Institute of Parallel and Distributed Systems

University of Stuttgart
Universitätsstraße 38
D–70569 Stuttgart

Bachelorarbeit

Towards Learners that Plan: Integrating Trainable Planning Modules for Data-Efficient Learning

Tim Schneider

Course of Study:	Informatik
Examiner:	Ph.D. Daniel Hennes
Supervisor:	Hung Ngo
Commenced:	November 3, 2017
Completed:	May 3, 2018

Abstract

Learning is one of the most important abilities of intelligent adaptive agents. The generalization capability and training efficiency of learning algorithms depend heavily on the abstract representations acquired. Planning, on the other hand, allows agents to anticipate the future consequences of their actions so as to act optimally at the now. The action-contingent predictive features generated by planning modules thereby provide a good abstract representation constituting the current state of the agent. From this insight, this thesis aims to integrate trainable planning modules for data-efficient learning in sequential decision making and manipulation problems, ranging from Go game to real-world robotic AI. Specifically, this thesis will investigate the effectiveness of such approach by trying to solve the key questions of (1) how to integrate planning modules into deep learning frameworks so as to train the whole system from data, and (2) how to exploit predictive, but possibly inaccurate, abstract features from planning modules to guide the learning process.

The main contributions of this thesis are to answer these questions within a broad literature survey and incorporate the ideas in an algorithm that can be applied to learn to plan in visual navigation tasks in a completely unsupervised manner.

Contents

List of Figures

1 Introduction

Most recently, Reinforcement Learning algorithms have been successfully combined with deep learning techniques to solve a wide field of challenging problems. These range from playing video games at super-human level [MKS+15] over robotic manipulation tasks up to beating the world-champion in Go [SHM+16], which was considered as a grand challenge to AI for years.

But in order to leverage the power of deep learning techniques, a lot of data is needed. This limits the algorithms to domains for which either data is available or can be generated. Additionally, the generalization capabilities of learning algorithms heavily depend on the abstract representations acquired. Considering for instance a learned value function for a single task, it needs to be retrained from scratch when facing another task in the same environment. Planning, on the other hand, allows an intelligent agent to reason about future consequences of actions. This opens great generalization capabilities. Also, for most planning algorithms no additional real experience is needed to act optimally at the now. Behaviour can be improved solely with simulated experience and computational power.

Thereby planning algorithms generate features that incorporate predictive future information in terms of values, future occupancy of features or just unrolled trajectories of states. Therewith the features represent a kind of forecast and so provide a good abstract representation constituting the current state of the agent. The main goal of this thesis is to exploit this better abstract representations to achieve more data-efficient learning.

Additionally, with recent success of deep learning methods more specialized network structures have been developed. So, a branch of methods follows the idea of expressing planning computations in a fully differentiable way and embed it in a neural network structure. This is called a planning module and allows end-to-end training with data. Thereby planning modules also provide predictive features as outlined before. This thesis aims to effectively combine these ideas.

To do so, we will give broad overview over existing approaches to integrate differentiable planning modules in chapter 3. Therewith we address the question of how to integrate planning modules into deep learning frameworks so as to train the entire system from data. Next we'll continue comparing these various approaches with respect to the way predictive features from planning modules are exploited in chapter 4. Within that we try to answer the question of how these predictive, but possibly inaccurate, abstract features are used to guide the learning process.

In the following we incorporate the ideas from literature in a *Planning for Learning* framework. It is introduced in detail in chapter 4. As a result, we extend existing approaches to RL based on latent representations acquired from unsupervised learning techniques. The approach called *Unsupervised Planning for Learning* learns an abstract model with multitask learning and *Hindsight Experience Replay*. Additionally, we incorporate the learned values to get a better representation of the current

state to finally solve visual navigation tasks. The experiments conducted in chapter 5 show the effectiveness of this approach to acquire better abstract representations in order to leverage more data-efficient learning.

So, to conclude, the main contributions of this thesis are: (1) A broad literature survey on existing approaches to integrate differentiable planning modules and exploitation of predictive abstract features and (2) a *Unsupervised Planning for Learning* algorithm, which addresses to learn visual navigation in a completely unsupervised manner.

But first in chapter 2 background information and some related work, that is needed throughout this thesis, will be introduced.

2 Background

2.1 Reinforcement Learning

Reinforcement Learning (RL) is a subfield of Machine Learning. RL means to learn from interactions with the environment. In such a framework there is only supervision provided by rewards. In general, this kind of supervision is sparse. In contrast to *Supervised Learning* there is no ground truth data for actions to take to learn from. A goal-directed *agent* is trying to find a policy that maximizes a reward signal [SB98].

2.1.1 Markov Decision Process

Mathematically a RL problem can be described as a *Markov Decision Process (MDP)* which is the five-tuple $\langle \mathcal{S}, \mathcal{A}, \mathcal{P}, \mathcal{R}, \gamma \rangle$. Thereby $\mathcal{S}$ is a set of states on which the Markov Property $\forall t : P(S_{t+1}|S_t) = P(S_{t+1}|S_1, \cdots, S_t)$ holds. $\mathcal{A}$ is the set of possible actions. In each state s_t the agent takes an action a_t and observes a reward r_t and a consecutive state s_{t+1}. The state transition distribution $\mathcal{P}(s_{t+1}, s_t, a_t) = P(s_{t+1}|s_t, a_t)$ and reward distribution $\mathcal{R} = P(r_t|s_t, a_t)$ are typically not given to the agent. The agent's task is to maximize the discounted cumulative reward, which is also called *return* $G_t = \sum_{i=t}^{\infty} \gamma^{i-t} r_i$.

2.1.2 Value Function

A value function is used to estimate the expected return under some policy π starting in a given state. Ordinarily used are the *state value function* $V^\pi(s)$ and the *state action value function* $Q^\pi(s, a)$.

$$Q^\pi(s, a) = \mathbb{E}\Big[\sum_{i=t}^{\infty} \gamma^{i-t} r_i | s_t = s, a_t = a \Big] \tag{2.1}$$

$$V^\pi(s_t) = \mathbb{E}\Big[\sum_{i=t}^{\infty} \gamma^{i-t} r_i | s_t = s] \tag{2.2}$$

For every MDP there exists an optimal policy $\pi*$ with the value function $V^*(s) \geq V(s)\ \forall V, s$. And for such an optimal value function the *Bellman Optimality Equation* holds:

$$Q^*(s, a) = \mathcal{R}(s, a) + \gamma \sum_{s' \in \mathcal{S}} \mathcal{P}(s', s, a) \cdot \max_{a'} Q^*(s', a') \tag{2.3}$$

The optimal policy $\pi^*(s) = \underset{a}{\operatorname{argmax}}\ Q^*(s, a)$ can be extracted by acting greedily with respect to the optimal state action value function [Bel57].

2.1.3 Classification of RL methods

Sutton and Barto [SB98] classify RL methods in two different classes: *model-free RL* and *model-based RL*. Both approaches to RL have strengths and weaknesses. A promising direction is to combine aspects from both approaches. The integration of differentiable planning modules is one possibility to do so.

Model-free RL

Given experience as data $D = \{(s_t, a_t, r_t, s_{t+1})\}$ a model-free RL method directly learns a *value function* estimating the value for each state (and action). The actual policy can be derived by acting greedily with respect to the value function as shown before. The principle is depicted in below.

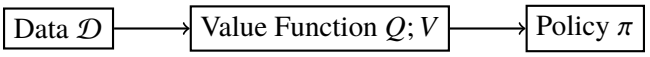


Figure 2.1: Principle of model-free RL

A common method to learn a value function is called *Temporal Difference Learning* or *TD-Learning*. The key observation therefore is that a state-value function, as described in 2.1, represents the expectation under some policy π. This expectation can be estimated by the incremental mean $V^\pi(s_t) = V^\pi(s_t) + \alpha(G_t - V^\pi(s_t))$ of returns G_t. Thereby the return is an unbiased sample of the true distribution of values. This ground-truth distribution is unknown and intended to learn. To extend this idea to actual *TD-Learning*, the true return G_t is replaced by the *TD-target* $r_t + \gamma V(s_{t+1})$, which is a biased estimate of the return with lower variance. This is called *bootstrapping* and allows learning from every step taken in the environment. Equation 2.4 shows the resulting update rule.

$$V^\pi(s_t) = V^\pi(s_t) + \alpha\big(r_t + \gamma V^\pi(s_{t+1}) - V^\pi(s_t)\big) \tag{2.4}$$

In general rewards are sparse in RL. So the crucial action for receiving a reward, after long sequence of actions, might be several steps in the past. This problem is called *credit-assignment problem*. TD-Learning can be extended with *eligibility traces* $E(s)$ to also update the values of previously visited states for better credit assignment. This results in the *TD(λ) algorithm* 2.5.

$$V^\pi(s) = V^\pi(s) + \alpha\delta_t E_t(s) \tag{2.5}$$
$$\delta_t = r_t + \gamma V^\pi(s_{t+1}) - V^\pi(s_t) \tag{2.6}$$
$$E_t(s) = \gamma\lambda E_{t-1}(s) + \mathbb{1}(s_t = s) \tag{2.7}$$

Often it is desired to learn *off-policy*, i. e. to learn about the optimal policy while executing an exploration policy.

This behaviour is achieved by *Q-Learning*. The update rule for the state action value function $Q(s, a)$ from data $\langle s, a, r, s' \rangle$ is given in equation 2.8.

$$Q(s, a) = Q(s, a) + \alpha\big(r_t + \gamma \max_{a'} Q(s', a') - Q(s, a)\big) \tag{2.8}$$

To learn off-policy is helpful to train neural function approximators for $Q(s, a)$. Mnih et al. [MKS+15] use a *replay buffer* to collect all the experience gathered from interaction with the environment. For learning phase, it is possible to sample a random *mini-batch* from the replay buffer to perform optimization. This stabilizes the learning. Off-policy learning ensures correct learning, even though the data in the buffer was gathered with a different previous policy than the current one.

Model-based RL

In contrast to all the aforementioned model-free approaches model-based RL methods make use of an environment model, as the name already implies. In an MDP $\mathcal{P}$ and $\mathcal{R}$ together sufficiently describe such a model. Henceforth, these are also referred as *system dynamics* and *environment model*. Typically, $\mathcal{P}, \mathcal{R}$ are either given or learned from data in a supervised manner.

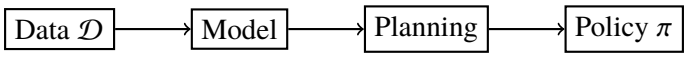


Figure 2.2: Principle of Model-based RL.

This model is afterwards used for planning to come up with a policy. Different planning methods are discussed in 2.2.

Policy Search and Actor Critic Methods

A different branch of RL methods approximate the policy $\pi_\theta(s, a)$ directly, without the intermediate step of approximating the value function. To optimize such a policy network, the *policy gradient* $\nabla_\theta \pi_\theta(s, a) = \pi_\theta(s, a) \nabla_\theta \log \pi_\theta(s, a)$ is needed. For an objective function J the following equation holds:

$$\nabla_\theta J(\theta) = \mathbb{E}\Big[\nabla_\theta \log \pi_\theta(s, a) Q^{\pi_\theta}(s, a)\Big] \tag{2.9}$$

For the algorithm called *REINFORCE* the unknown q-value $Q^{\pi_\theta}(s, a)$ is estimated with the actual return from episodes of experience v_t. This leads to the update rule in equation 2.10 with learning rate α.

$$\Delta\theta_t = \alpha \nabla_\theta \log \pi_\theta(s, a) v_t \tag{2.10}$$

But this sample has a high variance. So the idea of *Actor-Critic Methods* is to use a *critic* to have a biased estimate with lower variance for the q-value in equation 2.9. The *actor* updates its weights in the direction suggested by the critic. By subtracting the state value function $V^{\pi_\theta}(s)$ as a baseline, hence estimating the advantage function $A^{\pi_\theta}(s, a) = Q^{\pi_\theta}(s, a) - V^{\pi_\theta}(s)$ for this particular action, and performing asynchronous updates. This leads to a state-of-the-art algorithm called *Asynchronous Advantage Actor Critic (A3C)* [MBM+16].

2.2 Planning

Planning is the ability to anticipate the future consequences of actions, by reasoning with an understanding about how the environment works. So in contrast to learning algorithms, *planning* assumes access to the system's dynamics. These could be either given or learned from experience data as mentioned before. In general any algorithm that makes use of such a model to predict an optimal policy can be viewed as an instance of a planning algorithm.

2.2.1 Optimization Problem

From a different angle of view, planning could be viewed as *constrained optimization* (see equation 2.11). The task is to find an action a_t such that it maximizes the cumulative future reward, under the constraint that the environment dynamics $s_{i+1} = f(s_i, a_i)$ holds for all the states encountered while planning [PGDL18].

$$a_t = \underset{a_{t:t+T}}{\operatorname{argmax}} \sum_{i=t}^{t+T} r(s_i, a_i)\,, \text{ where } s_{i+1} = f(s_i, a_i)\ \forall i \in \{t, \cdots, t+T-1\} \tag{2.11}$$

2.2.2 Value Iteration

A rather simple planning algorithm is directly derived from *Bellman Equation* (2.3). The method is called *Value Iteration (VI)* and iteratively applies the following equation to arbitrary initialized values. It's proven that this method converges to an optimal value function V^*, for that the *Bellman optimality* holds, and hence is an equilibrium point of the iteration.

$$\forall s : V_{k+1}(s) = \max_a \left[\mathcal{R}(s, a) + \gamma \sum_{s'} \mathcal{P}(s', \pi(s), s) \cdot V_k(s')\right] \tag{2.12}$$

VI is an instance of *Dynamic Programming* methods, that were first introduced by Bellman [Bel57].

2.2.3 Search

Search is a special kind of planning method, that focuses the computational power on that state the agent is currently in. Starting from the current state, the planning algorithm spans a tree with edges corresponding to actions $\mathcal{A}$ and tree nodes standing for the states $\mathcal{S}$ of the MDP. This tree can be grown by iteratively applying a (transition) model.

This method is also called *lookahead planning*, because with each level of the tree different future trajectories of states are unrolled.

Weber et al. [WRR+17] also refer to this kind of planning as *'Imagination'*. This terminology is based on the idea that an agent uses a model, which is basically a summary of its 'understanding' about the environment's dynamics. And throughout planning this understanding is used to 'imagine' different potential future scenarios [1].

Monte Carlo Tree Search

Exhaustive search like *Depth First Search* or *Breadth First Search* is not feasible for problems with large state and action spaces. This is mostly true for real world problems, e.g. the number of states in the game of Go roughly exceeds the number of atoms in the universe [SHM+16].

Monte Carlo Tree Search (MCTS) addresses this problem. The tree is grown by performing so called *rollouts* and propagating their results throughout the tree. A *rollout* is basically a trajectory of random samples in decision space, that are consecutively applied to a state. Figure 2.3 shows the basic steps performed in an MCTS algorithm: (i) Selection , (ii) Expansion , (iii) Simulation and (iv) Backpropagation .

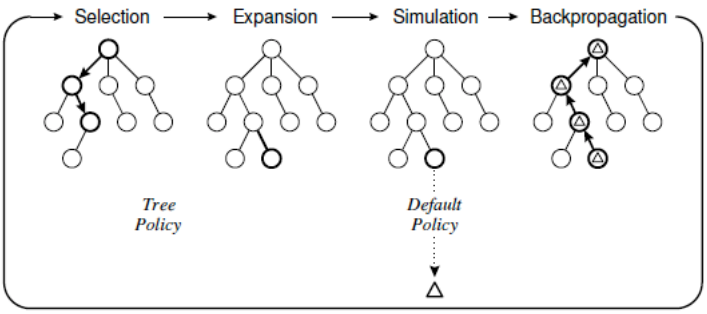


Figure 2.3: Overview over basic steps of the MCTS method [BPW+12].

During *Selection* (i) a node to expand is chosen with a *tree policy*. Afterwards the selected node is *expanded* (ii) by applying the model once and creating a new node for the resulting state. From this new state a *Simulation* (iii) is started, which corresponds to the rollout mentioned before. Finally, the return Δ obtained from the simulation is *backpropagated* (iv) through the tree back to the root. Therewith a state action value $Q(s, a)$ is estimated for each node in the tree [BPW+12].

One can balance exploration and exploitation in the tree by estimating some upper confidence for values. *Upper Confidence Tree (UCT)* is a MCTS algorithm with a tree policy, that selects actions a, which maximizes the *Upper Confidence Bound (UCB)* [GS07].

$$Q^{+}_{\mathrm{UCT}}(s, a) = Q_{\mathrm{UCT}}(s, a) + c\sqrt{\frac{\log n(s)}{n(s, a)}} \tag{2.13}$$

In this context n counts the number of visits of a particular state or state-action pair. The constant c balances the amount of exploration. The tree policy is derived by acting greedy w.r.t. $Q^{+}_{\mathrm{UCT}}(s, a)$ [GS07].

[1] Although it gives an intuition, we try to avoid this terminology in the following, as it is not precise and encourages humanization of algorithms.

2.3 Combine Learning and Planning

Planning provides great generalization possibilities and allows to improve a policy solely with additional computing power and simulated experience. On the other hand, it requires a model, what can be crucial for many domains. Planning with inaccurate model is likely to compound errors. Learning addresses most of these problems, but in general requires a lot of data. So in order to take the benefits from both worlds, there are several ideas on how to effectively combine learning and planning. Most relevant approaches are outlined in the following.

2.3.1 Dyna-Q

The idea of the *Dyna* architecture is to enhance the real experience samples from a true MDP $M = \langle \mathcal{S}, \mathcal{A}, \mathcal{P}, \mathcal{R}, \gamma \rangle$ with *simulated experience* from a learned model $\langle \hat{\mathcal{P}}, \hat{\mathcal{R}} \rangle$.

The *Dyna-Q algorithm* learns a Q-value function with use of such simulated experience. The pseudo code 2.1 shows how learning and planning with the simultaneously learned model are integrated.

Algorithm 2.1 Pseudo Code of Dyna-Q algorithm

initialize $Q(s,a)$, and $\langle \hat{\mathcal{P}}, \hat{\mathcal{R}} \rangle \;\; \forall\, s \in \mathcal{S}, a \in \mathcal{A}$
loop
  $s \leftarrow$ current state
  $a \leftarrow \epsilon$-greedy(Q, S)
  $r, s' \leftarrow$ execute action A in the environment
  $Q(s,a) \leftarrow Q(s,a) + \alpha[r + \gamma \max_{a'} Q(s',a') - Q(s,a)]$
  $\langle \hat{\mathcal{P}}, \hat{\mathcal{R}} \rangle \leftarrow r, s'$ // Update model with the new transition sample
  for $i \leftarrow 1, n$ **do**
    $s, a \leftarrow$ random
    $r, s' \leftarrow \langle \hat{\mathcal{P}}, \hat{\mathcal{R}} \rangle$
    $Q(s,a) \leftarrow Q(s,a) + \alpha[R + \gamma \max_{a'} Q(s',a') - Q(s,a)]$
  end for
end loop

Given a transition sample $\langle s, a, r, s' \rangle$, the Dyna-Q first updates the value function Q and its model $\langle \hat{\mathcal{P}}, \hat{\mathcal{R}} \rangle$. Afterwards n additional training iterations are performed with transitions, that are sampled from the model.

Data as a model

When the model $\langle \hat{\mathcal{P}}, \hat{\mathcal{R}} \rangle$ just stores every observed transition, it is equivalent to a replay buffer. In that case the model is the set of data seen so far. This data is an unbiased sample from the true transition function $\langle s, a, s' \rangle \sim \mathcal{P}(s', s, a)$ and the true reward function $r \sim \mathcal{R}(s, a)$. By storing all the observed samples from that ground truth distributions, one approximates those. Hence, sampling a random mini batch from the replay buffer corresponds to sampling an approximate model [Lin92] [MKS+15].

2.3.2 Multitask Learning

Often it is desired to learn for multiple task. Learning multiple tasks simultaneously forces a neural network to learn good representations in the hidden layer. Thereby a network can transfer knowledge across tasks. This is also known as pressure of multitask learning. Networks that are trained for multiple tasks learn a representation that captures *underlying factors*, that are relevant for the tasks [BCV13].

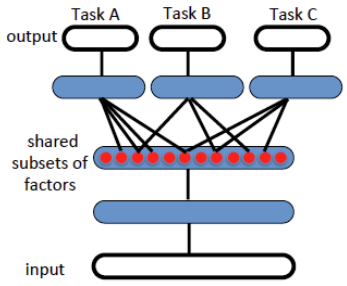


Figure 2.4: This illustration is taken from [BCV13]. It shows the idea of sharing underlying factors in a hidden layer to achieve knowledge transfer across tasks.

The pressure of multitask learning forces a network to acquire good abstractions. This can be considered as indirectly learning a model for the domain.

One approach of learning the value functions for several tasks simultaneously is to use *Universal Function Approximators (UVFA)*. The idea is to learn a Q-function, that is conditional on the goal g to achieve. So the value not only depends on the current state s and action a, but also on the goal [SHGS15].

$$Q^{\pi}(s, a, g) = \mathbb{E}\big[R_t | s_t = s, a_t = a, g\big] \tag{2.14}$$

2.3.3 Learning and Planning in AlphaGo

AlphaGo was the first algorithm that was able to beat a professional human player in the game of Go back in 2015. Therefore, it combines MCTS planning with model-free learning methods.

It uses a learned value function and policy network to *truncate the search tree*. The tree's breadth is reduced with a learned prior probability $p_{\sigma}(a|s)$ for actions. The depth of the tree is reduced with a learned value function $V(s)$ to evaluate board positions s. Additionally, a better default policy $p_{\pi}(a|s)$ is used to guide the search towards more promising directions. $V(s)$, $p_{\pi}(a|s)$ and $p_{\sigma}(a|s)$ are trained with supervised learning from expert data and with RL during self-play [SHM+16].

From a different view *AlphaGo* uses a model-free value function as a rough global estimate, which is then locally refined for the current state s by the online *MCTS* planner. During planning this method does not require any new real experience, as a simulator can generate infinite simulated experience only with computational power [FRIW17].

2.3.4 Learning and Planning in AlphaZero

Silver et al. [SSS+17] improve the original *AlphaGo* algorithm, as depicted in section 2.3.3, to use no domain specific knowledge. With this change the algorithm called *AlphaZero* can be successfully applied to other domains than Go like Chess [SHS+17].

AlphaGo uses domain specific knowledge in several forms: (i) large dataset of human expert play, (ii) handcrafted features with common game patterns, (iii) game rules and symmetry. AlphaZero defines the underlying Reinforcement Learning problem as a sequence of well-defined supervised learning problems. Therewith planning computations performed by MCTS can be viewed as a powerful policy improvement operation. This improved policy acts as a slightly better teacher by generating data for a neural network to learn in a supervised manner. Though the learner imitates the teacher and like in AlphaGo the learned value and policy guides the tree search. Starting from completely random play, it can iteratively achieve superhuman performance solely by RL and self-play [SHS+17].

In contrast to the implication of the term 'zero' in its name, the algorithm still uses domain knowledge in form of the game rules. By knowing the game rules the agent is equipped with a perfect simulator for that domain. This is a very strong assumption and is not available in other domains with rather complex system dynamics. So the algorithm still uses domain specific knowledge.

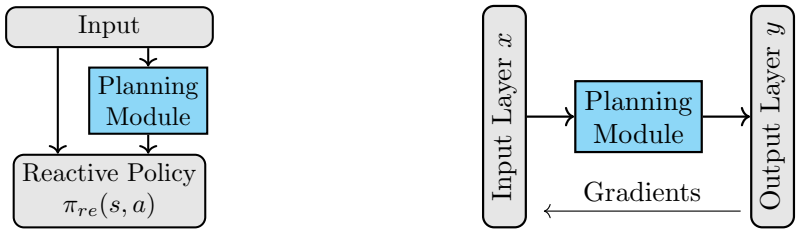
2.3.5 Integrating Differentiable Planning Modules

This is the approach investigated further by this thesis. The basic idea is to integrate differentiable planning modules in a neural network architecture such that the entire system can be trained from data. Therewith learning and planning is combined in a way that the planning computation is directly integrated inside the neural learner.

3 Integrating Differentiable Planning Modules

Many network architectures in deep RL are inspired from classification tasks. They basically view the problem as classifying states s with 'action labels' a by some trained neural network architecture $\pi(a|s)$. But an MDP with its sequential nature is inherently different from such a one-step decision-making and requires some kind of planning [TLA16].

The idea is to design networks that reflect this sequential nature of decision-making. Tamar et al. [TLA16] called such networks, as described above, *reactive policies* $\pi_{\text{re}}(s, a)$. With this they proposed a new network structure that embeds explicit planning computations inside a *planning module* to enhance reactive policies. The general principle is depicted in figure 3.1a.



(a) Enhancing reactive policies with embedded planning modules.

(b) Principle of integrating differentiable planning modules for end-to-end training

Figure 3.1: Integration of Differentiable Planning Modules

A *planning module* is a neural network structure that performs a planning computation. A key observation is that many planning algorithms can be described in a *differentiable* way. Being differentiable means that gradients with respect to the network's weights are defined and can be computed. A desired property of such a network with integrated planning module is to be trained *end-to-end* with data.

End-to-end training means that one directly learns a mapping $x \mapsto y$ from the *input layer* x to the outputs y of the network, as it is depicted in figure 3.1b. The mapping is optimized with a loss function $\mathcal{L}(y, \theta)$, where θ are the weights of the network. Optimization is typically performed with gradient descent by the *Backpropagation algorithm*. Thereby the gradients are propagated back from the output layer to the input layer, as indicted by the arrow in the figure [Roj13].

An overview over different works that follow this approach is shown in figure 3.2. We categorize these methods by the way the state space is represented. A branch of methods, as depicted in section 3.1, encode states sparsely. In contrast, a different branch of methods (section 3.4) prefers dense encoding of states in latent variables. In the following all these different approaches to integrate trainable planning modules are discussed and compared in detail.

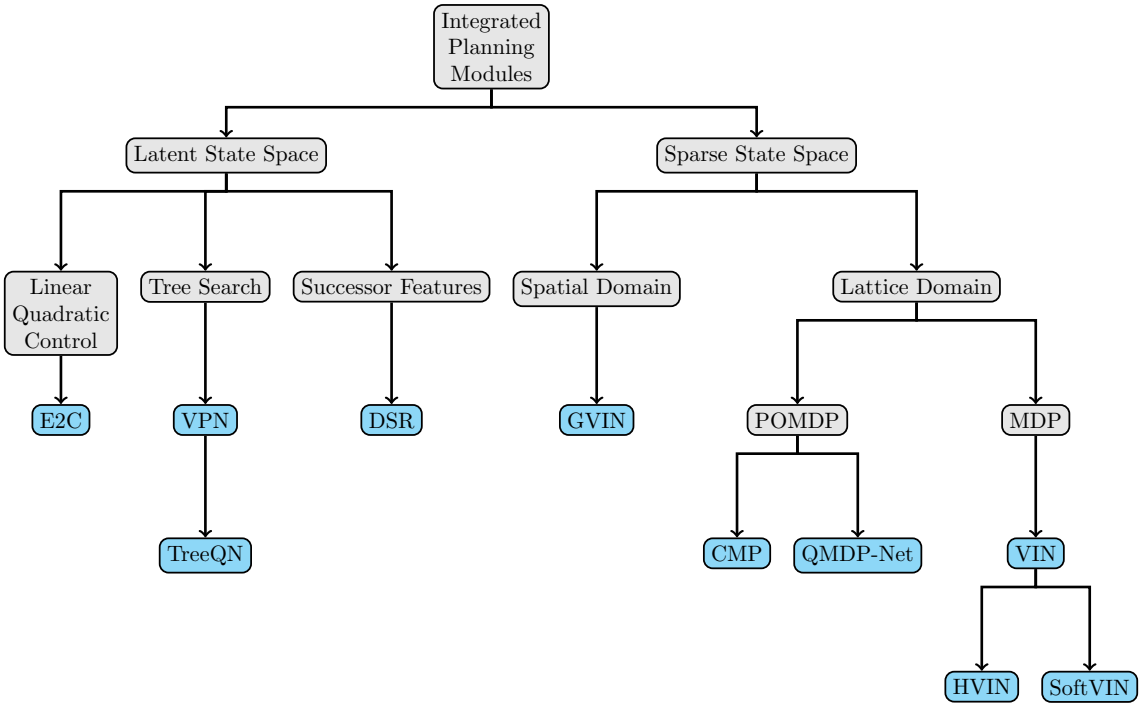


Figure 3.2: Overview over different approaches that integrate differentiable planning modules: Nodes that are indicating a specific algorithm are coloured blue.

3.1 Spatial State Space

A first branch of methods performs planning over a spatial state space. This means that there exists a spatial relationship among states.

s_6	s_7	s_8
s_3	s_4	s_5
s_0	s_1	s_2

Figure 3.3: Spatial State Tensor

All these methods have in common that they represent the entire state space as a tensor. An example is depicted in figure 3.3. Thereby each cell in the tensor corresponds to a particular state variable or stores a value that belongs to that state. We refer to this as a *sparse encoding* of state variables. So topology or distance among state variables is defined by the spatial shape of the tensor. In the example is s_1 next to s_4.

We consider the task is to solve an MDP $M = \langle S, A, R, P, \gamma \rangle$ with unknown environment dynamics P and R. In order to perform planning, M is mapped to another MDP $\bar{M} = \langle \bar{S}, \bar{A}, \bar{R}, \bar{P}, \gamma \rangle$. The intuition behind is that knowing the optimal policy for that MDP helps to act optimally in the actual MDP M. The MDP $\bar{M}$ is inferred solely on observations $\phi(s)$ in M. So henceforth $\bar{M}$ is also called inferred MDP [TLA16].

The mapping is shown in figure 3.4. $\bar{M}$ is created with a learned reward mapping $\bar{R} = f_R(\phi(s))$ and transition mapping $\bar{P} = f_P(\phi(s))$. The actual choice for $\bar{S}$ and $\bar{A}$ is

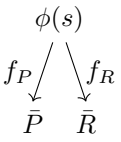


Figure 3.4: Mapping to Inferred MDP

left open as domain specific design choice. A common choice is to use the same state $S = \bar{S}$ and action $A = \bar{A}$ as in the true MDP [TLA16].

Inside the inferred MDP approximate planning is performed with a planning module, which is discussed in detail later.

3.2 Value Iteration Networks

The first method proposing this idea was the *Value Iteration Network*, henceforth VIN. VIN integrates the Value Iteration algorithm in a neural network structure. The key observation thereby is that *Value Iteration* can be formulated as a recurrent *Convolutional Neural Network (CNN)* so that it is fully differentiable [TLA16].

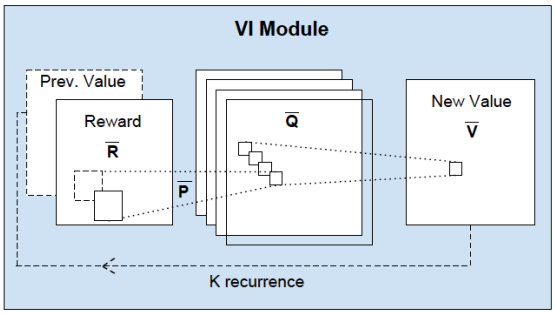


Figure 3.5: Value Iteration with deep CNN. The illustration is taken from [TLA16].

So the Value Iteration Module is the core part of VIN. Input to that planning module, as depicted in figure 3.5, is a *reward image* $\bar{R}$ with dimensions $\langle l, i, j \rangle$. It is assumed that the state space $\bar{S}$ matches to a two-dimensional lattice structure. Hence, the transition $\mathcal{P}(s'|s, a)$ can be expressed as a 3×3-convolution with weights $W^{\bar{a}}_{l,i,j}$ for each convolutional filter. The number of filters is defined by the action set $\bar{A}$ and a single filter performs the convolution in equation 3.1 [TLA16].

$$Q_{\bar{a},i',j'} = \sum_{l,i,j} W^{\bar{a}}_{l,i,j} \cdot \bar{R}_{l,i'-i,j'-j} \tag{3.1}$$

So the resulting tensor $\bar{Q}(\bar{s}, \bar{a})$ has a separate channel for each action. To match the computation with one pass of VI, as defined in equation 2.12, a *max-pooling* over the action dimension of the tensor is done [TLA16].

$$\bar{V}_{i,j} = \max_{\bar{a}} \; Q(\bar{a}, i, j) \tag{3.2}$$

The resulting values $\bar{V}$ are stacked with the rewards $\bar{R}$ and fed back into convolutional layer and max-pooling, which was described before. This procedure is done k times to perform k iterations of VI [TLA16].

When performing k iterations of VI, information will be propagated at most k steps. Hence, planning is limited to that finite horizon. For larger problem sizes an agent, that is currently in some state s, is unaware of rewards that are further away than k steps from state s. This leads

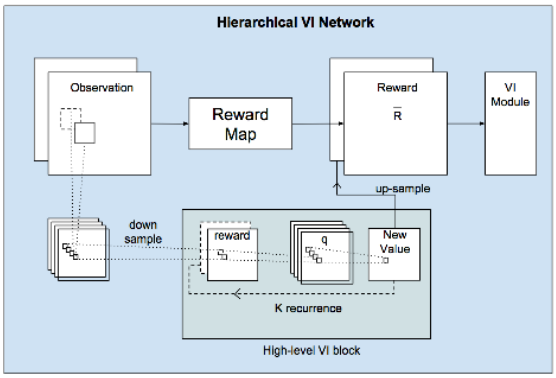


Figure 3.6: Hierarchical Approach to Value Iteration Networks. The illustration is taken from [TLA16].

to suboptimal plans. One could increase the number of iterations proportional with the problem size. But this would lead to much deeper *VIN* networks, that are slower and harder to train, due to vanishing gradients.

Hierarchical Value Iteration Networks (HVIN) are an extension to VIN, that was also proposed by Tamar et al. [TLA16]. It addresses the problem with hierarchical planning. This means that VI is performed at multiple levels of resolution. Therefore, a copy of the input is down-sampled by factor d and fed into the so-called *high-level VI module*. This module learns to perform VI at this lower resolution. The 'speed of information transmission' is thereby increase by factor d and so allows further planning with k iterations. The resulting values are up-sampled again and fed as input to the *standard VI module* [TLA16].

This idea can easily be extended to several hierarchical layers. Gupta et al. [GDL+17] plan with a d times down-sampled environment and conduct k value iterations. The output is up-sampled and the entire process is repeated until the resolution of the original problem is reached. This enables planning for goals, that are as far as $k2^d$ away, while only performing $d \cdot k$ value iterations [GDL+17].

SoftVIN enhances VIN by using a *softmax policy loss* for better propagating gradients back through the planning module [PAS18].

The *Generalized Value Iteration Network (GVIN)* is, as the name already implies, a generalization of the basic *Value Iteration Network (VIN)*, that was proposed in the section 3.2. One shortcoming of the original VIN is that it can only be applied on regular lattice structures like grids. This limitation is due to its formulation of VI as CNN. GVIN allows the generalization of the convolution operator to spatial graphs $G = \langle \mathcal{V}, \mathcal{X}, \mathcal{E}, A \rangle$ with n vertices $\mathcal{V}$ and edges $\mathcal{E}$. The adjacency is given by the matrix $A \in \mathbb{R}^{n \times n}$ and $\mathcal{X} \in \mathbb{R}^{n \times 2}$ denote the locations of the vertices in the spatial domain [NCG+17].

To actually perform convolution the so called *Graph Convolution Operation* $P \in \mathbb{R}^{n \times n}$ is used.

$$P_{i,j} = A_{i,j} \cdot \sum_{l=1}^{L} w_l \cdot K_d^{(t,\theta_l)}(\theta_{ij}) \tag{3.3}$$

Thereby the vector w contains the weights learned by the convolution. The directional Kernel $K_d^{(t,\theta_l)}(\theta) = \left(\frac{1+\cos(\theta-\theta_l)}{2}\right)^t$ activates areas around the reference directions θ_l with temperature t. The direction θ_{ij} can be computed based on the embeddings $\mathcal{X}$. The intuition behind this approach is that an action, that leads to a state in the direction of the goal, seems to be promising . Because P is a sparse matrix, the convolution $q^{(a)} = P^{(a)}(r + v)$ of values $v, q \in \mathbb{R}^n$ and rewards $r \in \mathbb{R}^n$, that are sitting at the vertices, can be computed efficiently [NCG+17].

So far all approaches considered that the problems are fully observable. The next approaches will show how to deal with partial observability.

3.3 Cognitive Mapping and Planning

Cognitive Mapping and Planning (CMP) is a neural network architecture for visual navigation of agents in novel environments. It uses a unified joint architecture for mapping of observations and planning with its belief such that the mapper is driven by the needs of the planner. The entire system is designed fully differentiable for end-to-end training. Basically CMP consists of two parts, a *spatial memory* and a *HVIN* planner, as described before in section 3.2 [GDL+17].

The *spatial memory* holds a metric egocentric belief f_t and confidence values c_t about the world in a top-down view. This belief is generated by the *mapper* from the belief in previous time step f_{t-1}, the current egomotion e_t and the visual observation I_t. It can be viewed as a gated recurrent network comparably to *LSTM* (see Olah [Ola] for comparison). Thereby the inherent partial observability of visual inputs is resolved. The following update U is performed to the memory:

$$f_t = U(W(f_{t-1}, e_t), f_t') \text{ , where } f_t' = \phi(I_t) \tag{3.4}$$

Here W is a function that transforms the previous belief f_{t-1} with the egomotion e_t to be egocentric again. The encoding of visual inputs $\phi(I_t)$ is implemented as a pretrained *Resnet-50* network that maps its latent variables to an estimate for the current egocentric top-down view. The confidence values for the current belief are updated in a similar way :

$$c_t = c_{t-1} + c_t' \tag{3.5}$$

As already mentioned the planner is a HVIN network that performs hierarchical planning on beliefs $\langle f_t, c_t \rangle$ of different scales $\langle s_0, \cdots, s_d \rangle$. Thereby s_i denotes an i times down-sampled belief. Starting with scale s_d value iteration is performed in an MDP induced by the reward mapping function f_R. The results of that computation are up-sampled to the next scale and the entire process is repeated. The values of scale s_0 are fed to a fully connected layer to select the action [GDL+17].

3.3.1 QMDP-Net

Karkus et al. [KHL17] follow a similar approach to deal with partial observability. The *QMDP-Net* holds and updates a Bayesian belief b_t about the agent's current state. For planning the Q_{MDP} algorithm is performed. This algorithm performs Value Iteration in the underlying MDP, which neglects the uncertain state of the POMDP. For the QMDP-Net this part follows the idea of Tamar et al. [TLA16]. Finally, the Q_{MDP} values are mapped with Bayesian belief b_t to a value $q(a) = \sum_{s \in S} Q_{\text{MDP}}(s, a) \cdot b_t(s)$ for the current action a [KHL17].

The major difference to CMP is that QMDP-Net does not perform a mapping from observations to a (top-down-view) Cognitive Map.

3.4 Latent State Space

In contrast, there is a branch of other methods that perform planning in *Latent State Space*. As the name already implies, latent states are hidden and so not known. The idea is that planning can be preformed easier in latent space compared to the high-dimensional input space.

Consider an input x as high-dimensional visual observation with dimension n_x. In general convolutional layers encode the input to some latent layer representation z. Farquhar et al. [FRIW17] defined this as *latent states* of the network. According to Bengio et al. [BCV13] we will henceforth also refer to z as learned representation of x.

Ordinary model-free networks, like DQN [MKS+15], use fully-connected layers to estimate a value based on this latent representation [FRIW17]. The idea of approaches presented in the following is to use the latent state space for planning instead.

In general the dimension n_z of a *latent state* z is much lower than the dimension n_x of the corresponding visual observation. This makes planning in latent spaces easier. So for a latent state representation equation 3.6 holds [WSBR15]:

$$n_z << n_x \tag{3.6}$$

A latent sate space can also be learned with unsupervised learning methods, as depicted in the next section.

3.4.1 Unsupervised Learning of Latent Space

Unsupervised Learning is another subfield of Machine Learning with no supervisor present. It's about detecting hidden structures in data. We will discuss two techniques to extract a latent state space with unsupervised learning methods: (i) Auto-Encoding and (ii) Slow Feature Analysis .

Auto-Encoding

Auto-Encoding (i) is an unsupervised learning technique that compresses input data X to a latent code z such that X can be reconstructed given z. Two types of Auto-Encoders are common: (a) convolutional and (b) variational Auto-Encoders.

A *Convolutional Auto-Encoder (CAE)* uses convolutional neural networks to fulfil the aforementioned task. The input data X is encoded with several *convolutional layers* to a latent representation z, which is the bottleneck layer of the network. A decoder, consisting of several de-convolutional layers, decompresses z to a reconstruction $\tilde{X}$. The auto-encoder is trained to minimize the mean-squared error between X and its reconstruction $\tilde{X}$. Hence, the loss $\mathcal{L} = \sum_i \mathcal{L}_i$ is:

$$\mathcal{L}_i = (X_i - \tilde{X}_i)^2 \tag{3.7}$$

for all data X_i in the training batch [Gal].

A *Variational Auto-Encoder (VAE)* models data as distributions: $q_\theta(z|x)$ for encoding and $p_\phi(x|z)$ for decoding. The encoder compresses an input x to a hidden representation z which is much lower dimensional space. The lower dimensional space is assumed Gaussian distributed with probability density $q_\theta(z|x)$. The decoder maps the hidden representation z to the probability distribution of the data that is encoded. With this mapping information is lost, because it is reconstructed from a smaller to a larger dimensionality. This information loss is measured with the reconstruction log-likelihood $p_\phi(x|z)$. So the loss function of the variational auto-encoder is given in equation 3.8.

$$\mathcal{L} = \sum_{i=1}^{N} -\mathbb{E}_{z\sim q_\theta(z|x_i)}[\log p_\phi(x_i|z)] + \mathrm{KL}\Big(q_\theta(z|x_i)\|p(z)\Big) \tag{3.8}$$

Therewith KL denotes the Kullback-Leibler divergence between the encoder's distribution $q_\theta(z|x)$ and $p(z)$. It measures how close q is to p. For VAE $p(z) = \mathcal{N}(z|0, 1)$ is a standard normal distribution with mean zero and variance of one [Alt].

Slow Feature Analysis

Slow Feature Analysis (SFA) is an unsupervised learning technique to extract a set of slowly varying features from an input data stream. Given a multidimensional and time dependent input signal $x(t) = [x_1(t), \cdots, x_N(t)]^T$, SFA finds a set of functions $g_1(x), \cdots, g_k(x)$ such that each output signal $y_i = g_i(x(t))$ varies as little as possible over time [SW13].

Schoenfeld and Wiskott [SW13] combine these slowly varying features with an *Independent Component Analysis (ICA)* to form a sparser coding of features.

3.4.2 Shaping the Latent Space

In the spatial state space representation each state variable corresponds to a specific cell in the tensor representing the state space. A metric for distance between different state variables is defined by the spatial shape of the tensor. So the distance in the state space corresponds to the distance of cells in the spatial tensor.

When reducing an input x to a latent representation z of much lower dimension, it encodes x densely in $\mathbb{R}^{n_z}$. Thereby a new metric in latent space is implied by the scalar product $\|z_1 - z_2\| = \sqrt{\langle z_1 - z_2, z_1 - z_2\rangle}$. This metric is also latent and so hard to interpret. It defines the shape of the latent space.

These definitions of latent space are not restrictive. So the encoding in a latent representation could basically still mean anything. This makes planning not effective. In order to perform planning in latent space, one would need to restrict the shape further.

One idea, that is shared by several approaches, is to use *auxiliary losses* to shape the latent space. This means to add additional loss terms, which are not inherently necessary by the task definition, but help to gather better representations. Hence, they force the latent state space to have particular properties [FRIW17].

There are several existing methods to do so. Common ideas are briefly summarized below:

State Discrimination One idea is to force latent states z to be good discriminators for the input x. Kulkarni et al. [KSGG16] and Watter et al. [WSBR15] achieve this by using auto-encoders that minimize an auxiliary reconstruction loss, as given in equation 3.7 and 3.8.

Reward Prediction Another auxiliary loss often enforces a *reward prediction* $\hat{r}(z_t)$, that is used for planning, to match with the immediate ground truth rewards r_t, that the agent observes in the environment. This is done by reducing the squared error $L = \left(\hat{r}(z_t) - r_t\right)^2$ of predictions [KSGG16] [OSL17] [FRIW17].

Consistent Transition A distinct set of methods tries to shape the latent space in a way that transitions can be easily performed. Watter et al. [WSBR15] force the dynamics to be linear in the latent representation. For tree planning consistent transitioning can be achieved by sharing weights of the transition function throughout the tree [OSL17] [FRIW17]. Also, differentiating through the entire tree ensures that the learned representation are good for planning [FRIW17].

Slowness Another approach is to extract the slowest varying features in observations. Thereby one captures the regularities in the environment and in its dynamics. This leads to meaningful representations [SW13].

How to actually plan in latent space is discussed in the following.

3.5 Tree Planning

One idea of planning in latent space is to perform *tree search*. Like MCTS, which was discussed earlier, these methods focus computations on planning for the current state. This is also considered as local planning.

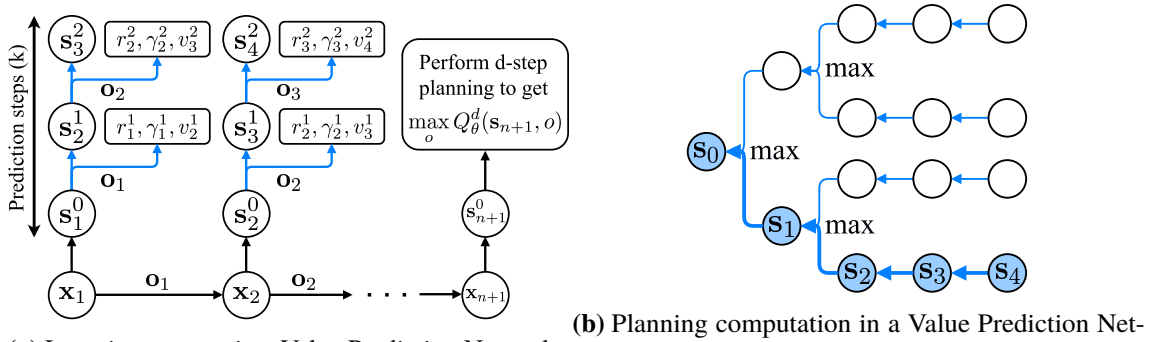
In contrast, value iteration methods perform planning over the entire state space. This kind of planning requires the state space to be sufficiently small. Tree planning on the other hand allows scaling even with large state space, because optimal actions are only computed for a small subset of states [OSL17].

In order to integrate tree planning in a differentiable module some adjustments need to be done. As a neural network has a fixed structure that needs to be defined before training starts, the branching and depth of the tree need to be limited. Especially deep trees encoded in a very deep network would suffer from vanishing gradients.

3.5.1 Value Prediction Network

The *Value Prediction Network (VPN)* is the first algorithm developed to perform tree planning in latent space. It therefore predicts the rewards and values of future latent states.

Given a current latent state s_t, VPN performs a local *Temporal Difference Search* with depth d. The principle is depicted in figure 3.7b. This search improves the value estimate for the current state by aggregating the maxima of all branches [OSL17].



(a) Learning process in a Value Prediction Network.

(b) Planning computation in a Value Prediction Network.

Figure 3.7: Differentiable Tree Planning in the Value Prediction Network. The figures are taken from [OSL17].

To grow the tree in latent space a *core module* f_{core} is recursively applied. It is composed of (i) a *transition module* $f_{\text{trans}} : s_t, o_t \mapsto s_{t+1}$ that predicts the consecutive latent state s_{t+1}, (ii) an *outcome module* $f_{\text{out}} : s_t, o_t \mapsto r_t, \gamma_t$ estimating the immediate reward and discounting from following option o, and (iii) a *value module* $f_{\text{value}} : s \mapsto V(s)$. All these modules are designed as neural networks that are integrated in the VPN. This ensures the tree to be fully differentiable and allows end-to-end training [OSL17].

The VPN is trained with a combination of n-step Q-Learning and supervised learning. Given an n-step trajectory $\langle x_0, o_0, r_0, \gamma_0, \cdots, x_n, o_n, r_n, \gamma_n, x_{n+1} \rangle$ the predicted values of the VPN are trained with n-step Q-Learning. The return values R_t are bootstrapped with a previous version of the value network [OSL17].

$$R_t = \begin{cases} r_t + \gamma_t R_{t+1} & \text{, if } t \leq n \\ \max_a Q(s_{t+1}, a) & \text{, else} \end{cases} \tag{3.9}$$

The immediate reward r_t^l and discounting factor predictions γ_t^l are trained with supervised learning from the data generated by the agent, that follows a behavioural policy. Thereby the upper index l indicates the depth of prediction, as it is depicted in figure 3.7a. Supervision is provided by the truly observed reward r_t and discount factor γ_t, that matches the timestamp t of the prediction. The training of the value function v_t^l behaves similar with the bootstrapped return R_t as learning target.

$$\mathcal{L}_t = \sum_{l=0}^{k} (R_t - v_t^l)^2 + (r_t - r_t^l)^2 + (\gamma_t - \gamma_t^l) \tag{3.10}$$

The total loss is composed of the partial losses for each time step in the trajectory [OSL17].

With this training setup the network is forced to match its predictions v_t^l, r_t^l and γ_t^l with observations during the path that was taken by the agent. This can be understood as a form of *on-policy* learning.

It remains to remark that during optimization the *core module* is not embedded in the planning algorithm. So the gradients do not propagate back through the planning computation. Planning is only used to compute bootstrapped targets for the value function v [FRIW17].

3.5.2 TreeQN

A different algorithm that follows a similar line of work is *TreeQN*. This approach also expresses tree search in latent space in a differentiable way. The major difference is the training setup: *TreeQN* is trained end-to-end with a version of n-step Q-learning, as shown by equation 3.11.

$$\mathcal{L}_{\text{n-step Q}} = \sum_{j=1}^{n} \left(\sum_{k=1}^{j} [\gamma^{j-k} * r_{t+n-k}] + \gamma^{j} \max_{a'} Q^{-}(s, a') - Q(s_{t+n-j}, a_{t+n-j}) \right) \tag{3.11}$$

Given a n-step trajectory $\langle s_t, a_t, \cdots, s_{t+n}, a_{t+n} \rangle$ the value function $Q(s, a)$ is optimized for all states in the trajectory. For each state action pair s, a several learning targets $\sum_{k=1}^{j} [\gamma^{j-k} \cdot r_{t+n-k}] + \gamma^{j} \max_{a'} Q^{-}(s, a')$ of different length $j \in \{1, \cdots, n\}$ are computed. All targets are bootstrapped after the last step with a previous version of the value network $Q^{-}(s, a)$ [FRIW17].

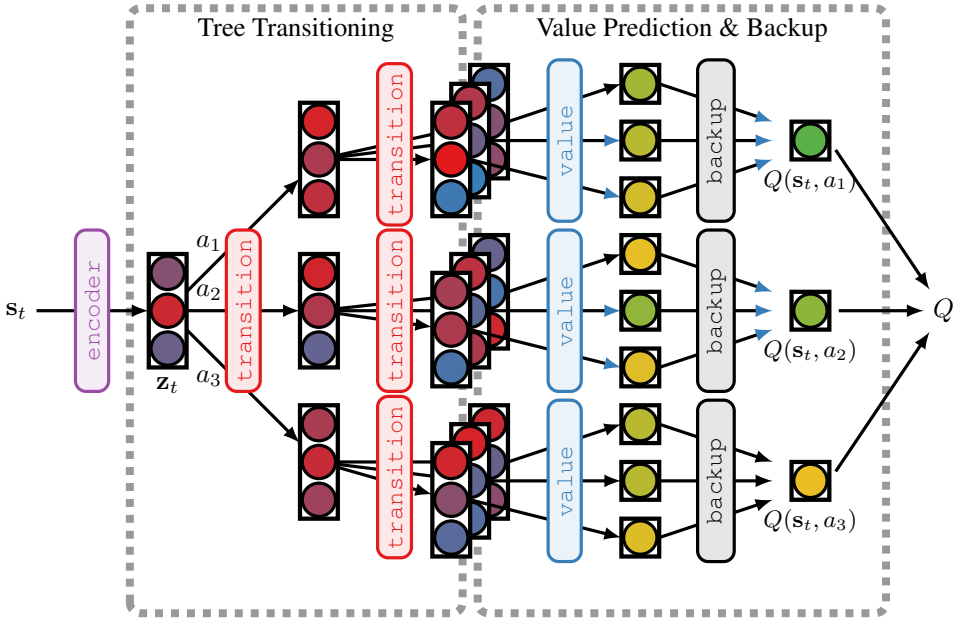


Figure 3.8: Differentiable Tree Planning in TreeQN. This illustration is taken from [FRIW17]. It shows the basic principle

In contrast to Oh et al. [OSL17], this setup allows that gradients propagate back through the planning computation. It is achieved by training the whole network end-to-end. The value network Q, which contains the embedded planning module, is directly optimized by the loss in 3.11. So gradients flow back from the finale layer through the whole network structure, that is depicted in figure 3.8, to the input stage. This includes also paths in the tree that actually have not been taken by the agent in its trajectory [FRIW17].

So the *TreeQN* also learns about the value of states that are located off the policy which is followed. This might be understood as a form of off-policy learning.

3.6 Successor Features

Tree planning methods, as introduced before in 3.5, use a current latent state representation s_t to make action conditional *single step* future predictions with a transition module. This is the way how the trees are incrementally grown.

A different branch of work considers not only single step future predictions. The idea is to learn multi-step predictions $\psi^\pi(s, a)$ for the future that are contingent upon a policy π.

This is achieved with the assumption that rewards are linear $r(s, a) = \phi(s, a)^T \cdot w$ in some features. By exploiting this linearity in the definition of the Q-Value, one can decouple the environment dynamics from the rewards and hence the values [BDM+17].

$$Q^\pi(s, a) = E^\pi[\phi_{t+1}^T \cdot w + \gamma\phi_{t+2}^T \cdot w + \gamma^2\phi_{t+3}^T \cdot w + \ldots | S_t = s, A_t = a] \tag{3.12}$$

$$= E^\pi\Big[\sum_{i=t}^{\infty} \gamma^{i-t} \cdot \phi_{i+1} | S_t = s, A_t = a\Big]^T \cdot w = \psi^\pi(s, a)^T \cdot w \tag{3.13}$$

So the *Successor Features (SF)* ψ^π are a way of summarizing the dynamics induced by π in a given environment [BDM+17].

3.6.1 Deep Successor Representation

The idea of *Deep Successor Representation (DSR)* is to regress these SF with a neural network. Therefore, DSR uses convolutional layers $f : \mathcal{S} \mapsto \mathbb{R}^d$ to compress information contained in state $s \in \mathcal{S}$ into a compact feature representation $\phi_s \in \mathbb{R}^d$ such that successor features can be computed based on ϕ_s. These layers f can be viewed as the encoder part of an *convolutional auto-encoder*. The decoder network $g_{\tilde{\theta}}(\phi_s)$ consist of deconvolutional layers. So the reconstruction loss

$$L_t^a(\tilde{\theta}, \theta) = \big(g_{\tilde{\theta}}(\phi_{s_t}) - s_t\big)^2 \tag{3.14}$$

forces features ϕ_s to be good discriminators for the state s. Additionally, the features ϕ_s are forced to be good predictors for the immediate reward $r(s_t, a_t)$ in state s_t (with action a_t). This is achieved by the auxiliary loss given in equation 3.15.

$$L_t^r(w, \theta) = (r(s_t, a_t) - \phi_{s_t} \cdot w)^2 \tag{3.15}$$

Thereby the reward prediction is implemented as a single layer with weights w [KSGG16].

To actually regress the SF with additional fully-connected layers $u_\alpha(\phi_s, a)$, the DSR network uses an adapted form of TD-Learning, as shown by the loss in equation 3.16.

$$L_t^m(\alpha, \theta) = \mathbb{E}\left[\Big(\phi(s_t) + \gamma u_{\alpha_{\text{prev}}}(\phi_{s_{t+1}}, a') - u_\alpha(\phi_{s_t}, a)\Big)^2\right] \tag{3.16}$$

Here $u_{\alpha_{\text{prev}}}$ denotes a previous version of the network to stabilize the training process. Optimization is performed on the compound loss $L_t = L_t^r + L_t^a + L_t^m$ [KSGG16].

3.7 Embed to Control

All the previous ideas used auxiliary losses to guide a neural learner towards learning better representations in its latent state z_t. A similar approach from the related field of *Optimal Control* shows how auxiliary losses can be used to identify a good latent representation for planning by explicitly forcing the latent system to be in a class of well-known problems.

Embed to Control (E2C) applies *optimal control* to systems with high-dimensional non-linear input x_t, e.g. raw images and without any access to the underlying physical model $s_{t+1} = f(s_t, u_t)$ with states s and control signal u. E2C tries to reduce this high-dimensional non-linear system by identifying a mapping $m(x_t) : \mathbb{R}^{n_x} \mapsto \mathbb{R}^{n_z}$ to a low-dimensional system in latent space $z_t = m(x_t) + \omega$, in which optimal control can be applied easily. In that context easy means that the dynamics are linear, see equation 3.17, and the costs $c(z_t, u_t)$ in each step are quadratic, as shown in equation 3.18 [WSBR15].

$$z_{t+1} = Az_t + Bu_t \tag{3.17}$$

$$c(z_t, u_t) = (z_t - z_{\text{goal}})^T R_z (z_t - z_{\text{goal}}) + u^T R_u u \tag{3.18}$$

The model is trained with data $\mathcal{D} = \{(x_t, u_t, x_{t+1})\}_t$ consisting the observation x_t at time t, the applied control u_t, and the consecutive observation x_{t+1}. E2C assumes all variables normal distributed. So for the encoder part the latent variable is distributed like $Q_\phi(Z|X) = \mathcal{N}(\mu_t, \text{diag}(\sigma^2))$ with mean μ_t and variance σ regressed by a neural network. And hence, the transition model $\hat{Q}_\psi(\hat{Z}|Z, u) = \mathcal{N}(A_t\mu_t + B_t u_t, A_t\Sigma_t A_t^T + H_t)$ transforms the distribution of latent states Q_ϕ with the learned linear dynamics A_t, B_t as shown in 3.17. A Bernoulli distribution $P_\theta(X|Z)$ describes a generative model, which reconstructs observations $x \in X$ from latent states [WSBR15].

The loss $\mathcal{L}^{\text{bound}}$ is a combination of reconstruction losses of two VAE. It enforces to properly encode the latent space variables z_t from observation x_t, and $\hat{z}_{t+1}$ from x_{t+1} respectively by minimizing the *variational bound*. See 3.8 for comparison [WSBR15].

$$\mathcal{L}^{\text{bound}}(x_t, u_t, x_{t+1}) = \mathbb{E}_{z_t \sim Q_\phi,\, \hat{z}_{t+1} \sim \hat{Q}_\psi}[-\log P_\theta(x_t|z_t) - \log P_\theta(x_{t+1}|\hat{z}_{t+1})] + \text{KL}(Q_\phi \| P(Z)) \tag{3.19}$$

The final loss $\mathcal{L} = \mathcal{L}^{\text{bound}}(x_t, u_t, x_{t+1}) + \text{KL}\left(\hat{Q}_\psi(\hat{Z}|\mu_t, u_t) \| Q_\phi(Z|x_{t+1})\right)$ is enhanced with an additional Kullback-Leibler divergence, which minimizes the difference between the distribution for predicted next state latent states $\hat{Q}_\psi$ and the encoding distribution Q_ϕ of the true observation in the next state x_{t+1}. This aligns predictions with observations [WSBR15].

Hence, a *Linear Quadratic Regulator (LQR)* can be applied to the learned system in latent space. Unlike to the other approaches introduced before, this planning algorithm is not directly embedded in the neural network structure. Future work may consider integrating a differentiable LQR inside the network to allow end-to-end training [WSBR15].

3.8 Conclusion

The various approaches to integrating differentiable planning modules have shown that the shape of learned representations is important. VIN and derived methods exploit the spatial shape to perform convolutions. Others use auxiliary losses to shape the latent space so that it has desired properties and hence ensure meaningful planning.

But there are also downsides, when using auxiliary losses. Requiring a latent representation to have certain properties with an auxiliary loss always introduces a certain amount of bias to the representation. So, Farquhar et al. [FRIW17] experienced the best results with as few auxiliary losses as possible and hence an unbiased learned latent representation.

On the other hand, with few auxiliary losses it is not very restrictive what latent representations encode. With this it is likely to over-interpret the meaning of latent representations and especially planning computations, that are encoded in the weights of the network [1].

So a proper implementation of embedded planning modules needs to trade-off between these aspects.

3.8.1 Stability

Furthermore, neural networks with embedded planning modules seem to be comparably unstable to train. For instance VIN [TLA16] is trained with supervised learning and ground truth data for optimal paths. Its performance, when trained with RL, is rather poor [NCG+17].

Also, Gupta et al. [GDL+17] trained in a supervised manner with ground-truth paths in the environment. To enable RL without ground-truth data, Niu et al. [NCG+17] introduced *Episodic Q-Learning*, a new RL algorithm that stabilizes their network's training process.

The use of auxiliary losses can further destabilize the training. I also observed this during my experiments, when applying DSR to Atari games. The auto-encoder and the corresponding reconstruction loss destabilized the training process and led to total divergence in some cases. Only removing this auxiliary loss completely helped to achieve learning in that particular task. Results are depicted in section A.2 in more details.

Next, we will have a look how the predictive features generated by planning modules can be exploited to achieve better learning.

[1] see https://redd.it/7a3av6 for an interesting discussion with Farquhar et al. [FRIW17]

4 Exploiting Forecast Features

In this chapter we discuss different approaches from literature on how features that are generated by an integrated planning module can be exploited for better learning of a policy. Thereafter, we will summarize the ideas in a *Planning for Learning* framework. Finally, we introduce *Unsupervised Planning for Learning*, an concrete application of the before introduced ideas for a visual navigation task.

In general features gathered from planning can be understood as discriminators about the future. Most planning algorithms gather values and rewards of different trajectories in the future to back up the forecast quantities e.g. future V, Q, R and get a better estimate of the current state's optimal future. Intuitively these features can be understood as a forecast. So the idea is to use these features to form a better representation of the current state so that a value network can better learn a value for that state.

For most of the modules introduced in chapter 3 the planning computation is not performed until convergence. It is only performed for a fixed number of steps or iterations. So the predictive features generated are only approximative and possibly inaccurate. A proper exploitation needs to deal with this fact.

4.1 Improved Value Estimates

The most straight forward idea is to directly use the improved values $Q(s, a)$ for the current state s. This can be done when the embedded planning computation refines the values locally.

To do so, Oh et al. [OSL17] and Farquhar et al. [FRIW17] perform embedded tree planning as local refinement of the state action values for the current abstract state z_t.

In contrast, when global planning is performed, the resulting values wouldn't be meaningful for a particular state. Tamar et al. [TLA16] extract only those values that are meaningful for the current state with a so-called attention module. But these values are exploited in a different way as explained later.

The entire approach is extended by Farquhar et al. [FRIW17] with the algorithm *ATreeC*. The idea is to use the network with embedded planning module as an *Actor* in an *Actor-Critic* setup (see section 2.1.3) and thereby improve the values. The critic is a neural network that regresses a value function $V(z_t)$ based on the latent representation z_t of the current state x_t.

4.2 Improved State Features

A different branch of methods does not exploit the values from planning computation directly. They rather improve the input features of a policy network to better capture the agents current state. From a different point of view this idea can be viewed as concatenating input features with a set of forecast features to better constitute the current state of the agent.

Tamar et al. [TLA16] concatenate input features $\phi(s)$ for the policy network with the values encoded in $\psi(s)$, that are gathered from planning in the inferred MDP $\bar{M}$, to come up with enhanced input features for the policy network $\pi(a|\phi(s), \psi(s)$ and respectively a better representation for the current state.

Weber et al. [WRR+17] follow a similar approach by enriching the features for the current state with an imagination code c_{ia}. Therewith they summarize features from different simulated future trajectories, as it is briefly outlined in the following.

4.2.1 Imagination Augmented Agents

Imagination Augmented Agents (I2A) [WRR+17] learn to interpret the prediction of an environment model. Therefore, the model's predictions are fed as additional context information to *deep policy networks*.

The environment model is used to produce *'imagined trajectories'* or rollouts. The actions in each rollout are chosen with a rollout policy $\hat{\pi}$. It's used to generate n trajectories $\mathcal{T}_1, \cdots, \mathcal{T}_n$. Each trajectory $\mathcal{T}$ consists of a sequence of features $\langle \hat{f}_t, \cdots, \hat{f}_{t+\tau} \rangle$ with rollout length τ. These features $\hat{f} = \langle \hat{o}, \hat{r} \rangle$ can consist of predicted observations $\hat{o}$ and rewards $\hat{r}$ for instance. Instead of completely relying on these predictions and using them for planning, I2A feeds them to a rollout encoder $\mathcal{E}$. It's implemented as an LSTM (compare [GSC99]) to which the features are fed in reversed order. Rollout embeddings $e_i = \mathcal{E}(\mathcal{T}_i)$ are the outcome. Finally, an aggregator $\mathcal{A}$ combines different rollout embeddings into a single contextual code $c_{ia} = \mathcal{A}(e_1, \cdots, e_n)$. In a last step c_{ia} and the results from a model-free path c_{mf} are combined in a policy network [WRR+17].

4.3 Policy Contingent Future Predictions

A trajectory of states $s_0, \cdots, s_T$ in an environment always heavily depends on actions $a_0, \cdots, a_T$ taken by the agent. So predictions into the future $s_{t+1}, \cdots, s_T$ are also contingent upon a policy. So one approach is to learn to predict the future that is contingent upon the policy π which is executed.

Kulkarni et al. [KSGG16] learn successor features ψ^π, which are basically policy-contingent predictions of future occupancy of features ϕ. Because rewards $r(s, a) = \phi(s, a)^T w$ are linear in those features, it can easily be exploited to determine the value $Q(s, a) = \psi^\pi(s, a)^T w$ of a state and action with a scalar product.

A different approach is to regress the future under some policy π_g that tries to reach a goal g. How these future predictions can be exploited is outlined in the following.

4.3.1 Goal-Contingent Future

As introduced in section 2.3.1, storing data in a replay buffer can be viewed as an approximate model. Also, learning for multiple tasks simultaneously can help to acquire better latent representations (see section 2.3.2) . These ideas can be combined to exploit goal-contingent future predictions, as outlined in the following.

Hindsight Experience Replay

A technique called Hindsight Experience Replay (HER) tries to combine those ideas. By relabeling data in the replay buffer, it assumes multitask-learning with every state being a possible goal, i.e. $\mathcal{G} = \mathcal{S}$. So intuitively, given a trajectory of states which has been observed, the agent imagines in hindsight that the state finally reached actually has been the desired goal state all the time. Thereby it can learn from positive example. This allows networks to learn in domains with very sparse rewards [AWR+17].

Andrychowicz et al. [AWR+17] proposed several techniques to relabel the data that was gathered by an agent acting in the environment with some behavioural policy. The method that performed best is to relabel transition data with future states from the same trajectory. This means, given a trajectory $s_t, \cdots, s_T$ and a transition $\langle s_t, a_t, r_t, s_{t+1}, g \rangle$, one replaces the true goal g of the transition with a random future state of the trajectory $g' \in \{s_{t+1,\cdots,s_T}\}$. Thereby every goal corresponds to a predicate $f_g(s) = \mathbb{1}[s = g]$. The rewards $r_g(s, a) = -\mathbb{1}[f_g(s) = 0]$ are also adjusted, in order to match the new goal g'. So the agent receives a negative reward at every time-step when the goal is not achieved. Using a UVFA the value $Q(s, a, g)$ is regressed for each goal $g \in \mathcal{G}$ simultaneously [AWR+17].

Temporal Difference Models

Pong et al. [PGDL18] proposed with *Temporal Difference Models (TDM)* an algorithm that extends this idea with planning horizons. With this extension the value function $Q(s, a, g, \tau)$ is also conditional to a planning horzion τ. The reward is adjusted in a way so that there only is a reward in the τ-th step, namely the negative distance to the goal $-\|s' - s_g\|$. So the learning target $y = -\|s' - s_g\|\mathbb{1}[\tau = 0] + \max_a Q'(s', a, s_g, \tau - 1)[\tau \neq 0]$ regresses the value to intuitively learn, how close the agent can get to the goal g in τ steps, when currently being in state s and taking action a [PGDL18].

So TDM can serve as a goal and horizon-contingent predictor for the future state after τ steps. This knowledge can directly be exploited, when the value function $Q(s, a, s_g, \tau) = -\|f(s, a, s_g, \tau) - s_g\|$ is regressed as the difference between some latent variable f and the goal state s_g. By minimizing Q, one forces f to match the future state after τ steps [PGDL18].

$$a_t = \underset{a_t, a_{t+T}, s_{t+T}}{\operatorname{argmax}} \; r_c\Big(f(s_t, a_t, s_{t+T}), a_{t+T}\Big) \tag{4.1}$$

This allows an unconstrained optimization over these future states f (as shown in equation 4.1), when a reward is only assumed to occur in the final state [PGDL18].

This method differs from other approaches, as it does not incorporate the features for learning, but performs approximate planning. Future work may consider to exploit the learned predictions in a better state representation.

4.4 Planning for Learning

In the following we want to incorporate these ideas in a combined *Planning for Learning* Framework. It's motivated by the observation that planning algorithms summarize information about the future, which is gathered by applying the transition function $P(s'|s,a)$, in a set of features ψ. These features can either be values (see section 3.2), successor features (see section 3.6), or encoded trajectories (see section 4.2.1). The idea, which is shared by the approaches reviewed in the previous sections, is to exploit those features to get a better representation for the current state. This improved representation is then used to learn a value function or policy for the task at hand.

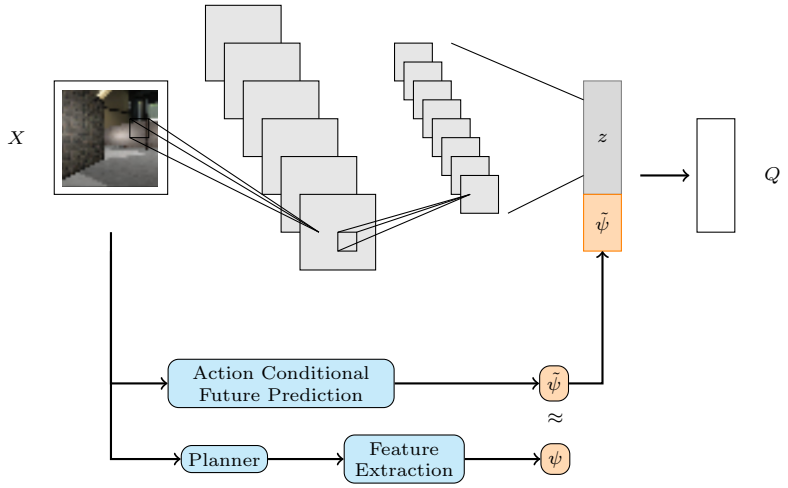


Figure 4.1: The Planning for Learning Framework for enhancing a latent representation z with predictive features $\tilde{\psi}$

The general principle is depicted in figure 4.1. An *action-contingent future predictor* predicts approximate features $\tilde{\psi}$. This could either be (1) an embedded planning module or (2) a neural network that regresses features ψ extracted from a planner like MCTS and hence $\psi \approx \tilde{\psi}$.

A latent representation z of the state x is concatenated with these features to form a better representation for regressing a policy.

Intuitively, in order to estimate the state action value Q for a particular x, the network's weights need to encode how much the task at hand is correlated with the action-contingent future prediction $\tilde{\psi}(a)$. So this is very similar to a planning computation: A future $\tilde{\psi}$ is anticipated with the future predictor and its goodness, with respect to a particular task, is evaluated by the assignment of weights to the features $\tilde{\psi}$.

Furthermore, we state this as the ability to anticipate the future learn and abstract plans in that feature space.

From a different point of view, the motivation of the idea can be compared to *Bayesian Smoothing*. Therewith the estimate $P(x_t) \propto P(\mathcal{F}|x_t, u_{t:T}) \cdot P(x_t|\mathcal{P}, u_{0:t})$ of the current state x_t is enhanced by incorporating future observations (or sensor measurments) $\mathcal{F} = y_{t+1:T}$ and controls $u_{t:T}$ in a backward filter $P(\mathcal{F}|x_t, u_{t:T})$.

4.4.1 Unsupervised Learning to Plan for Visual Navigation

In the following section we want to apply the ideas, which are summarized in the *Planning for Learning* framework (4.4), to visual navigation tasks.

Navigation tasks are interesting, because they're crucial for autonomous robots. An intelligent and adaptive agent is thereby required to integrate learning, planning and perception effectively. This is a challenging problem even for small domains.

The goal is to train an agent that navigates in an environment in a completely unsupervised manner. Also, the agent is not aware of its ground-truth position s in the world. Everything needs to be learned based on ego-perspective observations $\phi(s)$ of the environment. Hence, the agent must deal with partial observability.

The architecture is depicted in figure 4.2. The idea is to learn first a set of slowly varying features with a *Slow Feature Analysis* (SFA) (see section 3.4.1) from the observation $\phi(s)$ by randomly walking in the environment. These features $\psi(s)$ are improved with an Independent Component Analysis (ICA) so that each feature $\psi_i(s)$ encodes a particular area of the environment. We call this a landmark $\mathcal{L}$. Those are completely trained by Unsupervised Learning.

We will also call those features *landmark activations* in the following, because when the agent is near to the landmark the activation is high. Otherwise, the activation is low.

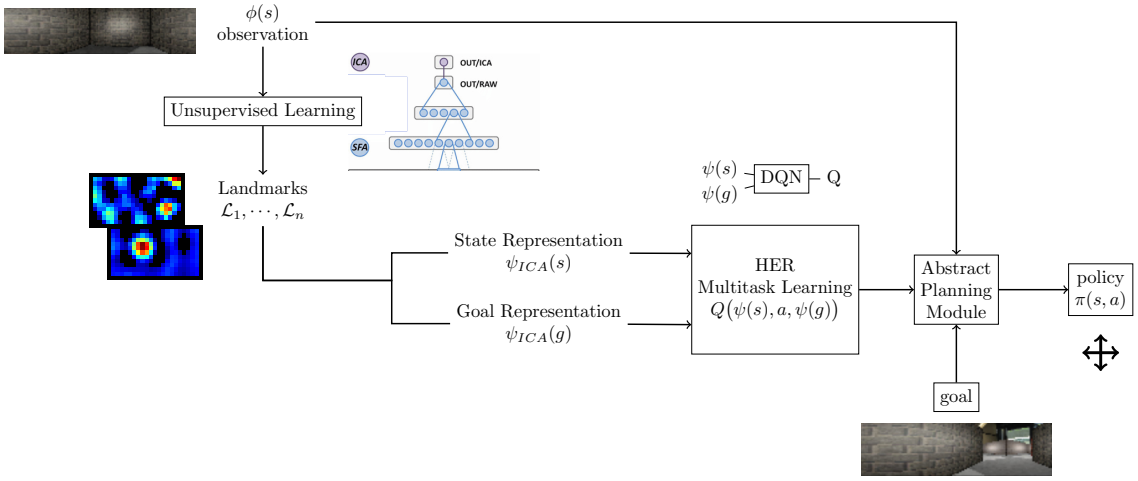


Figure 4.2: Overview of the Unsupervised Planning for Learning Architecture

Furthermore, we use these pre-trained features as state $\psi_{\text{ICA}}(s)$ and goal $\psi_{\text{ICA}}(g)$ representation in our Reinforcement Learning setup. We train a UVFA $Q\big(\psi(s), a, \psi(g)\big)$ in high-dimensional feature space with HER (4.3.1). Therefore, we replay transitions $\langle \psi(s), a, r, \psi(s'), \psi(g) \rangle$ with a customized goal predicate $f_g(s) = \mathbb{1}\big[\|\psi(s) - \psi(g)\| < \epsilon\big]$ in high-dimensional feature space. The algorithm is briefly depicted in pseudo code 4.1.

Knowing how to reach every position in the environment can be understood as an abstract model for the environment. It helps to learn better abstract plans for arbitrary tasks within the *Abstract Planning Module*.

As an extension to this framework the learned values $Q\big(\psi(s), a, \psi(g)\big)$ can be incorporated as action-contingent features to get a better representation $\psi'(s)$ for the current state of the agent. To be precise we use the learned values for a predefined set of landmarks $\{\mathcal{L}_1, \cdots, \mathcal{L}_n\}$ to form an improved feature set, as shown in equation 4.2. This improved representation can help to better learn to navigate inside the environment.

$$\psi'_i(s, a) = Q\Big(\psi(s), a, \psi(\mathcal{L}_i)\Big) \quad \forall i \tag{4.2}$$

$$\psi'_i(s) = \max_a Q\big(\psi(s), a, \psi(\mathcal{L}_i)\big) \quad \forall i \tag{4.3}$$

With the binary rewards (compare section 4.3.1), which basically count the steps taken by the agent, learned values encode an approximate metric for distance to a specific goal or landmark.

From a different point of view, this set of values is a *Horde*. According to Sutton et al. [SMD+11], domain knowledge can be represented with a set of independent RL agents, which are called *demons*. Each demon answers a question. In this particular case the question is: "How far am I from that landmark?"

Different approaches to exploit the learned values for a better representation are evaluated in the following experiments chapter.

Algorithm 4.1 Pseudo Code for Unsupervised Planning for Learning

for number_of_cycles **do**
  for number_of_episodes **do**
    reset environment randomly
    $episode\_experience \leftarrow \emptyset$
    for $t \leftarrow 0, T$ **do**
      $a \leftarrow \epsilon$-greedy$(\phi(s))$
      $\phi(s'), r \leftarrow$ env_step(a)
      $\psi(s') \leftarrow$ ICA$(\phi(s))$
      $episode\_experience \cup \{\langle \psi(s), a, r, \psi(s'), \psi(g) \rangle\}$
    end for

    for $\langle \psi(s), a, r, \psi(s'), \psi(g) \rangle \in epsiode\_experience$ **do**
      $\hat{g} \leftarrow$ future$(\psi(s))$ // Imagine future state in the trajectory as the goal
      $\hat{r} \leftarrow -\mathbb{1}[f_g = 0]$ with $f_g = \mathbb{1}[\|\psi(s') - \psi(\hat{g})\| < \epsilon]$
      $replay\_buffer \cup \{\langle \psi(s), a, \hat{r}, \psi(s'), \hat{g} \rangle\}$
    end for
  end for
  for number_of_optimizations **do**
    sample batch $\mathcal{B}$ from *replay_buffer*
    minimize $\left(Q(\psi(s), a, \hat{g}) - \left(\hat{r} + \gamma \max_{a'} Q^{-}(\psi(s'), a', \hat{g})\right)\right)^2$ for experience in $\mathcal{B}$
  end for
  for number_of_evaluations **do**
    sample g randomly
    evaluate greedy w.r.t. $Q\big(\psi(s), a, \psi(g)\big)$
  end for
  synchronize Q and Q^{-}
end for

5 Experiments

The following experiments investigate the ideas that have been introduced in section 4.4.1.

5.1 Environment

The aforementioned visual navigation task can formally be defined as a *Partially Observable Markov Decision Process (POMDP)* with high-dimensional visual observations $\phi(s)$. Here $s \in \mathbb{R}^4$ is the ground truth state of the agent in the environment, that consists of the location $\langle x, y, z \rangle \in \mathbb{R}^3$ and an orientation $\theta \in [0, 2\pi]$ in the environment. The agent can perform discrete actions

$$\mathcal{A} = \left\{ \begin{pmatrix} \cos(\theta+\delta) & -\sin(\theta+\delta) \\ \sin(\theta+\delta) & \cos(\theta+\delta) \end{pmatrix} \begin{pmatrix} 1 \\ 0 \end{pmatrix} \middle| \delta \in \{\frac{\pi}{4}, \frac{\pi}{2}, \cdots, 2\pi\} \right\} \tag{5.1}$$

in a coordinate frame that is local to its current state. So the agent can step in its ego-centric perspective. The agent's task is to reach any desired goal state g. At each step the agent receives a reward of -1 unless the goal is reached in some epsilon-ball. So, the shortest paths are enforced by maximizing the return.

During evaluation the agent acts greedy with respect to the value function. Success is defined as reaching the goal within a limit of steps.

The visual observations are generated with *Ratlab* ([SW13]). It is an easy to use *Python* framework, that allows to run place code simulations. It includes possibilities to extract place field information, including SFA and ICA. These implementations will serve as a starting point for further experiments. An example for a visual observation $\phi(s)$ is depicted in figure 5.1. The environment is slightly adapted to allow deep RL using the discrete action set (see equation 5.1) in *Tensorflow* [MAP+15]. Implementation details are outlined in section A.1.

For the experiments a simple obstacle-free rectangular environment is used.

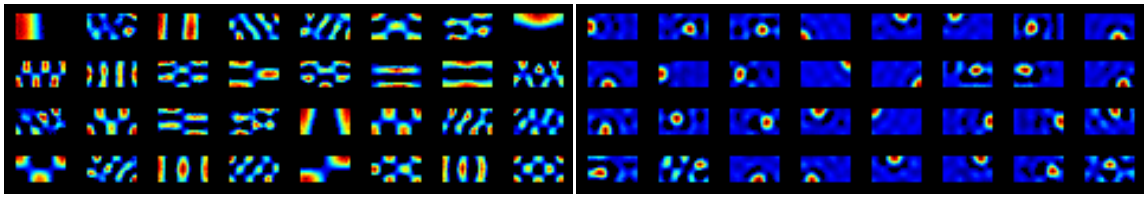
Figure 5.1: Example for a visual observation generated with Ratlab [SW13].

5.2 Landmark Activations from Unsupervised Learning

As introduced in the previous chapter, we train landmark activations with *Slow Feature Analysis (SFA)* from visual observations by extracting the slowest varying features. An example for such activations is depicted in figure 5.2a.

Additionally, it is possible to add a final *Independent Component Analysis (ICA)* layer to the network, which separates the activations to be independent of each other. An example is depicted in figure 5.2b. Both are generated with the tools provided by Schoenfeld and Wiskott [SW13].

As it is intended to use these landmark activations as a basis, we will henceforth also refer to those as *basis functions*.



(a) SFA activations learned from 10^5 steps of random walk in the environment.

(b) ICA activations learned from 10^5 steps of random walk in the environment.

Figure 5.2: Learned landmark activations - the continuous environment is sampled at discrete positions to gather the activation values. Values are averaged over the different directions θ.

During first trials we observed that the quality of learned landmark activations highly effects the convergence and stability of RL algorithms. So to better understand, what constitutes a good basis, we model landmark activations with *Radial Basis Functions (RBF)* in the following.

Radial Basis Functions

Radial Basis Functions roughly match the characteristics of the learned landmark activations. They simulate smooth spatial activations $\psi_i(s)$, which are only high in a local area around their centroids. The value of a single activation approaches zero, when leaving the local activation area. Equation 5.2 shows the simulated activations:

$$\psi_i(s) = e^{-\frac{\|s-s_i\|^2}{2\sigma^2}} \tag{5.2}$$

So in the following RBFs are placed at position s_i. In this context s is the ground truth position of the agent in the environment. This information is not given to the agent, but used to simulate the activations ψ_i. The variance σ determines the width of the receptive field.

5.2.1 Effect of Superimposition

In the following the effect of superimposing basis functions to learning will be evaluated. To do so, we set up a simpler environment with $\delta \in \{\frac{\pi}{2}, \pi, \frac{3\pi}{2}, 2\pi\}$ in a global frame. To analyse the effect of superimposition RBFs are placed equally distributed in the environment, but with different choices

for the width σ of the basis functions. So we vary the relative variance $c_\sigma = \frac{\sigma}{d_{ij}}$ with d_{ij} denoting the centroid distance between to neighbouring basis functions for different training setups. With this we have a measure for superimposition that is independent of actual locations and distances.

Training was performed until the best-performing setup reached a success rate of 100% in evaluation. Figure 5.3 shows the evaluation success rates for all setups.

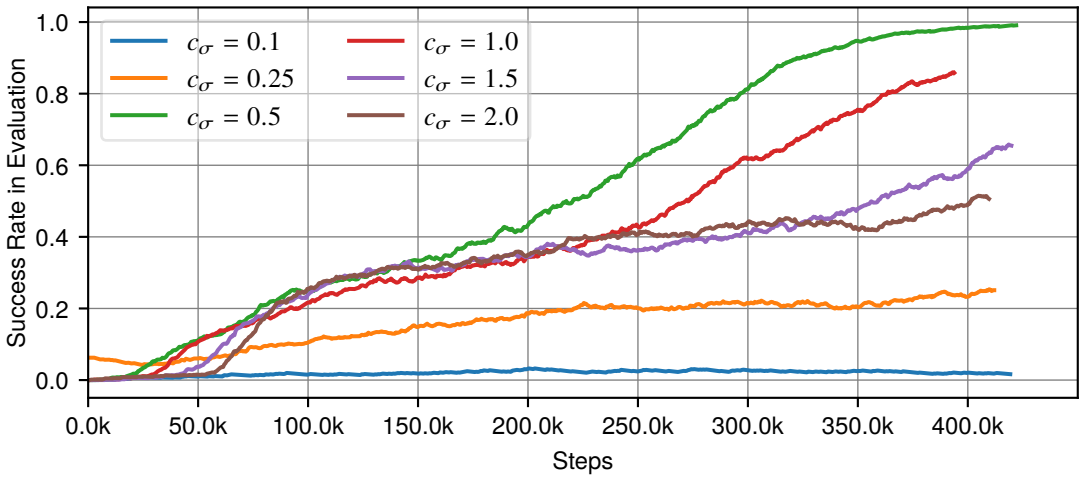


Figure 5.3: Success rate during evaluation plotted over the number of training steps for different variance scale c_σ and hence different superimposition.

The best performing setup $c_\sigma = 0.5$, which means that a basis function covers approximately halfway the distance to all neighbouring basis functions. So they are just slightly superimposing. With less superimposition ($c_\sigma = 0.25$) the network performed a lot worse. When decreasing the superimposition further ($c_\sigma = 0.1$), which means basically no superimposition at all, the network seemed not to learn anything.

Surprisingly, there also exists something like to much coverage of a single basis function. For broader basis functions $c_\sigma \in \{1.0, 1.5, 2.0\}$ the performance of the network after the same number of steps was a lot weaker.

5.2.2 Unsupervised Learning with Forecasts

Based on these observations we trained the ICA (and SFA) activations with 10^5 observations from random walk in order to get sufficiently superimposing activations. Then we fix the weights of the network, which produces these activations ψ and train a value function $Q\big(\psi(s), a, \psi(g)\big)$ with the training procedure introduced in pseudo code 4.1.

Figure 5.4 shows how the success rate in evaluation develops with the total number of steps taken by the agent in the environment. During evaluation the agent acts greedy with respect to the learned value function $Q\big(\psi_{\text{ICA}}(s), a, \psi_{\text{ICA}}(g)\big)$ and $Q\big(\psi_{\text{SFA}}(s), a, \psi_{\text{SFA}}(g)\big)$ respectively.

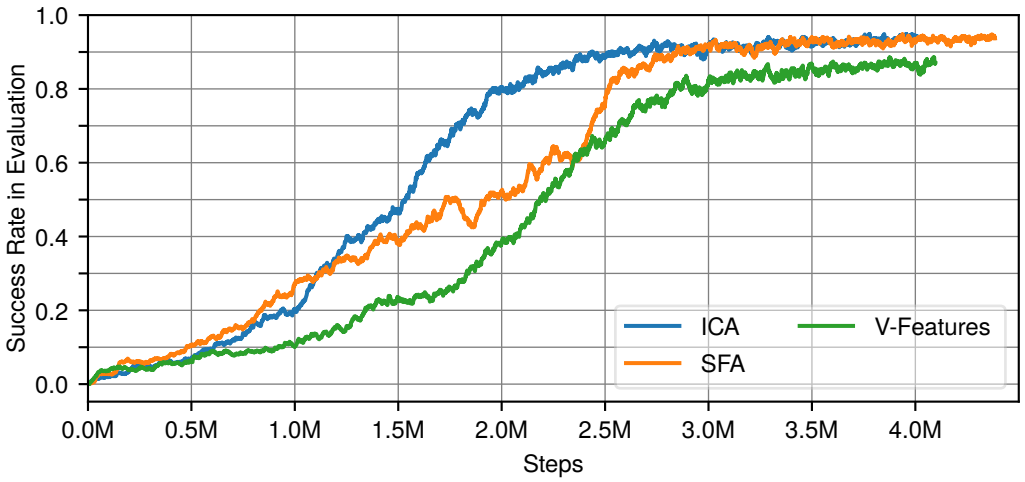


Figure 5.4: Success Rate in Evaluation, when acting greedy with respect to the learned value functions $Q\big(\psi_{\text{ICA}}(s), a, \psi_{\text{ICA}}(g)\big)$ and $Q\big(\psi_{\text{SFA}}(s), a, \psi_{\text{SFA}}(g)\big)$ respectively. Additionally, the best performing V-Features, that are trained based on the ICA, are added for comparison.

The setup with ICA features $\psi_{\text{ICA}}(s)$ as state representation slightly outperformed the SFA features $\psi_{\text{SFA}}(s)$. Hence, we will favour ICA features in the following. These plots will also serve as a baseline to evaluate enhanced features.

5.3 Enhancing Features

In the following we try to enhance the features $\psi_{\text{ICA}}(s)$ with the learned values $Q\big(\psi_{\text{ICA}}(s), a, \psi_{\text{ICA}}(l)\big)$ for a set of landmarks $l \in \{\mathcal{L}_1, \cdots, \mathcal{L}_n\}$.

For the following experiments we choose the landmarks to match with the centroids encoded in the ICA activations:

$$\mathcal{L}_i = \underset{l}{\text{argmax}} \;\; \psi_i(l) \tag{5.3}$$

The learned values can be understood as an approximate metric (for distance towards the landmark), as aforementioned. The resulting V-values $V(s) = \max_a Q(s, a)$ from the training in section 5.2.2 are shown in figure 5.5.

5.3.1 V-Value Features

Next, we will evaluate how to extract a meaningful set of features from the learned values (compare 5.5). Different approaches are considered: (a) linear scaling, (b) RBF kernel, (c) Cone kernel .

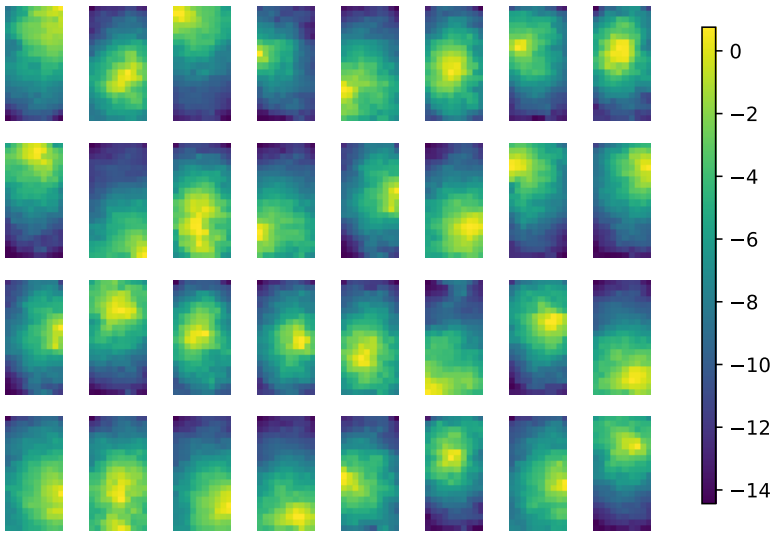


Figure 5.5: Learned values for a set of landmarks $\{\mathcal{L}_1, \cdots, \mathcal{L}_n\}$. These V-Values serve as enhanced basis for RL.

In general the values have arbitrary ranges. This might make learning difficult, as neural networks are likely to diverge when trained unscaled features. So we (a) scale the values linearly $\psi(q) = \frac{q-q_{\min}}{q_{\max}-q_{\min}}$ to fit in range $[0, 1]$.

Based on the observation that there exists something like the right level of superimposition, we exploit the values as approximate distance to the landmark. (b) We use this to place a RBF (equation 5.2). Hence, the true distance is replaced by the approximate metric $V(s) \approx \|s - s_i\|$. Last but not least, the same idea applied to form a local activation that is formed like a (c) cone: $\psi_i(s) = \max(1 - \frac{\|s-s_i\|}{\sigma}, 0)$.

The results are depicted in figure 5.6. As expected the unscaled values are not a good representation. But also the scaled values do not perform much better. This is consistent with the observation made that there exists something like too much coverage. In the following we will prefer the RBF kernel idea, which performed best. Due to sensibility to the aforementioned effect of superimposition, the RBF's receptive field width is optimized with a hyper-parameter search. Results for this particular environment are depicted in figure 5.7.

But even with the best hyper-parameter $\sigma = 5$ this setup cannot outperform the baseline provided in figure 5.6. So we conclude that the ICA-basis trained with 10^5 observations is already quiet a good basis for this easy domain.

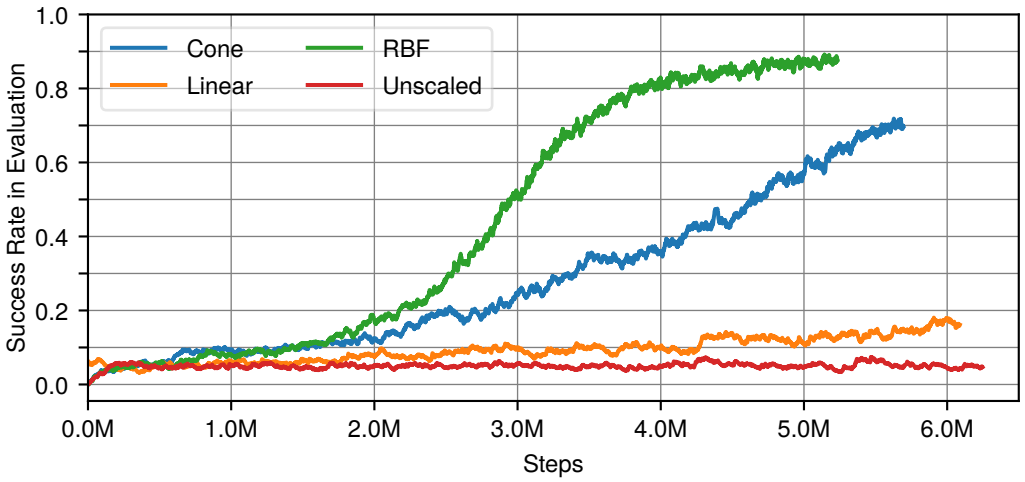


Figure 5.6: Success rate of different approaches exploiting the V-Values: Cone (c), Linear (a), RBF (b) and unscaled values.

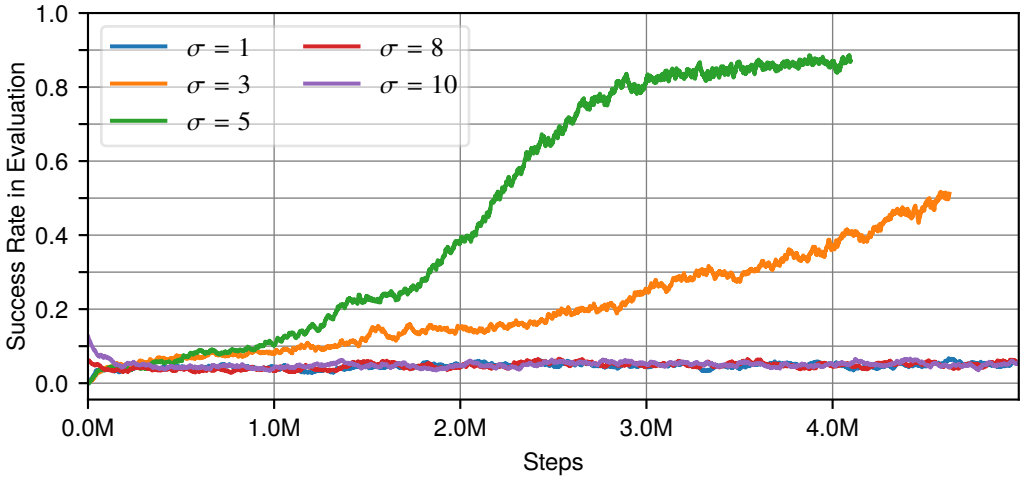


Figure 5.7: Learning Curve for different RBF kernel V-Features. Values hereby serve as an approximate distance metric to the kernel centroid. Results reflect the effect of superimposition, that has been previously investigated.

Weak ICA Features

In order to better analyse if values can improve a representation, we train the ICA network with only 10^4 observations. We therewith gather a weaker basis representation that does converge to a suboptimal solution, when used as features for RL. See figure 5.8 for comparison.

After training $Q\big(\psi_{\text{ICA}}(s), a, \psi_{\text{ICA}}(g)\big)$ with *ICA-Features* for $\approx 7 \cdot 10^6$ steps, values are extracted and used as input *Value-Features* to train $Q\big(\psi'(s), a, \psi'(g)\big)$. For better comparison results are shown in figure 5.8 with greedy and in figure 5.9 with ϵ-greedy behaviour.

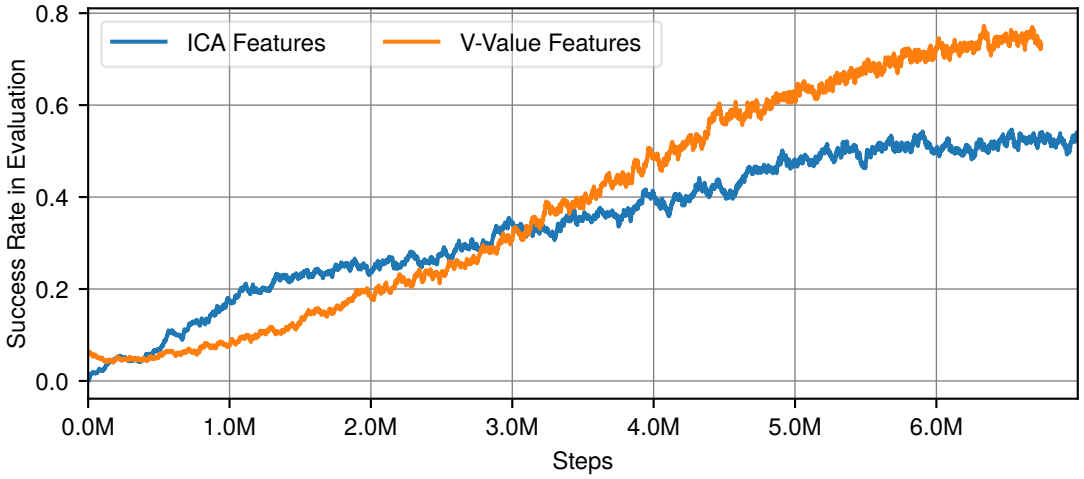


Figure 5.8: Success rate in evaluation of the weak ICA basis. V-Value features are extracted from the network, that is trained with ICA basis.

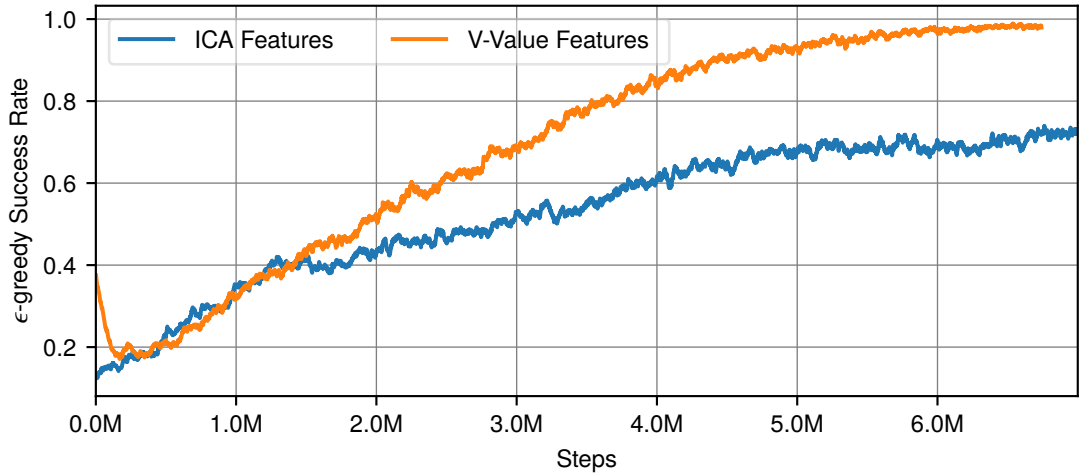


Figure 5.9: Success rate during ϵ-greedy exploration with the weak ICA basis. V-Value features are extracted from the network, that is trained with ICA basis.

The enhanced V-Value features ψ' outperformed the weak ICA features ψ by a significant margin in this experiment. It achieved the same level of performance with less training data. So the features ψ' are a better representation, that helps for data-efficient learning.

Furthermore, the value features also outperformed the baseline in the best achieved performance. It would be interesting to see if representations can iteratively improve with these newly learned values as input features. We leave this as an interesting direction for future work.

5.3.2 Q-Value Features

As introduced in equation 4.2 we will also evaluate how to enhance the representation $\psi'(s)$ with Q-Values. From several trials it turned out that the most promising approach is to have separate network head with weights θ_a for each action a. The value $Q(\psi'(s,a),a) = h_{\theta_a}(\psi'(s,a))$ for a particular action is regressed by a network h with weights θ_a. We input only the features $\psi'(s,a)$ to the network that correspond to this action a. Beside these adjustments we train with the same setup as aforementioned. We tried two different approaches to scale the values (see 5.3.1 for reference). Results are depicted in figure 5.10. One can see that the Q-Features learn quick at first, but then converge to a suboptimal solution from which they can not recover. The green line shows the best performing V-Features from the previous section for comparison.

Different setups totally diverged. To do a broader hyper-parameter search was not feasible due to time and resource constraints. Further improvements may stabilize the training, which would be promising, because this approach learned very fast at first. We leave this as a direction for future work.

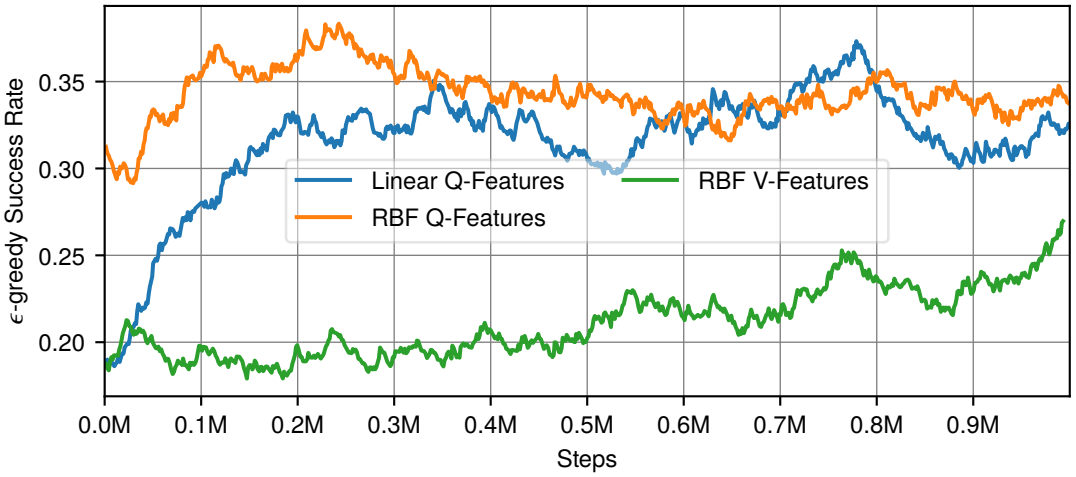


Figure 5.10: Performance of Q-Features compared to the best performing V-Feature RBF kernel. Plot is limited to the first million of steps.

6 Conclusion and Outlook

Throughout this thesis I have given an overview over various existing approaches to integrate planning computations in a neural network. We have seen that various planning algorithms can be expressed in a fully-differentiable way in order to be integrated in a neural network, which allows end-to-end training with data.

In that context, these planning modules exploit the shape of latent representations to generate predictive features, that summarize information about the future. Thereby we analysed the approach of auxiliary losses and other methods to restrict the shape of latent representations. We have also shown the down-side of such approaches concerning bias and training stability.

So we have seen that acquired latent representations matter for both, meaningful planning computations and learning a value function or policy. Therewith we explored different approaches to augment latent representations for the current state with predictive features from a planning module and summarized the ideas in a Learning to Plan Framework.

We applied the ideas incorporated in the framework to a visual navigation task. Although, the chosen task during experiments was comparably simple, it showed that the ideas can be effectively combined. Furthermore, we improved the latent representation with learned value features and showed that it can improve the data-efficiency during learning for that particular task.

But we have also seen that convergence and performance highly depends on hyper-parameters. To draw a firm conclusion one would need to analyse the effect of hyper-parameters in a broader setup. This was not feasible due to time and resource constraints during this thesis.

Nevertheless, I consider the approach of enhancing latent representations with incorporated future information as an approach worth to further follow on the (long) way towards *Learners that Plan*.

Outlook

There are several directions that future work may consider:

Other Tasks: During the experiments we evaluated the acquired representations on the same navigation task as the value features have been trained. It would be interesting to see how 'knowing to navigate' helps the learning of other tasks in the environment.

Hierarchical Approach: During the experiments we have seen that the value features outperformed the baseline in both data-efficiency and overall performance. An interesting direction would be to explore, if latent representation thereby can iteratively improve and so establish a hierarchy of abstract representations constituting the agents current state.

Harder Navigation Tasks: The task during experiments was chosen easy due to time constraints. It's left for the future to evaluate the approach with harder navigation tasks.

Q-Features: Although Q-Features performed rather badly during the experiments, we consider those as a promising direction. Future work may consider other network structures and a broader optimization of hyper-parameters.

Exploration: Knowing to navigate in an environment opens new possibilities for better exploration strategies in other tasks.

Other domains: Last but not least, the ideas incorporated in *Learning to Plan* are not limited to visual navigation tasks. Other domains can be explored in future work.

A Appendix

A.1 Implementation Details

All crucial hyper-parameters are derived from [AWR+17]. Other parameters follow the default implementations[1] of Tensorflow [MAP+15].

So for learning the Q-values based on landmark activations, a network with three fully connected layer and 64 neurons each has been chosen. All with *ReLU* activations, except for the last layer. The training is performed for 200 epochs, that consist of 50 cycles each. A cycle consists of 16 episodes of acting ϵ-greedy in the environment, followed by 40 optimizations with mini-batches of size 128 sampled from the replay buffer. At the end of the cycle, 16 episodes of greedy evaluation are performend and finally the weight are synchronised with the fixed weights of a target network Q^-.

For the V-value feature setup a trained network with the aforementioned characteristics is used to preprocess raw ICA activations. Therefore, weights are fixed. Values queried from that network (for a set of landmarks) are the input for a value network with exactly the same setup.

The ICA/SFA activations are trained with the tools provided by [SW13]. The outputs of the learned network, which depend on the current observation, are simulated to allow faster training on a headless server without OpenGL rendering. Considering that, the trained ICA/SFA network is sampled at each unit position and direction in the environment as a pre-phase. During training on a headless server, this data is linearly interpolated based on the ground-truth position and orientation of the agent.

A.2 DSR Experiments

In the following we will discuss how DSR, as introduced in section 3.6.1, can be applied to the Atari Games included in the OpenAI Gym [BCP+16] benchmark environments. For the experiments the game *Pong* was chosen, because it can be solved with comparably little computing power.

For the implementation[2] of DSR all hyper-parameters have been chosen according to Kulkarni et al. [KSGG16], except the optimizer.

[1] Implementations will be available at https://timphillip.github.io/bsc_thesis.html
[2] Reproduction of DSR will be available at https://timphillip.github.io/dsr.html

Figure A.1 shows the rewards accumulated over the entire episode for different optimizers, that have been selected from the optimizer implementations provided by Tensorflow [MAP+15]. Hereby a match is considered as an episode. It follows the rules of 21-point matches in table tennis. The agent receives a reward of +1 for every point made and −1 for every opponent's point. So an episode of −21 is the worst to achieve.

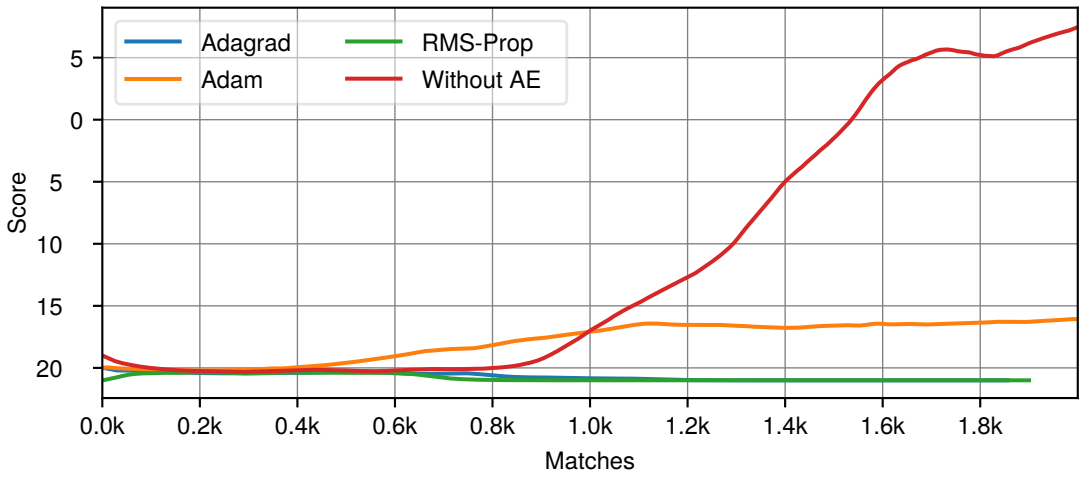


Figure A.1: Results for applying DSR to Atari Pong. It shows the mean reward of the last 100 matches played.

So for the *Adagrad* and the *RMS-Prop* optimizer the learning totally diverged. Surprisingly, with the use of an *Adam* optimizer, the learning at least converged to a suboptimal solution. But performance is still quite bad.

So the network behaved very unstable unless removing the auto-encoder (AE) and corresponding auxiliary reconstruction loss part. With this change the agent is able to actually win matches. So we can conclude that for this setup the auxiliary loss did not help the learning. It rather led to complete divergence.

A.3 Zusammenfassung

Zu lernen ist eine der wichtigsten Fähigkeiten intelligenter und anpassungsfähiger Agenten. Wie gut Algorithmen des Maschinellen Lernens generalisieren und wie effizient diese lernen, hängt aber maßgeblich von den abstrakten Repräsentationen ab, welche sie erlernen. Im Gegensatz dazu erlaubt es Planung den Agenten, die Konsequenzen von gewählten Aktionen zu antizipieren. Dabei lassen sich „Features“ gewinnen, welche den aktuellen Zustand des Agenten unter Einbezug der möglichen Zukunft sehr gut beschreiben. Diese "Featuresërlauben somit daten-effizienteres Lernen. Diese Abschlussarbeit befasst sich mit der Effektivität solcher Ansätze und beantwortet mit Hilfe einer Literaturübersicht die Fragen: (1) Wie können Planungsmodule in neuronale Netze integriert werden und (2) wie lassen sich dabei gewonnene „Features“ am besten einsetzen, um das Lernen der Netze zu verbessern.

Zu den wichtigsten Beiträgen dieser Abschlussarbeit zählt zudem die Synthese der Ideen in einem Algorithmus zur visuellen Navigation und dessen Evaluierung in Experimenten.

Bibliography

[Alt] J. Altosaar. *What is a Variational Autoencoder?* URL: https://jaan.io/what-is-variational-autoencoder-vae-tutorial/ (cit. on p. 25).

[AWR+17] M. Andrychowicz, F. Wolski, A. Ray, J. Schneider, R. Fong, P. Welinder, B. McGrew, J. Tobin, O. Pieter Abbeel, W. Zaremba. "Hindsight Experience Replay". In: *Advances in Neural Information Processing Systems 30*. Ed. by I. Guyon, U. V. Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, R. Garnett. Curran Associates, Inc., 2017, pp. 5048–5058. URL: http://papers.nips.cc/paper/7090-hindsight-experience-replay.pdf (cit. on pp. 35, 51).

[BCP+16] G. Brockman, V. Cheung, L. Pettersson, J. Schneider, J. Schulman, J. Tang, W. Zaremba. *OpenAI Gym*. 2016. eprint: arXiv:1606.01540 (cit. on p. 51).

[BCV13] Y. Bengio, A. Courville, P. Vincent. "Representation learning: A review and new perspectives". In: *IEEE transactions on pattern analysis and machine intelligence* 35.8 (2013), pp. 1798–1828 (cit. on pp. 17, 24).

[BDM+17] A. Barreto, W. Dabney, R. Munos, J. J. Hunt, T. Schaul, H. P. van Hasselt, D. Silver. "Successor Features for Transfer in Reinforcement Learning". In: *Advances in Neural Information Processing Systems 30*. Ed. by I. Guyon, U. V. Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, R. Garnett. Curran Associates, Inc., 2017, pp. 4055–4065. URL: http://papers.nips.cc/paper/6994-successor-features-for-transfer-in-reinforcement-learning.pdf (cit. on p. 29).

[Bel57] R. E. Bellman. "Dynamic Programming". In: (1957) (cit. on pp. 11, 14).

[BPW+12] C. B. Browne, E. Powley, D. Whitehouse, S. M. Lucas, P. I. Cowling, P. Rohlfshagen, S. Tavener, D. Perez, S. Samothrakis, S. Colton. "A Survey of Monte Carlo Tree Search Methods". In: *IEEE Transactions on Computational Intelligence and AI in Games* 4.1 (Mar. 2012), pp. 1–43. DOI: 10.1109/tciaig.2012.2186810 (cit. on p. 15).

[FRIW17] G. Farquhar, T. Rocktäschel, M. Igl, S. Whiteson. "TreeQN and ATreeC: Differentiable Tree Planning for Deep Reinforcement Learning". In: *CoRR* abs/1710.11417 (2017). arXiv: 1710.11417. URL: http://arxiv.org/abs/1710.11417 (cit. on pp. 17, 24–28, 31, 33).

[Gal] P. Galeone. *Convolutional Autoencoders*. URL: https://pgaleone.eu/neural-networks/2016/11/24/convolutional-autoencoders/ (cit. on p. 25).

[GDL+17] S. Gupta, J. Davidson, S. Levine, R. Sukthankar, J. Malik. "Cognitive mapping and planning for visual navigation". In: *arXiv preprint arXiv:1702.03920* 3 (2017) (cit. on pp. 22, 23, 31).

[GS07] S. Gelly, D. Silver. "Combining Online and Offline Knowledge in UCT". In: *Proceedings of the 24th International Conference on Machine Learning*. ICML '07. Corvalis, Oregon, USA: ACM, 2007, pp. 273–280. ISBN: 978-1-59593-793-3. DOI: 10.1145/1273496.1273531. URL: http://doi.acm.org/10.1145/1273496.1273531 (cit. on p. 15).

[GSC99] F. A. Gers, J. Schmidhuber, F. Cummins. "Learning to forget: Continual prediction with LSTM". In: (1999) (cit. on p. 34).

[KHL17] P. Karkus, D. Hsu, W. S. Lee. "QMDP-Net: Deep Learning for Planning under Partial Observability". In: *Advances in Neural Information Processing Systems 30*. Ed. by I. Guyon, U. V. Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, R. Garnett. Curran Associates, Inc., 2017, pp. 4694–4704. URL: http://papers.nips.cc/paper/7055-qmdp-net-deep-learning-for-planning-under-partial-observability.pdf (cit. on p. 23).

[KSGG16] T. D. Kulkarni, A. Saeedi, S. Gautam, S. J. Gershman. "Deep successor reinforcement learning". In: *arXiv preprint arXiv:1606.02396* (2016) (cit. on pp. 26, 29, 34, 51).

[Lin92] L.-J. Lin. "Self-improving reactive agents based on reinforcement learning, planning and teaching". In: *Machine learning* 8.3-4 (1992), pp. 293–321 (cit. on p. 16).

[MAP+15] Martın Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Y. Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dandelion Mané, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, Xiaoqiang Zheng. *TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems*. Software available from tensorflow.org. 2015. URL: https://www.tensorflow.org/ (cit. on pp. 41, 51, 52).

[MBM+16] V. Mnih, A. P. Badia, M. Mirza, A. Graves, T. P. Lillicrap, T. Harley, D. Silver, K. Kavukcuoglu. "Asynchronous Methods for Deep Reinforcement Learning". In: *CoRR* abs/1602.01783 (2016). arXiv: 1602.01783. URL: http://arxiv.org/abs/1602.01783 (cit. on p. 13).

[MKS+15] V. Mnih, K. Kavukcuoglu, D. Silver, A. A. Rusu, J. Veness, M. G. Bellemare, A. Graves, M. Riedmiller, A. K. Fidjeland, G. Ostrovski, S. Petersen, C. Beattie, A. Sadik, I. Antonoglou, H. King, D. Kumaran, D. Wierstra, S. Legg, D. Hassabis. "Human-level control through deep reinforcement learning". In: *Nature* 518 (Feb. 2015), p. 529. URL: http://dx.doi.org/10.1038/nature14236 (cit. on pp. 9, 13, 16, 24).

[NCG+17] S. Niu, S. Chen, H. Guo, C. Targonski, M. C. Smith, J. Kovacevic. "Generalized Value Iteration Networks: Life Beyond Lattices". In: *CoRR* abs/1706.02416 (2017). arXiv: 1706.02416. URL: http://arxiv.org/abs/1706.02416 (cit. on pp. 22, 23, 31).

[Ola] C. Olah. *Understanding LSTM Networks*. URL: http://colah.github.io/posts/2015-08-Understanding-LSTMs/ (cit. on p. 23).

[OSL17] J. Oh, S. Singh, H. Lee. "Value Prediction Network". In: *Advances in Neural Information Processing Systems 30*. Ed. by I. Guyon, U. V. Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, R. Garnett. Curran Associates, Inc., 2017, pp. 6118–6128. URL: http://papers.nips.cc/paper/7192-value-prediction-network.pdf (cit. on pp. 26–28, 33).

[PAS18] M. Pflueger, A. Agha, G. S. Sukhatme. *Soft Value Iteration Networks for Planetary Rover Path Planning*. 2018. URL: https://openreview.net/forum?id=Sktm4zWRb (cit. on p. 22).

[PGDL18] V. Pong, S. Gu, M. Dalal, S. Levine. "Temporal Difference Models: Model-Free Deep RL for Model-Based Control". In: *CoRR* abs/1802.09081 (2018). arXiv: 1802.09081. URL: http://arxiv.org/abs/1802.09081 (cit. on pp. 14, 35).

[Roj13] R. Rojas. *Neural networks: a systematic introduction*. Springer Science & Business Media, 2013 (cit. on p. 19).

[SB98] R. S. Sutton, A. G. Barto. *Reinforcement Learning: An Introduction (Adaptive Computation and Machine Learning)*. A Bradford Book, 1998. ISBN: 9780262193986 (cit. on pp. 11, 12).

[SHGS15] T. Schaul, D. Horgan, K. Gregor, D. Silver. "Universal Value Function Approximators". In: *Proceedings of the 32nd International Conference on Machine Learning*. Ed. by F. Bach, D. Blei. Vol. 37. Proceedings of Machine Learning Research. Lille, France: PMLR, July 2015, pp. 1312–1320. URL: http://proceedings.mlr.press/v37/schaul15.html (cit. on p. 17).

[SHM+16] D. Silver, A. Huang, C. J. Maddison, A. Guez, L. Sifre, G. van den Driessche, J. Schrittwieser, I. Antonoglou, V. Panneershelvam, M. Lanctot, S. Dieleman, D. Grewe, J. Nham, N. Kalchbrenner, I. Sutskever, T. Lillicrap, M. Leach, K. Kavukcuoglu, T. Graepel, D. Hassabis. "Mastering the game of Go with deep neural networks and tree search". In: *Nature* 529 (Jan. 2016), p. 484. URL: http://dx.doi.org/10.1038/nature16961 (cit. on pp. 9, 15, 17).

[SHS+17] D. Silver, T. Hubert, J. Schrittwieser, I. Antonoglou, M. Lai, A. Guez, M. Lanctot, L. Sifre, D. Kumaran, T. Graepel, T. P. Lillicrap, K. Simonyan, D. Hassabis. "Mastering Chess and Shogi by Self-Play with a General Reinforcement Learning Algorithm". In: *CoRR* abs/1712.01815 (2017). arXiv: 1712.01815. URL: http://arxiv.org/abs/1712.01815 (cit. on p. 18).

[SMD+11] R. S. Sutton, J. Modayil, M. Delp, T. Degris, P. M. Pilarski, A. White, D. Precup. "Horde: A scalable real-time architecture for learning knowledge from unsupervised sensorimotor interaction". In: *The 10th International Conference on Autonomous Agents and Multiagent Systems-Volume 2*. International Foundation for Autonomous Agents and Multiagent Systems. 2011, pp. 761–768 (cit. on p. 38).

[SSS+17] D. Silver, J. Schrittwieser, K. Simonyan, I. Antonoglou, A. Huang, A. Guez, T. Hubert, L. Baker, M. Lai, A. Bolton, et al. "Mastering the game of go without human knowledge". In: *Nature* 550.7676 (2017), p. 354 (cit. on p. 18).

[SW13] F. Schoenfeld, L. Wiskott. "RatLab: an easy to use tool for place code simulations". In: *Frontiers in Computational Neuroscience* 7 (2013), p. 104. ISSN: 1662-5188. DOI: 10.3389/fncom.2013.00104. URL: https://www.frontiersin.org/article/10.3389/fncom.2013.00104 (cit. on pp. 25, 26, 41, 42, 51).

[TLA16] A. Tamar, S. Levine, P. Abbeel. “Value Iteration Networks”. In: *CoRR* abs/1602.02867 (2016). arXiv: 1602.02867. URL: http://arxiv.org/abs/1602.02867 (cit. on pp. 19–23, 31, 33, 34).

[WRR+17] T. Weber, S. Racanière, D. P. Reichert, L. Buesing, A. Guez, D. J. Rezende, A. P. Badia, O. Vinyals, N. Heess, Y. Li, R. Pascanu, P. Battaglia, D. Hassabis, D. Silver, D. Wierstra. “Imagination-Augmented Agents for Deep Reinforcement Learning”. In: (July 19, 2017). arXiv: 1707.06203v2 [cs.LG] (cit. on pp. 15, 34).

[WSBR15] M. Watter, J. Springenberg, J. Boedecker, M. Riedmiller. “Embed to Control: A Locally Linear Latent Dynamics Model for Control from Raw Images”. In: *Advances in Neural Information Processing Systems 28*. Ed. by C. Cortes, N. D. Lawrence, D. D. Lee, M. Sugiyama, R. Garnett. Curran Associates, Inc., 2015, pp. 2746–2754. URL: http://papers.nips.cc/paper/5964-embed-to-control-a-locally-linear-latent-dynamics-model-for-control-from-raw-images.pdf (cit. on pp. 24, 26, 30).

All links were last followed on April 28, 2018.

Declaration

I hereby declare that the work presented in this thesis is entirely my own and that I did not use any other sources and references than the listed ones. I have marked all direct or indirect statements from other sources contained therein as quotations. Neither this work nor significant parts of it were part of another examination procedure. I have not published this work in whole or in part before. The electronic copy is consistent with all submitted copies.

---

place, date, signature