, we obtain paths that are closer to the optimal paths. For grid worlds of size 64×64, the performance differences even increase. Our approach achieves significantly higher *accuracies* and *success* rates, while the path lengths are significantly shorter. On the 128×128 grid worlds, our approach still outperforms VINs. However, HVINs achieve higher *success* rates and shorter paths. The performance drop of our approach can be explained since the abstraction maps have a size of 32×32 for input maps of size 128×128. This seems to be a critical threshold, above which the performance of VINs rapidly decreases. Since we employ VINs on each abstraction level, the performance of our approach drops significantly. HVINs also start by employing VINs on a 32×32 map. Subsequently, they improve the result by iteratively refining the resulting state-value map up to the original input resolution. This refinement process seems to be less sensitive to large input maps and may recover some errors introduced by applying VINs to the 32×32 map in the first step.

Above, we evaluated the result quality of the different network architectures. However, the graphics memory consumption and necessary training time is also important since too large resource requirements make it difficult to scale the methods to more complex tasks. Memory consumption and runtime are depicted in Figure 6.4 and Table 6.5. The graphics memory consumption of VINs rapidly grows with increasing map sizes. Since all levels of HVINs operate on the whole environment size, they only slightly decrease memory requirements. Our approach outperforms VINs and HVINs on all grid world sizes.

Regarding the training time per epoch (right side of Figure 6.4), our approach outperforms VINs significantly on larger grid sizes due to the reduced state space

Table 6.5: Evaluating our approach against VINs and HVINs on the 2D grid world path planning task. For all networks, single-action labels and weighted losses are used.

Grid		Success			Accuracy		
size	Method	mean	median	std dev	mean	median	std dev
32×32	VIN	91.72%	94.90%	8.15%	80.38%	**83.60%**	10.31%
	HVIN	93.84%	93.90%	1.35%	80.08%	81.00%	1.78%
	AVIN	**96.84%**	**96.60%**	**0.38%**	**83.40%**	82.80%	**1.31%**
64×64	VIN	71.98%	79.70%	20.86%	70.08%	74.90%	12.98%
	HVIN	86.82%	89.80%	8.90%	77.42%	77.90%	**2.29%**
	AVIN	**96.94%**	**97.40%**	**1.43%**	**84.96%**	**85.30%**	2.37%
128×128	VIN	31.56%	34.00%	11.13%	55.96%	55.90%	5.96%
	HVIN	**83.24%**	**82.60%**	6.05%	76.50%	76.40%	**2.17%**
	AVIN	80.34%	78.60%	**5.13%**	**80.86%**	**80.40%**	3.59%

Grid		Path diff.			Graphics	Time
size	Method	mean	median	std dev	memory	(per epoch)
32×32	VIN	2.48%	2.09%	1.98%	761 MB	8 sec
	HVIN	2.23%	2.07%	0.79%	739 MB	**5 sec**
	AVIN	**1.73%**	**1.84%**	**0.39%**	**561 MB**	11 sec
64×64	VIN	2.94%	2.42%	1.59%	1815 MB	34 sec
	HVIN	2.55%	2.20%	1.06%	1399 MB	**10 sec**
	AVIN	**0.87%**	**0.73%**	**0.25%**	**765 MB**	22 sec
128×128	VIN	8.46%	7.18%	5.33%	8247 MB	247 sec
	HVIN	**2.09%**	**2.08%**	**0.49%**	4085 MB	**35 sec**
	AVIN	2.48%	2.81%	0.59%	**1731 MB**	63 sec

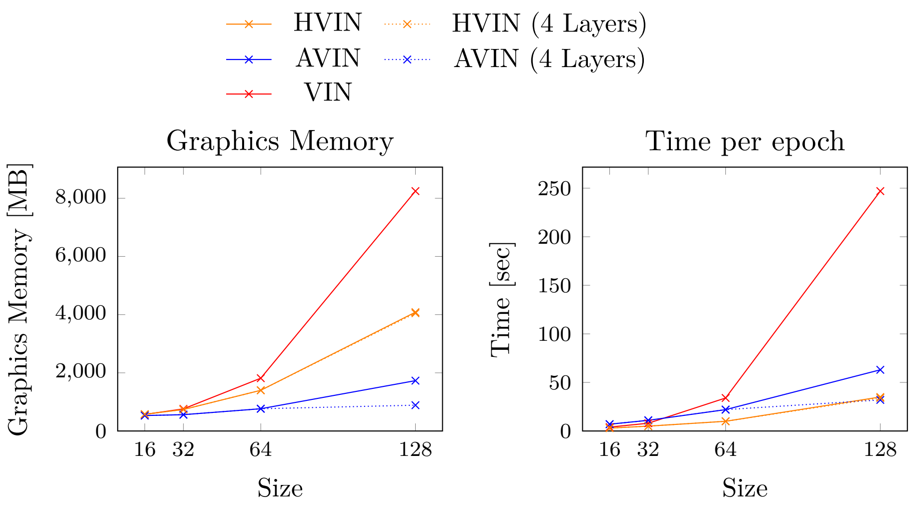


Figure 6.4: Graphics memory consumption and runtime of VINs, HVINs and our approach dependent on the grid size.

size and thus fewer Bellman updates. On 32 × 32 grid worlds, the additional runtime needed to create and process the different abstraction maps dominates this effect, which results in faster training times for VINs for this and smaller map sizes. However, we do not match the runtime of HVINs since the refinement process is much faster than performing full value iterations for all abstraction maps.

Additionally, we compare the planning time of our approach against an A* planner. Within the A* planner, we use the euclidean distance as a heuristic to estimate the remaining costs from the current position to the goal position. Figure 6.5 shows that the A* planner significantly outperforms our approach. Since the planning time needed to predict one action is significantly higher four our approach than for the A* planner, the performance difference even increases on larger grid worlds.

Four Abstraction Levels To further investigate the performance drop of our approach on 128 × 128 grid worlds, we add an additional abstraction level (with 10 features) to our method, extending it to four levels. Thus, the size of each abstraction map is reduced to 16 × 16. Additionally, we also extend HVINs to four hierarchical levels. As before, we use the same random seed for both architectures when sampling the training data. The results are depicted in Table 6.6.

With four abstraction levels, our approach outperforms HVINs with respect to *accuracy* and path length. However, the average *success* rate is lower compared to HVINs and even lower compared to our approach with three abstraction levels. This is due to the fact that one of the five training runs only achieved a *success* rate

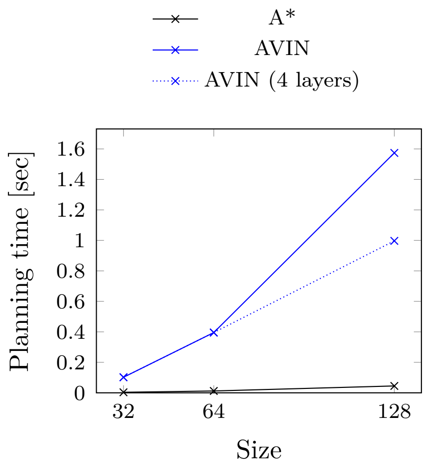


Figure 6.5: Planning times of our approach and an A* planner for 2D grid world path planning.

Table 6.6: Using four abstraction levels for 128 × 128 grid worlds.

	Success			Accuracy		
	mean	median	std dev	mean	median	std dev
HVIN	83.24%	82.60%	6.05%	76.50%	76.40%	2.17%
HVIN (4 Levels)	**84.00%**	**85.20%**	**2.34%**	78.04%	78.70%	**1.50%**
AVIN	80.34%	78.60%	5.13%	**80.86%**	**80.40%**	3.59%
AVIN (4 Levels)	78.14%	83.90%	23.33%	80.06%	80.10%	5.30%

	Path diff.			Graphics	Time
	mean	median	std dev	memory	(per epoch)
HVIN	2.09%	2.08%	**0.49%**	4085 MB	35 sec
HVIN (4 Levels)	2.80%	2.42%	0.64%	4049 MB	34 sec
AVIN	2.48%	2.81%	0.59%	1731 MB	63 sec
AVIN (4 Levels)	**1.81%**	**1.51%**	0.60%	**887 MB**	**32 sec**

of 38.2% due to bad weight initialization. However, the best training run for our approach achieved a *success* rate of 94.8%, while the best *success* rate for HVINs is 86.3% (see Figure 6.3). If we omit the training run with the worst performance for each method, our approach achieves an average of 88.13%, while HVINs only achieve 84.85%. This indicates that in general, our approach achieves better results then HVINs when using four abstraction levels on 128×128 grid worlds and that it also achieves better results than our approach with three levels. While the performance of both methods is increased by using four levels for 128×128 grids, our approach benefits more from the additional level than HVINs. Introducing an additional level results in a coarser resolution for the lowest level of HVINs. A possible reason why HVINs benefit less from the additional level than our approach is, that this coarser resolution may result in information loss which the refinement process of HVINs does not compensate. Our approach seems to compensate the information loss better since it enriches the coarse resolution map with additional features.

Furthermore, using four abstraction levels for 128×128 grid worlds significantly reduces the memory consumption and training time for our approach, while the resource requirements of HVINs do not change significantly (see Figure 6.4 and Table 6.6). The memory requirements for our approach are only 10.8% of original VINs and 21.9% of HVINs and we achieve a similar runtime as HVINs. Furthermore, using four abstraction levels significantly reduces the planning time of our approach (see Figure 6.5), although it is still significantly slower than an A* planner.

2D Mazes Above, we evaluated our approach, VINs, and HVINs on grid worlds which are larger than the ones used in the original VIN paper but which have a similar density of obstacles. To investigate how the different approaches can handle more complex environments, we additionally evaluate them on mazes. Each corridor has a width of one cell and each goal position is connected to the start position only by one single path. Since path planning for theses mazes is much more difficult than for the grid worlds considered above, we use maze sizes from 16×16 to 64×64. The number of training examples and the training method are the same as for the previous 2D grid world experiments. Table 6.7 shows the corresponding performances and some example paths are depicted in Figure 6.7.

This experiment effectively shows, how much information is preserved in the abstract representations. Since VINs consider the complete map at full resolution, they achieve the best results independent of the maze size. For each maze size, the high standard deviation of the *success* rate of VINs is due to one single training run with a bad weight initialization. Our approach looses information due to the more

Table 6.7: Evaluating our approach against VINs and HVINs on 2D maze worlds.

Maze		Success			Accuracy		
size	Method	mean	median	std dev	mean	median	std dev
16×16	VIN	**94.48%**	**99.60%**	11.51%	**94.42%**	**99.30%**	9.61%
	HVIN	87.42%	89.70%	3.36%	87.20%	88.90%	3.24%
	AVIN	83.00%	82.80%	**0.74%**	82.52%	82.60%	**0.83%**
32×32	VIN	**82.10%**	**97.20%**	31.58%	**85.60%**	**95.50%**	19.13%
	HVIN	48.54%	47.30%	3.84%	69.94%	69.60%	2.55%
	AVIN	71.30%	71.00%	**3.14%**	82.76%	83.10%	**1.89%**
64×64	VIN	**78.02%**	**90.40%**	27.63%	**84.58%**	**85.90%**	11.15%
	HVIN	14.22%	14.20%	**1.26%**	58.82%	58.50%	**0.94%**
	AVIN	57.06%	56.00%	5.74%	80.62%	80.50%	1.82%

Maze		Path diff.			Graphics	Time
size	Method	mean	median	std dev	memory	(per epoch)
16×16	VIN	**0.51%**	**0.06%**	0.72%	569 MB	4 sec
	HVIN	1.16%	1.11%	0.31%	575 MB	**3 sec**
	AVIN	2.05%	2.04%	**0.21%**	**531 MB**	7 sec
32×32	VIN	**0.88%**	**0.57%**	0.96%	761 MB	8 sec
	HVIN	2.02%	1.91%	0.44%	739 MB	**5 sec**
	AVIN	1.10%	1.10%	**0.14%**	**561 MB**	11 sec
64×64	VIN	1.49%	1.32%	0.87%	1815 MB	34 sec
	HVIN	1.99%	1.93%	0.24%	1399 MB	**10 sec**
	AVIN	**0.58%**	**0.58%**	**0.07%**	**765 MB**	22 sec

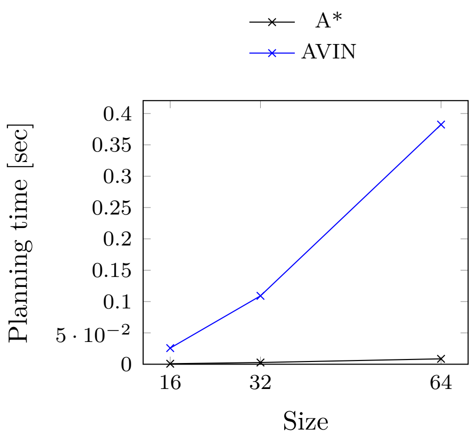


Figure 6.6: Planning times of our approach and an A* planner on 2D mazes.

abstract representations. The features of the higher abstraction level cells cannot encode all necessary information to match the performance of VINs. However, we significantly outperform HVINs on the larger maze sizes, which indicates that our approach nevertheless learns useful abstract representations.

On 16×16 mazes, HVINs perform better than our approach. For this maze size, the area described at full resolution within our planner only covers 4×4 cells, which may be to small.

Interestingly, VINs achieve better results for mazes than for the grid worlds of the previous experiment. Here, we compare the median to reduce the influence of far outliers, which significantly reduce the mean *success* rates for VINs on the maze planning task. When training on mazes, all training examples contain obstacles next to the current agent position. Thus, VINs learn more effectively to avoid obstacles, which results in a better performance.

Compared to the previous grid world experiments, training time and memory consumption do not change since these only depend on the map size but not on concrete obstacle configurations. The difference between the planning times of our approach and an A* planner (Figure 6.6) is also similar to the previous experiment.

6.1.3. Path Generation with History

For the 2D grid worlds with coarse obstacle distribution, the most frequent reason that a predicted path does not successfully lead to the goal state are collisions. However, for the 2D mazes, paths often are not successful since they contain circles. Thus, the agent alternates between two or more states and never reaches the goal. An example is shown in Figure 6.8 a). Our approach (depicted in blue) predicts a wrong action and corrects the mistake when predicting the next action

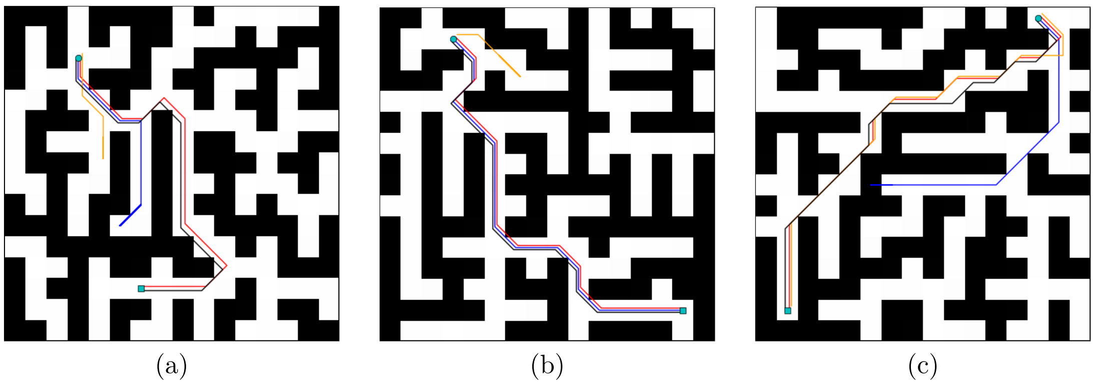


Figure 6.7: Patches of 32 × 32 mazes including the optimal path (black) and the paths predicted by VIN (red), HVIN (orange), and our approach (blue). Left: Only VIN successfully reaches the goal. Middle: Our approach and VIN predict optimal paths, while HVIN does not find a path. Right: VIN predicts an optimal path, HVIN successfully reaches the goal, while our approach leads to a collision.

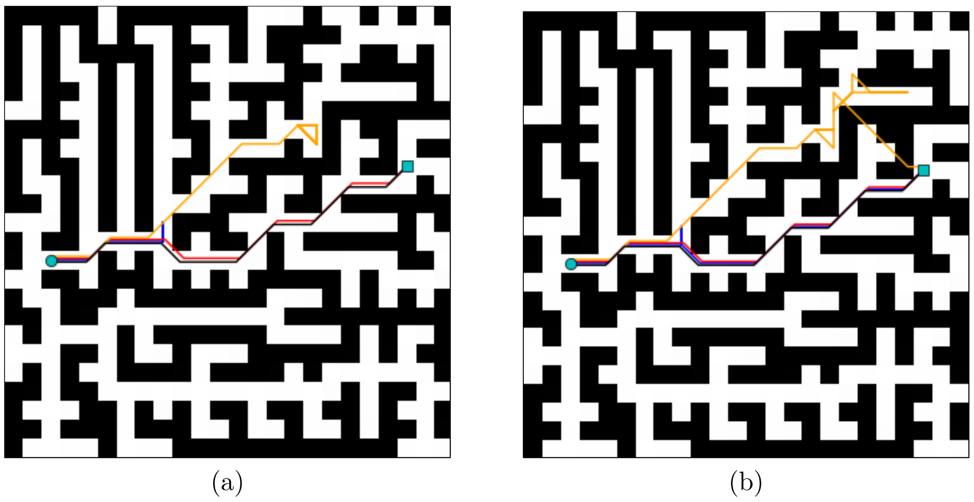


Figure 6.8: Different path generation methods. Depicted is a patch of a 64 × 64 maze including the optimal path (black) and the paths predicted by VIN (red), HVIN (orange), and our approach (blue). Left: Always choosing the action with highest score. Right: Improved path generation method (Section 4.5).

Table 6.8: *Success* rates of the two different path generation methods on maze worlds. The path generation method which considers the history (Section 4.5) improves the *success* rates for all architectures and maze sizes.

Maze size	Method	Without history mean	Without history median	Without history std dev	With history mean	With history median	With history std dev
16×16	VIN	94.48%	99.60%	11.51%	**95.46%**	**99.70%**	9.43%
	HVIN	87.42%	89.70%	3.36%	90.58%	92.10%	3.15%
	AVIN	83.00%	82.80%	**0.74%**	87.26%	87.60%	0.96%
32×32	VIN	82.10%	97.20%	31.58%	**82.72%**	**97.50%**	30.85%
	HVIN	48.54%	47.30%	3.84%	55.40%	53.90%	3.45%
	AVIN	71.30%	71.00%	**3.14%**	75.42%	75.60%	4.02%
64×64	VIN	78.02%	90.40%	27.63%	**79.24%**	**91.40%**	26.68%
	HVIN	14.22%	14.20%	**1.26%**	17.25%	17.20%	2.78%
	AVIN	57.06%	56.00%	5.74%	59.42%	58.60%	5.66%

by returning to the previous position. Sine the network output is deterministic, this results in alternating between the two states. In Section 4.5, we introduced a path generation method that addresses this problem. Figure 6.8 b) shows how the paths change if this method is applied. After undoing the wrong action, our planner now successfully reaches the goal. However, the path predicted by HVIN (orange) is not improved since the network does not immediately realize the error.

We apply the new path generation method to re-evaluate the networks from the maze path planning task of the previous section. Table 6.8 shows that this indeed increases the *success* rates. Especially, our approach and HVIN benefit from the new path generation method.

6.2. 3D Locomotion Planning

Since we are interested in applying learning-based planners to real robot motion planning tasks, we investigate in this section, how well our approach can be adapted to 3D locomotion planning. We plan for Centauro (Klamt, Rodriguez, et

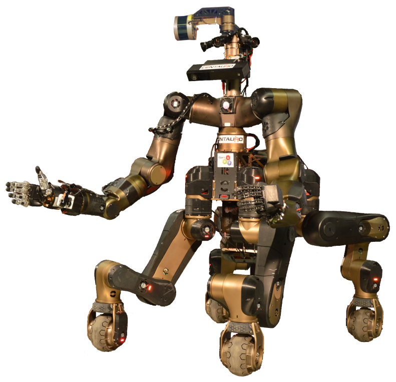


Figure 6.9: The Centauro robot.

al. 2018) which is a centaur-like robot consisting of a hybrid legged-wheeled lower body and an anthropomorphic upper body (Figure 6.9). It is developed to solve a wide range of mobile manipulation tasks in search and rescue environments. Each leg ends in a 360° steerable actively driven wheel which allows omnidirectional driving.

In the following, we evaluate the different design choices for 3D locomotion planning introduced in Chapter 5. Afterwards, we integrate the method which achieves the best performance into a planning pipeline for Centauro.

6.2.1. Design Choices

In this section, we evaluate the different methods presented in Chapter 5 how to adapt our method to the 3D robot locomotion planning task on grid worlds of size 32×32. We choose a footprint configuration with a longitudinal and lateral distance of four cell widths between the wheels. For a grid resolution of 0.2 m, this corresponds to a stable driving position for Centauro with 0.8 m longitudinal and lateral distance between the wheels. Hence, the same footprint configuration can be used for the Centauro experiment in Section 6.2.2.

First, we evaluate the cyclic learning rate scheduler described in Section 5.2. To do this, we compare the performance of the network from Section 5.1.3 trained with and without using the scheduler. The corresponding training performances for one training run for each method are depicted in Figure 6.10. One clearly sees the different learning rate cycles. Due to the cosine annealing scheme, the training error rapidly decreases within each cycle. At the end of each cycle, a lower training error is achieved compared to the cycles before. When using the cyclic learning rate scheduler, the network converges faster and achieves lower training error and a higher *success* rate.

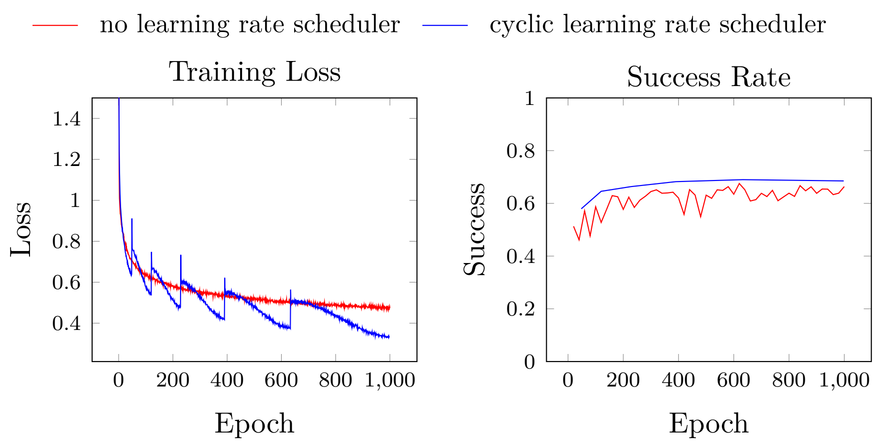


Figure 6.10: Evaluation of the cyclic learning rate scheduler. The success rate for training with the cyclic learning rate scheduler is evaluated at the end of each learning rate cycle and thus less frequent then when training with no scheduler.

Table 6.9 shows the final results over multiple training runs. Although higher *accuracies* are achieved by not using a learning rate scheduler, the more important *success* rates are higher when using the cyclic scheduling method. Furthermore, the cyclic scheduler achieves shorter paths.

When using no scheduler, the network converges to sub-optimal local minima. During the cyclic learning rate scheduling, we reset the learning rate several times to higher values before decreasing it again. This enables the network to leave local minima and therefore achieves better results. The ability to step out of local minima — which may be reached due to bad initial weights — also explains why the cyclic learning rate scheduler is less sensitive to weight initialization. This can be seen, since the standard deviation between the different training runs is smaller when using the scheduler compared to using no learning rate scheduling. All subsequent experiments use the cyclic scheduling scheme.

Subsequently, we evaluate the different network architectures presented in Section 5.1. The results are depicted in Table 6.10.

The Full 3D architecture achieves the lowest *success* rates. It does not learn to successfully predict paths since it consists of too many adjustable parameters due to the 3D convolutions. However, it achieves the lowest runtime and graphics memory consumption. This is due to the fact that the Full 3D architecture uses abstraction for both, the spatial position and the orientation.

Processing the orientation channels of the input maps independently by regarding them as 2D maps, results in a better performance since the complexity of Abstraction Module and Reward Module is decreased. However, no abstraction with

Table 6.9: Evaluating the cyclic learning rate scheduling against no learning rate scheduling. For both cases, the network architecture from Section 5.1.3 is used.

	No scheduling			Cyclic scheduling		
	mean	median	std dev	mean	median	std dev
Success	64.18%	64.30%	1.11%	**67.58%**	**67.60%**	**0.29%**
Accuracy	**67.06%**	**67.20%**	1.74%	66.74%	66.90%	**1.41%**
Path difference	2.71%	2.41%	0.64%	**1.85%**	**1.87%**	**0.25%**

respect to orientations is used resulting in higher runtime and graphics memory consumption. Initializing the weights for Abstraction and Reward module with the pretrained weights from the 2D network does not increase the performance. Contrarily, the performance decreases. A possible reason is that the preprocessed occupancy maps for the 3D task significantly differ from the ones considered in the 2D task, i.e., they contain much larger obstacles (see Figure 5.3). Therefore, the pretrained weights do not generalize well to the new input maps. A better way to pretrain those weights probably is to train them directly on the 2D slices of the 3D input maps, which was not realized within this work due to time limitations.

Using 2D input maps and expanding to 3D reward maps within the Reward Module, achieves even better results. However, since this method does not use abstraction for representing orientations, runtime and graphics memory consumption are still high. By reducing the number of orientations for higher abstraction levels, runtime and memory consumption is effectively reduced. Furthermore, this method achieves the highest *success* rates and *accuracies*. This shows that reducing the state space size using abstraction facilitates the training process and thus results in a better performance.

6.2.2. Integration For Centauro

Subsequently, we employ our approach to generate 3D locomotion plans for Centauro. Since it achieves the highest *success* rates on the synthetically generated 32×32 grid worlds, we use the network architecture described in Section 5.1.4 for the subsequent experiments. In particular, we use the network state with the highest *success* rate across five training runs.

We choose a fixed leg configuration for Centauro with 0.8 m longitudinal and lateral distance between the wheels. A 3D rotating Velodyne Puck laser scanner at the robot head with spherical field-of-view perceives the environment. Measure-

Table 6.10: Evaluating the different methods how to adapt our method to 3D robot locomotion planning on maps of size 32×32. For all methods, the cyclic learning rate scheduler is used.

	Success			Accuracy		
Method	mean	median	std dev	mean	median	std dev
Full 3D	21.56%	21.90%	3.83%	64.16%	63.60%	1.29%
Ind. orientations	55.76%	61.40%	11.37%	63.84%	66.30%	5.18%
Ind. orientations (pretrained)	54.26%	52.90%	7.62%	62.84%	64.50%	6.63%
2D input	67.58%	67.60%	**0.29%**	66.74%	66.90%	1.41%
2D input abstract orient.	**73.72%**	**73.50%**	1.75%	**71.14%**	**71.00%**	**0.98%**

	Path diff.			Graphics	Time
Method	mean	median	std dev	memory	(per epoch)
Full 3D	**1.60%**	1.36%	0.61%	**793 MB**	**22 sec**
Ind. orientations	2.43%	**1.06%**	3.27%	1431 MB	37 sec
Ind. orientations (pretrained)	2.66%	1.35%	3.19%	1431 MB	37 sec
2D input	1.85%	1.87%	0.25%	1073 MB	39 sec
2D input abstract orient.	1.65%	1.61%	**0.22%**	821 MB	27 sec

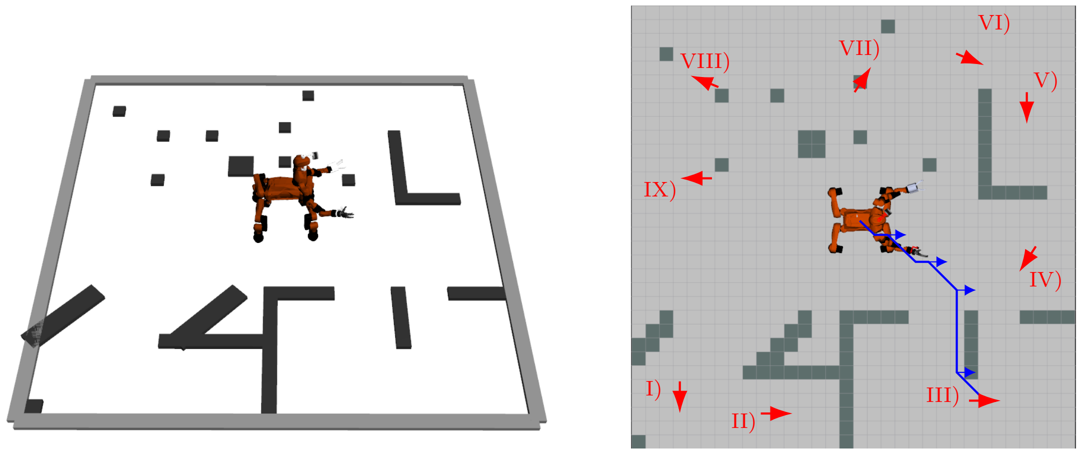


Figure 6.11: 3D Locomotion planning experiment. Left: Gazebo arena with Centauro. Right: The corresponding occupancy map with the nine chosen goals and one example result path.

ments are processed to registered point clouds while the robot is localized using the method of Droeschel, Schwarz, and Behnke 2017. Finally, occupancy maps with a resolution of 0.2 m are generated from these point clouds and are used as input to our network. Robot perception and control is implemented in C++ and the communication with the network is realized using ROS.

Experiments are performed in the Gazebo simulation environment. For this, we designed a test environment with cluttered terrain, in which we placed nine different goal poses (Figure 6.11). A video with additional footage of the experiments is available online[2].

To evaluate the performance of our approach, we compare it against the A* planner. The heuristic used for the A* planner consists of the sum of the Euclidean distance to the goal and the distance between the current orientation and the goal orientation. In particular, let (p, θ) be the current pose, where p is a 2-dimensional vector representing the base location and θ denotes the current orientation. Analogously, let $(p_{\text{goal}}, \theta_{\text{goal}})$ describe the goal pose. We estimate the remaining costs as

$$h = \|p_{\text{goal}} - p\|_2 + |\theta_{\text{goal}} - \theta| \sqrt{\left(\frac{r_l}{2}\right)^2 + \left(\frac{r_w}{2}\right)^2},$$

where r_l and r_w denote the length and width of the robot footprint.

Table 6.11 shows the results of our approach and the expert A* planner for the tasks depicted in Figure 6.11. For most tasks, our planner generates optimal or close to optimal paths. It successfully considers the robot footprint and thus is

[2]https://www.ais.uni-bonn.de/videos/ICRA_2019_Schleich

Table 6.11: Results of our approach and the A^*-planner for the tasks depicted in Figure 6.11.

	AVIN		A^*-planner	
	Path length	Planning Time	Path length	Planning Time
I)	24.59	0.420 sec	23.41	0.131 sec
II)	Not found		24.14	0.754 sec
III)	17.90	0.312 sec	17.90	0.080 sec
IV)	18.80	0.355 sec	18.80	0.265 sec
V)	Not found		27.76	1.630 sec
VI)	17.01	0.292 sec	17.01	0.133 sec
VII)	14.16	0.271 sec	13.33	0.046 sec
VIII)	24.67	0.438 sec	22.92	0.177 sec
IX)	22.13	0.417 sec	22.13	0.615 sec

able to plan paths where obstacles have to be taken between the legs (e.g., III and VII). One example path is depicted in Figure 6.11. However, our planner does not find successful paths for II and V. For both tasks, the generated path results in a collision.

The planning times of our approach have a lower variation than for the A* planner. In most of the cases, the A* planner achieves the best planning times. However, for IX, it is outperformed by our approach.

To give a more reliable comparison of the runtime of both planning systems, we also evaluate the planning times on the synthetically generated test set, which was used in the previous section. Here, we only consider test scenes, where our approach generates a successful path. Thus, we average the planning times over 3792 different tasks. While the A* planner needs 0.309 sec on average, our approach is marginally better with an average planning time of 0.268 sec. Interestingly, the runtime of our approach is lower on the synthetically generated scenes compared to the Centauro experiment (Table 6.11). This may be due to the fact that the synthetically generated scenes contain randomly placed goals, which may be closer to the initial robot pose than the goals depicted in Figure 6.11.

As shown previously in Figure 6.5 and 6.6, on the less complex 2D grid world task, the A* planner significantly outperforms our approach with respect to planning times. However, for 3D locomotion planning, the runtime of our approach is similar to the runtime of the A* planner. This indicates, that for more complex planning tasks than 3D locomotion planning, learning-based planners, like the one presented in this work, are able to achieve lower planning times than traditional planners. However, the success rate of these planners does not reach the quality of traditional planners for more complex tasks yet.

7. Conclusion

This thesis presents a method how to extend VINs to incorporate multiple levels of abstraction. We evaluated different design decisions and compared our method against VINs and HVINs on 2D grid worlds. Our experiments show that our method improves the applicability of VINs on large grids with coarse obstacle distribution and stabilizes the training process. Our approach successfully reduces the state space size compared to VINs, which is indicated by a reduced graphics memory consumption and lower training times of our approach for large map sizes. This facilitates scaling to larger environments. Although we use abstract environment representations, our method achieves significantly better results with respect to *success* rates and path quality than original VINs. This may be explained since, by providing the highest information density for the *Level-1* area, our method places the focus of the planner on the vicinity of the robot, which is the most important area for determining the next action.

However, due to information loss within higher abstraction levels, we do not reach the performance of VINs on complex environments like 2D mazes. Nevertheless, our experiments show that in general, the information loss can be reduced by the introduction of additional descriptive features. This is indicated by the fact that our method significantly outperforms HVINs on complex 2D mazes. However, HVINs achieve surprisingly good results on large 128×128 grid worlds with coarse obstacle distribution.

Furthermore, we applied our method to 3D robot locomotion tasks. Different methods how to adapt our method to this task have been evaluated and the best one was integrated into a planning pipeline for the Centauro robot. Our system generates optimal or near to optimal paths and even matches the planning time of an A* planner. In the future, the planning time of our approach may even be further decreased by using all approximate state-values from the *Level-1* map to predict a short action sequence instead of a single next action.

For the 2D grid world task, there are huge differences between the planning time of our approach and the planning time of the A* planner, which is significantly faster. However, this difference is reduced for the more complex 3D locomotion planning task and, depending on the scene complexity, both approaches even achieve similar planning times. This confirms our initial assumption that the performance of learning-based planners depends less on the problem complexity than the performance of traditional planners. However, the *success* rates of our method for 3D locomotion planning are significantly lower compared to the *success* rates for the less complex 2D grid world planning tasks. Although our method enables VINs to solve the 3D locomotion planning tasks considered in this work, this points out current limitations of learning-based planners. Thus, the applicability to more complex and higher-dimensional problems is uncertain.

Keeping the state space small, e.g., through abstraction, is essential for achieving good results with planners based on VINs. This is shown by our comparison of three and four abstraction levels for large 128×128 gird worlds. It is also indicated by our evaluation of different methods for the 3D locomotion planning task, where the best results are achieved by using abstract representations for both, position and orientation.

Usually, large parts of the environment map are not needed to predict successful paths. Therefore, it might be interesting to embed a module into the network architecture, which reduces the state space size by predicting a region of interest. Instead of planning on the whole environment map, the planner can subsequently be applied to this region.

To be able to scale to larger and more complex state spaces, it may be interesting to use hierarchical versions of the method proposed in this thesis. On a global low resolution map, subgoals might be determined which are locally connected by the planning system.

All in all, we have increased the applicability of learning-based motion planners to real world problems.

List of Figures

List of Tables

Bibliography

Bagnell, J. A. and J. G. Schneider (2001). “Autonomous helicopter control using reinforcement learning policy search methods”. In: *IEEE International Conference on Robotics and Automation (ICRA).*

Behnke, S. (2003). “Local multiresolution path planning”. In: *Robot Soccer World Cup.* Springer, pp. 332–343.

Bellman, R. (1957). “A Markovian decision process”. In: *Indiana University Mathematics Journal* 6 (4), pp. 679–684.

– (2013 [1957]). *Dynamic programming.* Courier Corporation.

Bojarski, M., D. Del Testa, D. Dworakowski, B. Firner, B. Flepp, P. Goyal, L. D. Jackel, M. Monfort, U. Muller, J. Zhang, et al. (2016). “End to end learning for self-driving cars”. In: *arXiv preprint:1604.07316.*

Dijkstra, E. W. (1959). “A note on two problems in connexion with graphs”. In: *Numerische Mathematik* 1.1, pp. 269–271.

Droeschel, D., M. Schwarz, and S. Behnke (2017). “Continuous mapping and localization for autonomous navigation in rough terrain using a 3D laser scanner”. In: *Robotics and Autonomous Systems* 88, pp. 104–115.

Gupta, S., J. Davidson, S. Levine, R. Sukthankar, and J. Malik (2017). “Cognitive mapping and planning for visual navigation”. In: *arXiv preprint:1702.03920* 3.

Hart, P. E., N. J. Nilsson, and B. Raphael (1968). “A formal basis for the heuristic determination of minimum cost paths”. In: *IEEE Transactions on Systems Science and Cybernetics* 4.2, pp. 100–107.

Kambhampati, S. and L. Davis (1986). “Multiresolution path planning for mobile robots”. In: *IEEE Journal on Robotics and Automation* 2.3, pp. 135–145.

Karaman, S. and E. Frazzoli (2011). “Sampling-based algorithms for optimal motion planning”. In: *The International Journal of Robotics Research* 30.7, pp. 846–894.

Karkus, P., D. Hsu, and W. S. Lee (2017). “QMDP-Net: Deep learning for planning under partial observability”. In: *Advances in Neural Information Processing Systems*, pp. 4694–4704.

Kavraki, L., P. Svestka, J.-C. Latombe, and M. H. Overmars (1996). “Probabilistic roadmaps for path planning in high-dimensional configuration spaces”. In: *IEEE Transactions on Robotics and Automation* 12.4, pp. 566–580.

Khan, A., C. Zhang, N. Atanasov, K. Karydis, V. Kumar, and D. D. Lee (2017). “Memory augmented control networks”. In: *arXiv preprint:1709.05706.*

Khatib, O. (1985). “Real-time obstacle avoidance for manipulators and mobile robots”. In: *IEEE International Conference on Robotics and Automation (ICRA).*

Klamt, T. and S. Behnke (2017). “Anytime hybrid driving-stepping locomotion planning”. In: *IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS).*

– (2018). “Planning hybrid driving-stepping locomotion on multiple levels of abstraction”. In: *IEEE International Conference on Robotics and Automation (ICRA).*

Klamt, T., D. Rodriguez, M. Schwarz, C. Lenz, D. Pavlichenko, D. Droeschel, and S. Behnke (2018). “Supervised autonomous locomotion and manipulation for disaster response with a centaur-like robot”. In: *IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS).*

Krizhevsky, A., I. Sutskever, and G. E. Hinton (2012). “ImageNet classification with deep convolutional neural networks”. In: *Advances in Neural Information Processing Systems (NIPS).*

Kulkarni, T. D., K. Narasimhan, A. Saeedi, and J. Tenenbaum (2016). “Hierarchical deep reinforcement learning: integrating temporal abstraction and intrinsic motivation”. In: *Advances in Neural Information Processing Systems (NIPS).*

LaValle, S. M. (1998). *Rapidly-exploring random trees: a new tool for path planning.*

Lee, L., E. Parisotto, D. S. Chaplot, and R. Salakhutdinov (2018). *LSTM Iteration networks: an exploration of differentiable path finding.* URL: `https://openreview.net/forum?id=ryF9WtyDf`.

Levine, S., C. Finn, T. Darrell, and P. Abbeel (2016). “End-to-end training of deep visuomotor policies”. In: *The Journal of Machine Learning Research (JMLR)* 17.1, pp. 1334–1373.

Loshchilov, I. and F. Hutter (2016). “SGDR: Stochastic gradient descent with warm restarts”. In: *arXiv preprint:1608.03983.*

Mnih, V., K. Kavukcuoglu, D. Silver, A. A. Rusu, J. Veness, M. G. Bellemare, A. Graves, M. Riedmiller, A. K. Fidjeland, G. Ostrovski, et al. (2015). “Human-level control through deep reinforcement learning”. In: *Nature* 518.7540, p. 529.

Niu, S, S. Chen, H. Guo, C. Targonski, M. C. Smith, and J. Kovacevic (2017). “Generalized value iteration networks: life beyond lattices”. In: *Arxiv preprint: 1706.02416.*

Schwarz, M., A. Milan, A. S. Periyasamy, and S. Behnke (2018). “RGB-D object detection and semantic segmentation for autonomous manipulation in clutter”. In: *The International Journal of Robotics Research (IJRR)* 37.4-5, pp. 437–451.

Srinivas, A., A. Jabri, P. Abbeel, S. Levine, and C. Finn (2018). “Universal planning networks”. In: *arXiv preprint:1804.00645.*

Sutton, R. S. and A. G. Barto (1998). *Introduction to reinforcement learning.* Vol. 135. MIT press Cambridge.

Tamar, A., Y. Wu, G. Thomas, S. Levine, and P. Abbeel (2016). “Value iteration networks”. In: *Advances in Neural Information Processing Systems (NIPS).*

Tieleman, T. and G. E. Hinton (2012). *Lecture 6.5-rmsprop: divide the gradient by a running average of its recent magnitude.* Available at `https://www.coursera.`

org/lecture/neural-networks/rmsprop-divide-the-gradient-by-a-running-average-of-its-recent-magnitude-YQHki.

A. Detailed Test Results

Table A.1: Detailed results for the different design choices on 64 × 64 grid worlds. Depicted are five different training runs as well as mean, median, and standard deviation. All values are expressed in percentages.

		Training run							
		0	1	2	3	4	mean	median	std
(Ind. levels, mean padding, weighted loss, single-action)	Success	93.4	94.2	94.0	92.8	97.8	94.44	94.0	1.96
	Accuracy	79.7	80.4	83.2	82.0	82.9	81.64	82.0	1.54
	Path diff	21.68	1.64	1.41	1.39	0.84	1.39	1.41	0.34
(Information flow, mean padding, weighted loss, single-action)	Success	98.8	97.4	95.5	97.5	95.5	96.94	97.4	1.43
	Accuracy	87.8	85.8	81.3	85.3	84.6	84.96	85.3	2.37
	Path diff	0.68	1.15	0.73	0.65	1.14	0.87	0.73	0.25
(Information flow, padding layer, weighted loss, single-action)	Success	93.3	91.5	91.6	96.6	91.4	92.88	91.6	2.22
	Accuracy	82.3	80.4	80.0	86.0	78.5	81.44	80.4	2.89
	Path diff	1.27	2.90	1.67	0.85	1.92	1.72	1.67	0.77
(Information flow, mean padding, unweighted loss, single-action labels)	Success	94.9	96.4	94.7	91.2	93.4	94.12	94.7	1.95
	Accuracy	80.8	86.4	84.4	77.3	83.2	82.42	83.2	3.51
	Path diff	1.96	1.56	1.14	1.65	2.05	1.67	1.65	0.36
(Information flow, mean padding, weighted loss, single-action) adapted sampling	Success	99.3	96.5	94.0	93.6	94.2	95.52	94.2	2.40
	Accuracy	88.2	86.4	85.8	86.9	84.9	86.44	86.4	1.23
	Path diff	0.63	0.70	0.80	0.82	0.70	0.73	0.70	0.08
(Information flow, mean padding, weighted loss, multi-action)	Success	86.5	87.3	47.8	49.2	89.2	72.00	86.5	21.48
	Accuracy[1]	94.7	94.8	87.6	89.0	94.3	92.08	94.3	3.49
	Path diff	0.36	0.55	1.04	0.88	0.17	0.60	0.55	0.36

[1]In the case of multi-action labels, the accuracy does not measure whether the action predicted by the network is the one predicted by the expert planner, but whether the predicted action is one of the optimal next actions.

Table A.2: Detailed results on 2D mazes. Depicted are five different training runs as well as mean, median, and standard deviation. All values are expressed in percentages.

			Training run							
			0	1	2	3	4	mean	median	std
16 × 16	VIN	Success	73.9	99.6	99.8	99.8	99.3	94.48	99.6	11.51
		Accuracy	77.4	99.3	99.3	99.7	96.4	94.42	99.3	9.61
		Path diff	1.70	0.06	0.06	0.02	0.70	0.51	0.06	0.72
	HVIN	Success	83.4	89.9	90.0	84.1	89.7	87.42	89.7	3.36
		Accuracy	83.2	89.9	89.8	84.2	88.9	87.20	88.9	3.24
		Path diff	1.61	0.88	0.87	1.33	1.11	1.16	1.11	0.31
	AVIN	Success	82.6	83.6	82.10	83.9	82.8	83.00	82.8	0.74
		Accuracy	82.6	83.7	81.5	82.8	82.0	82.52	82.6	0.83
		Path diff	2.19	1.74	2.04	2.30	1.99	2.05	2.04	0.21
32 × 32	VIN	Success	25.8	97.2	91.8	97.6	98.1	82.1	97.2	31.58
		Accuracy	52.1	95.9	87.3	95.5	97.2	85.60	95.5	19.13
		Path diff	2.52	0.22	0.90	0.19	0.57	0.88	0.57	0.96
	HVIN	Success	45.3	55.2	47.8	47.1	47.3	48.54	47.3	3.84
		Accuracy	66.7	73.5	69.6	71.1	68.8	69.94	69.6	2.55
		Path diff	2.67	1.51	1.91	1.83	2.20	2.02	1.91	0.44
	AVIN	Success	73.6	74.6	66.5	70.8	71.0	71.30	71.0	3.14
		Accuracy	84.3	84.5	79.9	82.0	83.1	82.76	83.1	1.89
		Path diff	1.00	0.95	1.28	1.19	1.10	1.10	1.10	0.14
64 × 64	VIN	Success	90.8	28.7	90.4	87.5	92.7	78.02	90.4	27.63
		Accuracy	91.1	65.7	85.9	85.9	94.3	84.58	85.9	11.15
		Path diff	1.32	2.78	1.70	1.27	0.36	1.49	1.32	0.87
	HVIN	Success	15.8	14.2	12.3	14.6	14.2	14.22	14.2	1.26
		Accuracy	60.2	58.5	57.8	59.3	58.3	58.82	58.5	0.94
		Path diff	2.16	2.28	1.65	1.93	1.91	1.99	1.93	0.24
	AVIN	Success	61.6	56.0	52.1	51.3	64.3	57.06	56.0	5.74
		Accuracy	80.5	80.9	80.2	78.2	83.3	80.62	80.5	1.82
		Path diff	0.61	0.47	0.58	0.67	0.58	0.58	0.58	0.07

Table A.3: Detailed results on 2D mazes using path generation with history. Depicted are five different training runs as well as mean, median, and standard deviation. All values are expressed in percentages.

			Training run							
			0	1	2	3	4	mean	median	std
16×16	VIN	Success	78.6	99.7	99.8	99.8	99.4	95.46	99.7	9.43
		Path diff	7.67	0.11	0.60	0.03	0.77	1.84	0.60	3.28
	HVIN	Success	86.6	93.4	93.0	87.8	92.1	90.58	92.1	3.15
		Path diff	4.76	4.36	4.98	5.59	3.75	4.69	4.76	0.69
	AVIN	Success	86.5	88.2	86.0	88.0	87.6	87.26	87.6	0.96
		Path diff	5.82	6.44	5.65	6.59	6.74	6.25	6.44	0.48
32×32	VIN	Success	27.7	97.5	92.3	97.9	98.2	82.72	97.5	30.85
		Path diff	7.66	0.34	1.09	0.35	0.58	2.00	0.58	3.18
	HVIN	Success	53.6	61.3	55.5	53.9	52.7	55.40	53.9	3.45
		Path diff	12.11	6.74	9.84	8.60	7.63	8.98	8.60	2.09
	AVIN	Success	78.8	79.1	69.2	74.4	75.6	75.42	75.6	4.02
		Path diff	3.85	3.56	2.86	3.03	3.40	3.34	3.40	0.40
64×64	VIN	Success	91.7	31.6	91.4	88.5	93.0	79.24	91.4	26.68
		Path diff	1.45	4.43	1.88	1.50	0.42	1.94	1.50	1.50
	HVIN	Success	20.9	18.8	13.6	17.2	15.8	17.25	17.2	2.78
		Path diff	13.30	12.80	4.33	7.20	5.16	8.56	7.20	4.24
	AVIN	Success	64.1	58.6	53.5	54.6	66.3	59.42	58.6	5.66
		Path diff	1.23	1.25	1.05	1.98	1.23	1.35	1.23	0.36

Table A.4: Detailed results on 2D grid worlds. Depicted are five different training runs as well as mean, median, and standard deviation. All values are expressed in percentages.

			Training run							
			0	1	2	3	4	mean	median	std
32×32	VIN	Success	77.4	94.9	93.6	97.8	94.9	91.72	94.9	8.15
		Accuracy	62.8	83.6	81.8	90.0	83.7	80.38	83.6	10.31
		Path diff	5.81	1.51	2.09	0.62	2.37	2.48	2.09	1.98
	HVIN	Success	95.1	93.9	95.0	91.8	93.4	93.84	93.9	1.35
		Accuracy	81.6	81.5	81.0	78.0	78.3	80.08	81.0	1.78
		Path diff	1.56	2.93	1.42	3.16	2.07	2.23	2.07	0.79
	AVIN	Success	97.2	97.3	96.6	96.6	96.5	96.84	96.6	0.38
		Accuracy	85.6	83.6	82.6	82.8	82.4	83.40	82.8	1.31
		Path diff	1.84	1.10	1.61	2.02	2.07	1.73	1.84	0.39
64×64	VIN	Success	34.8	83.2	79.3	82.9	79.7	71.98	79.7	20.86
		Accuracy	47.2	74.9	73.1	79.2	76.0	70.08	74.9	12.98
		Path diff	5.74	2.12	2.56	1.87	2.42	2.94	2.42	1.59
	HVIN	Success	88.5	91.5	89.8	93.1	71.2	86.82	89.8	8.90
		Accuracy	76.6	78.5	80.1	77.9	74.0	77.42	77.9	2.29
		Path diff	2.68	2.12	2.20	1.48	4.28	2.55	2.20	1.06
	AVIN	Success	98.8	97.4	95.5	97.5	95.5	96.94	97.4	1.43
		Accuracy	87.8	85.8	81.3	85.3	84.6	84.96	85.3	2.37
		Path diff	0.68	1.15	0.73	0.65	1.14	0.87	0.73	0.25
128×128	VIN	Success	16.9	46.5	35.1	25.3	34.0	31.56	34.0	11.13
		Accuracy	49.9	55.9	51.5	57.4	65.1	55.96	55.9	5.96
		Path diff	6.42	7.18	16.85	9.41	2.43	8.46	7.18	5.33
	HVIN	Success	89.0	79.3	75.7	89.6	82.6	83.24	82.6	6.05
		Accuracy	79.0	78.1	75.5	73.5	76.4	76.50	76.4	2.17
		Path diff	2.08	2.66	2.51	1.67	1.55	2.09	2.08	0.49
	HVIN 4 Levels	Success	85.2	85.3	82.6	86.3	80.6	84.00	85.2	2.34
		Accuracy	78.9	78.7	79.7	76.4	76.5	78.04	78.7	1.50
		Path diff	2.31	2.42	3.16	2.34	3.76	2.80	2.42	0.64
	AVIN	Success	82.1	75.4	77.2	88.4	78.6	80.34	78.6	5.13
		Accuracy	81.7	76.3	79.7	86.2	80.4	80.86	80.4	3.59
		Path diff	2.86	2.21	2.95	1.56	2.81	2.48	2.81	0.59
	AVIN 4 Levels	Success	38.2	94.6	94.8	83.9	79.2	78.14	83.9	23.33
		Accuracy	72.6	85.5	84.6	77.5	80.1	80.06	80.1	5.30
		Path diff	1.51	1.43	1.22	2.37	2.54	1.81	1.51	0.60

Table A.5: Detailed results for the 3D locomotion planning task. Depicted are five different training runs as well as mean, median, and standard deviation. All values are expressed in percentages.

		Training run							
		0	1	2	3	4	mean	median	std
Full 3D	Success	19.6	16.3	26.3	21.9	23.7	21.56	21.9	3.83
	Accuracy	66.0	65.0	63.0	63.2	63.6	64.16	63.6	1.29
	Path diff	2.13	0.87	2.33	1.30	1.36	1.60	1.36	0.61
Independent orientations	Success	64.0	36.4	54.8	61.4	62.2	55.76	61.4	11.37
	Accuracy	67.4	54.9	63.8	66.3	66.8	63.84	66.3	5.18
	Path diff	1.12	8.28	0.92	1.06	0.78	2.43	1.06	3.27
Independent orientations (pretrained)	Success	59.9	52.9	43.3	62.9	52.3	54.26	52.9	7.62
	Accuracy	67.4	64.5	51.3	67.1	63.9	62.84	64.5	6.63
	Path diff	0.99	1.61	8.35	1.35	0.99	2.66	1.35	3.19
2D input	Success	68.0	67.6	67.2	67.5	67.6	67.58	67.6	0.29
	Accuracy	68.2	65.5	68.0	66.9	65.1	66.74	66.9	1.41
	Path diff	1.62	1.87	1.57	2.06	2.11	1.85	1.87	0.25
2D input (without lr scheduler)	Success	65.5	64.3	64.7	62.5	63.9	64.18	64.3	1.11
	Accuracy	66.0	67.2	69.3	68.0	64.8	67.06	67.2	1.74
	Path diff	3.79	2.21	2.41	2.37	2.79	2.71	2.41	0.64
2D input (abstract orientations)	Success	73.5	74.1	71.3	73.5	76.2	73.72	73.5	1.75
	Accuracy	71.8	71.0	69.9	70.6	72.4	71.14	71.0	0.98
	Path diff	1.61	1.77	1.44	1.47	1.96	1.65	1.61	0.22