Master Thesis

Bielefeld University

Faculty of Technology

Memory-based online learning of simple object manipulation

In Partial Fulfillment of the Requirements for the Degree of

Master of Science (M.Sc.)

Jan Pöppel

Intelligent Systems

November 2015

Referee: Apl. Prof. Dr.-Ing. Stefan Kopp

Referee: Dipl.-Inf. Maximilian Panzner

Abstract

The classical machine learning approach is to train suitable models on reasonable big data sets before deployment. However, this approach is not suitable in many scenarios such as robotics due to scarcity of data. Furthermore, general purpose robots are expected to be able to deal with unknown circumstances. This requires the ability to incrementally adapt to unknown situations.

This thesis considers learning about simply object manipulations such as pushing as a simplified scenario of adapting to unknown environments. Within this context, this thesis suggests and discusses two memory-based concepts that challenge the tasks of incremental online learning with limited domain knowledge. The concepts learn forward and inverse models of their environment, allowing them to predict future states as well as incrementally reach desired target states.

The learned forward and inverse models of the environment's dynamics are evaluated separately in different scenarios. The evaluations show that the proposed methods both successfully learn to predict future states and interact with their given environment while providing insight of remaining problems.

Contents

Chapter 1

Introduction

Inspired by the learning capabilities of humans, the field of machine learning tries to develop methods that allow machines to learn from data. After decades of research a vast amount of powerful tools has been developed for machine learning. Each of these methods has its own advantages and disadvantages and are applicable to certain situations. Currently, research in machine learning is often performed by choosing a problem, e.g. image recognition or movement control, and trying to find and tune the most successful method to solve the chosen problem. This includes for example fine tuning the features that are used to represent the given data. This way the researchers were able to create better classifiers for image recognition (e.g. [1]) and better controllers for complex movements (e.g. [2]). A good overview over different machine learning approaches can be found in the book by Bishop [3].

In most of these cases the machine is trained to solve one specific problem. Depending on the chosen methods it is not possible to extend the machines knowledge easily without retraining the entire system afterwards due to the stability-plasticity-problem [4]. More precisely because of the phenomenon of catastrophic forgetting[1] [5].

In recent years, deep learning [6, 7] has attracted a lot of attention by achieving great results in high dimensional tasks such as image classification [8]. The ability of deep learning approaches to automatically extract relevant features from high dimensional data is one of the main reasons for the strong interest. However, so far deep learning requires a lot of training data to be successful which is the main reason why it is not further considered in this thesis.

With advances in robotic hardware and the successful application of machine learning in specialized tasks, such as image recognition, the goal of robotic research trends towards

[1] The phenomenon of forgetting previously learned information after receiving new information. Most prominent in neural networks and when trying to learn non-stationary data.

multi-purpose robots. However, a big problem for multi-purpose machines is that the current approach to machine learning is not applicable. Since the number of tasks the robot has to face is not known in advance, it is difficult to train suitable tools beforehand. Furthermore, even if one would attempt to train specialized parts for all kinds of problems, acquiring sufficient and accurate training data in advance is infeasible at best. On top of that, robotic platforms often only have limited computational resources which limits the amount of data they can process at once.

Therefore, instead of trying to train the machine beforehand, it might be better to provide the robot with the means to adapt to new situations on its own while it is encountering them. This continuous learning is also referred to as lifelong learning and resembles the human learning process more closely [9]. Instead of learning a complete model[2] on previously recorded training data, the robot adapts its model to the continuous stream of data while it is already using what it has learned so far. The biggest reason against such an continuous incremental approach is its difficulty. When incrementally training a single model to solve multiple different problems, the catastrophic forgetting effect is often experienced. Furthermore, different kind of tasks may require different features which makes training a single model infeasible. The usual approach is to learn local models for each task separately, however this introduces the need to recognize and distinguish the different problems in order to know which kind of local model the robot needs to employ.

Instead of challenging the entire problem of incrementally learning an unlimited number of arbitrary tasks, this thesis concentrates on the incremental learning of one task without prior training. While there are multiple machine learning methods that allow incremental updates, not all of them are suitable for this kind of task. First of all, the method should be as independent of prior knowledge or the used features as possible so that it can be used for a wide variety of task the robot might encounter. Furthermore, the update and query times of the chosen method need to be quick enough to allow continuous interaction with the environment. On top of that, the chosen method should not suffer from the catastrophic forgetting effect since the robot would constantly keep updating its model.

One important aspect of general purpose robotics is object manipulation. In order to successfully interact with the objects in its environment the robot needs to learn what kind of interactions are possible and what their effects are. Furthermore, object interactions are hard to model manually as they follow complex dynamics. On top of that, different object can behave completely differently, so that knowledge acquired in previous training sessions might not be useful later on. Consequently, object interactions make an ideal target for online self adaptation.

This thesis presents and compares two concepts that provide incremental learning of push-

[2] A model represents the robot's knowledge about the world.

ing interactions between an actuator[3] controlled by action primitives and some object in the environment. While pushing interactions are only a very small subset of possible interactions a robot can have with objects, their dynamics still provide sufficient complexity to evaluate incremental learning systems. Since the behavior of differently shaped objects can vary a lot, learning about different kind of objects can even be regarded as learning similar but different tasks. The proposed concepts need to provide a forward model as well as an inverse model. The forward model makes predictions about the state of all entities in the environment after an action primitive has been performed. The inverse model provides an action primitive that is used to reach a specified target configuration within the environment.

Memory-based approaches, such as Growing Neural Gas (GNG) have already been successfully used in robotic scenarios in order to solve the problems associated with incremental learning [10]. Due to their one-shot learning ability, memory-based methods can produce good prediction results from very little training data. It is for these reasons, that this thesis also focuses on a memory-based approach. An adaption of the GNG is developed and used as the underlying regression and classification method for the developed concepts. Furthermore, a prototype based inverse model is developed in order to allow extrapolation when deducing suitable action primitives.

The goal of this thesis can be summarized to provide and evaluate simple models that:

1. Update themselves incrementally during the interaction
2. Allow predictions of simple object interactions
3. Allow the deduction of action primitives required to reach a given target

The remainder of this thesis is structured as follows: In chapter 2 an overview of memory-based learning as well as an outline of recent research concerning incremental learning and object manipulation in the context of robotics is given. Two concepts were developed in order to fulfill the stated goals. Their general idea is described in chapter 3 before concrete implementation details are given in chapter 4. These models are then evaluated with regards to the goal mentioned above in chapter 5. The evaluation is discussed in chapter 6 before this thesis thesis concludes in chapter 7.

[3]The part of a robot that acts upon its environment.

Chapter 2

Related Work

2.1. Memory-based models

In machine learning, one generally assumes that data, for example class labels, are given by some function f^*. Given some training data D with $d^i = (\vec{x}^i, \vec{y}^i)$ where $\vec{x}^i$ represents the given input and $\vec{y}^i$ the corresponding output of the i's data point, one tries to estimate f^*. Although different algorithms approximate f^* differently, they can be categorized in two general families: The first one tries to detect correlations between features in order to make predictions. The simplest example for this is given by the linear regression [11] where the goal is to find weights w_j so that $f(\vec{x}) = \sum w_j \cdot x_j$.

The other family focuses more on similarities in the input space instead of feature correlations. These methods perform what is often called instance- or memory-based learning. One of the most prominent examples for this is the k-Nearest Neighbor (k-NN) regression [12]. Previously seen training data is stored and compared to new input data. New input data $\vec{x}$ is then labeled according to the k closest stored instances. The easiest form is to average the outputs of the k closest instances around $\vec{x}$ (here denoted as the set $N(\vec{x})$):

$$\vec{y}_{new}(\vec{x}) = \frac{1}{k} \sum_{\vec{y} \in N(\vec{x})} \vec{y} \tag{2.1}$$

These instances are often called prototypes, if only a few instances are used to represent a subspace in the input space instead of all training instances. Since the closest instance is determined by similarity, the used features and metric is usually critical for these methods (see for example [13, 14]). This dependence on careful preprocessing of the used features and metric can be considered the biggest disadvantage of these methods. However, there is a lot of ongoing research regarding automatically adapting the used metric while training

in order to better fit the data, e.g. by Hammer et al. [15]. Unfortunately, online metric adaptation needs a lot of iterations before yielding satisfying results in most cases.

Another disadvantage of memory-based models is the memory consumption over time. Since the training examples need to be stored in order to be retrieved later, a lot of memory can be consumed. This can lead to increased query and update times, since these scale linearly with the number of stored instances. In 1991 Geva et al. showed that the number of required instances can be reduced depending on the used algorithm [16]. Extending the area of influence of the given instances, for example by using Local Linear Maps (LLMs) [17], can further decrease the number of required instances.

The advantages of memory- or instance-based models is that they are local by design. This means that their output only depends on a small part, located around the given input, of the model. Likewise, when they are updated by adding or removing some stored instances, they do so based on local criteria. This is also the reason why memory-based models usually do not suffer from the catastrophic forgetting phenomenon.

Another reason to look into memory-based learning is the evidence for episodic memory in humans [18]. These findings suggest that humans store past experiences in order to utilize them in new situations. This general idea is reflected in memory-based learning approaches.

For the implementation of the concepts developed in this thesis, an adaptation of the GNG using LLM as output function is used as underlying regression and classification model. This adaptation is called Adapted Instantaneous Topological Map (AITM) for the remainder of this thesis and is explained in section 4.4.2 in detail.

2.2. Incremental online learning

In order to deal with non stationary data, i.e. data whose characteristics change over time, as well as processing data streams continuously with limited resources, incremental learning is gaining increasing attention [19]. Usually memory- or instance-based approaches are adapted to allow incremental training, such as the Support Vector Machine (SVM) for classification [20]. Non stationary data as well as limited processing power are two constraints researchers in the field of robotics have to deal with. As such, the robotic research community is focusing more on the use of such methods: Carlevarino et al. successfully train an adaption of the GNG to incrementally learn an inverse model for robot control [10]. Losing at al. use a different memory based approach with the Learning Vector Quantization (LVQ) in order to incrementally train an obstacle classifier on a robotic platform [21]. Unfortunately, all these works require a relative large amount of training examples before

a good performance is achieved. In the situation discussed in this thesis, the developed model should be able to produce reasonable results after only a few interactions.

Liu et al. propose an biology-inspired adaptation of the Markov Decision Process (MDP) by using episodic memories [22, 23]. Their approach incrementally learns an experience map of an uncertain environment. Unfortunately, their approach requires the definition of fixed goal states or targets. Furthermore, transitions between different goals are difficult since one episode, or one EM-MDP is learned for each goal. In the scope of this thesis, a robot should be able to learn about the interactions in the environment without being provided additional information, such as goal states or rewards for subgoals, by humans.

2.3. Learning object interactions

2.3.1. Human object manipulation

While it was never the goal of this thesis to provided biology-inspired models of object interactions, there are some indications in recent findings that suggest similar processes in human brains to the ones developed here:
A good overview about recent findings in neurobiology is given in [24]. One of these findings suggest that contact events are especially important in order to synchronize multi-modal information [25]. While it is not enough to wait for a contact event in this context, the second concept, described in section 3.3 trains a gating function specifically designed to predict when one object influences another.

Already in 1999, Kawato analyzed internal object specific forward and inverse models in humans [26]. Multiple studies have been performed successfully in order to find evidence of the existence of internal object specific models, e.g. [27, 28]. Both developed concepts employ local models in order to reduce the size of the learned space. Especially the findings of Flanagan et al. [27] that suggest size-weight specific internal models for objects correlates to the employed idea to train local models for different object groups in the second concept.

2.3.2. Robotic object manipulation

Robots are supposed to interact with and manipulate their environment in order to be useful. Therefore, a lot of research is being done regarding object manipulation.

Much of the earlier work regarding object manipulation was concentrated in grasping objects. A good overview on different approaches and challenges is provided in [29]. Having

determined successful grasp types and strategies, more recent work combines grasping with other modalities such as vision [30]. Here Saxena et al. train their system to detect suitable grasp locations in images from novel objects. Deep learning is also used for the same problem in order to avoid crafting features by hand [31]. While grasping is very important for general-purpose robots, the proposed solutions do not learn about the object's dynamics. In fact these systems are usually considered trained successfully if the objects do not move until they have been grasped. This thesis on the other hand concentrates on learning object dynamics as an exemplary scenario for the incremental acquisition of knowledge in unknown environments.

Once objects have been grasped the robot still needs to learn to manipulate them in a desired way. In most cases, a policy[1] is learned in order to reach a given target configuration. Many different methods of learning policies in different scenarios have been proposed. A good overview on different strategies developed for robotics is given by Deisenroth et al. [32]. Recently, Levine et al. achieve great results in learning policies of complex dynamics such as assembling plastic toys [33]. The authors report that they require much fewer training iterations than earlier policy search approaches. They achieve this by training linear-Gaussian controllers on trajectories which are later combined by a guided policy search component. However, the main difference to the given problem for this thesis is that Levine et al. consider the objects already firmly grasped which eliminates most of the dynamics in the interaction between the grasped object and the actuator. Furthermore, they specify trajectory specific distance scores in order to optimize the desired trajectory which makes their approach unsuited for unsupervised incremental learning.

Other works are often train their systems in an offline fasion, e.g. [34, 35, 36]. Out of these, the works of Moldovan et al. and Kroemer et al. are most notable with regard to the topic of this thesis: Moldovan et al. use a probabilistic approach to learn affordance models useful for object interactions [35]. Affordances, as introduced by Gibson [37], describe the relations between object, actions and effects. Moldovan et al. are able to extend their approach to multiple object scenarios. The biggest difference of their approach to the here presented one, apart from the required offline training and tuning, is that it uses more prior knowledge in order to construct a suitable Bayesian Network (BN).

More recently, Kroemer et al. developed a memory-based model that uses the similarities of contact distributions between objects in order to classify different kind of interactions [36]. In their work an interaction is given if one object supports another object or if an object can be lifted given a certain grasp configuration. The approach of Kroemer et al. works with multiple objects. Since only the contact distributions between two objects are considered, a classification can be made for each object pair separately. Unfortunately, their approach is limited to binary predictions of predefined interaction classes. The robot still needs to learn a suitable forward and inverse models in order to manipulate the objects. Furthermore,

[1] A policy defines what action primitive is to be used depending on the current situation and target.

according to the authors their proposed sampling approach shows poor performance and is likely to become infeasible in unconstrained complex environments.

The most similar work to this thesis both in terms of setting and in their approach has been done by Lau et al. [38]. The authors use k-NN regression to make predictions about pushing an object. They also provide an algorithm to extract suitable actions that allow to push the object towards a specified target. Unfortunately, the authors restrict their model to the pushing interaction: Moving the actuator around or towards the object is not part of the learned model but is provided instead. Furthermore, while they describe their approach to work with position and orientation, they only provide results for position.

Overall it can be said, that a lot of research is being done regarding online learning and object manipulation in robotics. Many of these approaches are memory- or prototype-based since memory-based approaches can work well under the conditions in robotic scenarios such as non stationary and limited data. This thesis combines ideas from different approaches, such as using local models and episodic like memory, in order to incrementally learn about an unknown environment.

Chapter 3

Concept

In order to deal with the continuous information stream a robot experiences when it interacts with an unknown environment, a robot needs an architecture. This architecture needs to organize the incoming data and determine what should be learned from these experiences. Furthermore, the architecture can combine different components in order to make the best use of what it has already learned. This chapter presents the two concepts that were developed in order to learn about simple object interactions. These two concepts differentiate mainly in the way they represent the interactions that they learn about. The first concept, further described in section 3.2, uses pairwise interactions to represent the objects in the environment. In the alternative approach, described in section 3.3, the objects are represented individually. Additionally, a gating function is introduced to distinguish when one object actually influences another. A special inverse model is proposed in section 3.4 that is tailored to the continuous interaction with the environment. Before going into the description of both concepts and the inverse model, section 3.1 summarizes the actual problem these concepts need to solve.

3.1. Problem specification

As stated in the introduction, the goal of this thesis is to provide possible models that incrementally learn simple pushing interactions between physical objects. More specifically, the models need to learn two tasks:

1) Given some known action primitive that controls the robots actuator, the models need to learn to predict the next states of all objects in the environment (**forward model**). While these action primitives are determined beforehand, the models do not necessarily know the effects of the primitives. In this case, the models need to not only learn about the pushing interactions, but also about the consequences of their own actions.

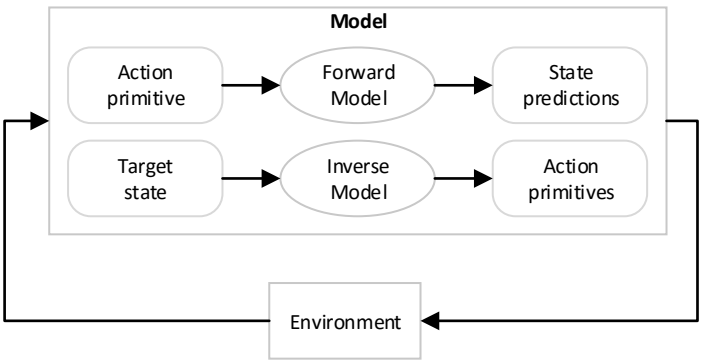

Figure 3.1.: Overview of the problem. The desired models need to include a forward model in order to make predictions given the current environment state and a certain action primitive. When given a target state an inverse model provides suitable action primitives to reach the desired state.

2) The robot is supposed to be able to reach some specified target configuration (inverse model). Any object in the environment, not just the actuator, can be specified which means that the models must learn how they can influence the objects through their actuator. The target configurations will rarely be reachable within a single action primitive but the inverse model should still provide actions that reduce the distance to the target configuration. Since the actuator can only influence another object by pushing it, the actuator might be required to circle around an object in order to be able to move it towards the desired configuration.

A sequence of action primitives that would reach the target after their execution is not requested. This is because the models are expected to react to changes in the environment directly. Furthermore, if such a sequence is required, for example in order to make higher level plans, they can be created using the described forward and inverse models.

Both of these tasks need to be learned incrementally without prior training. This requires the models to be tightly coupled to the environment. An overview about the models interaction with the environment can be seen in figure 3.1. The models are updated each time feedback from the environment is received, improving the quality of following predictions.

3.2. Modeling pairwise interactions

The first approach that is proposed here tries to find distinct subspaces in the **interaction space** between two objects. In this context, the pairwise interaction space represents the space of all interactions between two objects. This includes their relative placement and movement to each other, as well as their influence, for example by pushing, on each other. Instead of modeling and learning forward and inverse models for each object separately, only the pairwise interaction space between two objects is considered.

For every object pair an **interaction state** is considered. This interaction state represents one object's state relative to the other object's state. This includes for example transforming each object's position and orientation to the local coordinate system of the reference object. The predicted object states are extracted from the predicted interaction states. The way these interaction states are computed from the object states and how the object states can be extracted from the predicted interaction states, need to be provided to the model beforehand. Considering the robotic context, this means that the way the robot's sensor inputs are combined into features need to be predetermined.

At each update step the current interaction states for all object pairs are computed from the information provided by the environment. The previous state, the performed action and the resulting state are collected and stored as an **episode** e. The general idea is to store these episodes as past knowledge, similar to the approach in case based reasoning [39]. When predicting, these episodes can be searched for the most similar one. The similarity can be determined by comparing the previous state of each episode to the given interaction state and the used action to the current action:

$$e^{best} = \underset{e^i \in E}{\operatorname{argmin}} \, ||e^i_{preState} - curState, e^i_{action} - curAction||_{ep} \tag{3.1}$$

e^i represents the i's episode that has been stored yet. $e^i_{preState}$ means only the previous interaction state of the episode, while e^i_{action} represents the used action in that episode. The norm $||a, b||_{ep}$ is used to allow different weighting of the features from the previous state and the action. For planning, the action difference can be replaced by the difference between the resulting state of each episode and the desired target state.

Once the most similar episode has been found, the desired information can be extracted from it. For prediction, this corresponds to the resulting state in the episode, whereas it corresponds to the action for the inverse case. The formula above represents a nearest neighbor search on all episodes. Although such an approach can work, it quickly becomes infeasible when the number of stored episode keeps growing. Since the nearest neighbor

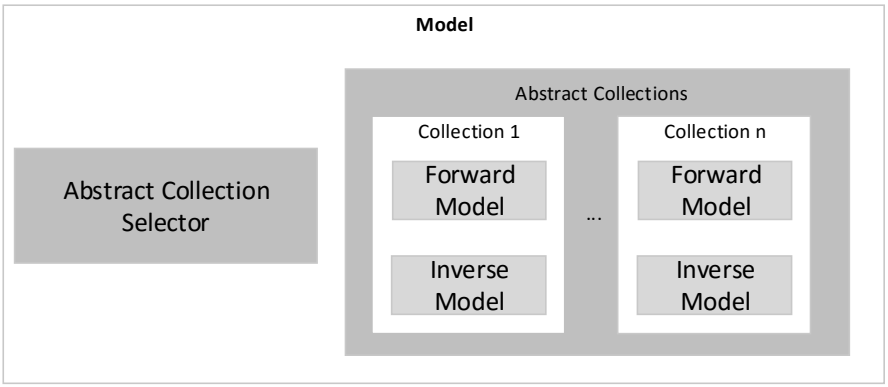

Figure 3.2.: Overview of the interaction state model. The different Abstract Collections contain forward and inverse models of distinct subspaces of the interaction space. The selector is needed in order to select the appropriate local model for prediction.

search scales linearly with the number of stored examples it is unsuitable for lifelong learning. Furthermore, the simple nearest neighbor approach does not offer good generalization since it does not interpolate between similar episodes.

As mentioned at the beginning of this chapter, the idea of this concept is to split the interaction space into subspaces and train local models for each of these subspaces. This concept and it's implementation will often be referred to as interaction state concept/model for the remainder of this thesis accordingly. An overview of the proposed architecture can be seen in figure 3.2. All episodes corresponding to the same subspace are collected in Abstract Collections (ACs). Each collection can train its own local forward and inverse model which is explained in section 3.2.1. An Abstract Collection Selector (ACS) is needed in order to choose the most appropriate AC in a given situation.

3.2.1. Abstract Collection

The interaction space can be split in a multitude of subspaces. Ideally, one would want to split the space in subspaces with some semantic meaning, for example into no interaction, turning and pushing. In this example, the no interaction subspace would correspond to the episodes where the actuator moves without influencing another object. Turning would correspond to an interaction where mainly the orientation of an object changes through the interaction. Lastly, the position of an object changes mainly in the pushing subspace. However, in order to separate the episodes like that, accurate labels are required. Such a

classification is infeasible without prior domain knowledge. Instead of relying on domain knowledge, this approach uses the information available to the robot from the environment: The changing features within an episode.

In order to split the interaction space into subspaces, episodes that correspond to the same local group are clustered together. The clustering of episodes suggested here is based on the set of features that changed between the previous state and the resulting state after performing an action. The set S of features that changed is defined as follows

$$S = \{f | f \in F \ \wedge \ ||Pre(f) - Post(f)|| > \epsilon_{Noise}\} \tag{3.2}$$

where F denotes the set of all features. $Pre(f)$ and $Post(f)$ denote the value of feature f in the initial and resulting state of the episode respectively. ϵ_{Noise} is a threshold to cancel out potential noise of the environment and should be set according to the accuracy of the used sensors. Ideally, one would like to have feature specific thresholds since the possible range of the features might vary. However, this would require the model to have additional information about each feature.

Each different set is represented by its own Abstract Collection. The idea is that, while not necessarily holding semantic meaning directly, these collections correspond to different interaction scenarios. Depending on the features used, some ACs might even be interpreted semantically. Consider an exemplary interaction state with two features. The first feature represents the closest distance between the two objects in the interaction state. The second feature represents the angular direction of the second object with respect to the local coordinate system of the reference object. While the model itself has no knowledge about what each feature represents, a maximum of four ACs can be created: No feature changes, either one of the two features changes and both change at the same time. In this example the AC containing only the distance feature represents all interactions where one object moves straight towards or away from the other object. The set where nothing changes would signal a pushing interaction, considering a movement action was performed. While not all created ACs can be interpreted semantically, the model can create these separations without any knowledge about what is represented by the features.

Since each distinct interaction scenario results in a different set of changing features, the interaction space is split into subspaces by these collections. The resulting collections create an abstract representation for each of these interaction scenarios.

These collections can then train their respective local forward and inverse models based on the episodes associated with them. Any suitable regression method can be used for these local models. The above mentioned nearest neighbor structure that works directly on the recorded episodes is just one simple example. Since the collections are independent of each other, different collections can even use different methods if desired. Furthermore, local optimizations such as feature selection could be performed in each collection.

3.2.2. Object state prediction with the Abstract Collection Selector

Predicting object states within this concept requires several steps since only interaction states are modeled explicitly. The general idea of the information flow when predicting one interaction state is visualized in figure 3.3. The model expects an interaction state that has been computed from the objects' information provided by the environment. This interaction state is used together with the given action primitive as input for the Abstract Collection Selector (ACS). The selector needs to estimate which AC is most likely responsible for the next interaction. In order to make this estimation, any suitable classifier, e.g. a decision tree [40], is trained on all input-collection pairs the model experienced so far. The total number of collections is limited to the size of the superset of F, although in practice, not all possibilities are likely. Furthermore, collections with less than ϵ_{min} episodes could be ignored. This threshold can be used to reduce the number of outliers when training the classifier.

After the most likely AC has been selected, its forward model can be consulted for the prediction. The local model is queried using the interaction state together with the given action primitive as input. The output of the local model is the predicted interaction state. More precisely, the local model predicts how the current interaction state changes within the features defined by its set of changing features S. These predicted changes are then added to corresponding features dimensions of the given interaction state that this AC is responsible for. The predictions of the actual object states need to be extracted from this interaction state. The extraction depends on the actual features used and needs to be provided by the user just like the structure of the interaction state itself.

3.2.3. From target state to action primitive

In order to get a suitable action primitive given a target configuration several steps are required as visualized in figure 3.4. First the difference state between the current situation and the target configuration is computed. Similar to the computation within the episodes above, a difference set S can be computed from this difference state. Afterwards, the AC that corresponds to the same set of features is selected. In the case that this collection is not yet known, the most similar collection is chosen instead. In this context, the most similar AC is the one that covers most of the changed features in the computed difference set. If multiple alternatives exist, the AC responsible for the changes in the most features is chosen. This is because the inverse model has more degrees of freedom to reach the target in that case.

The selected AC queries its own inverse model for preconditions that produce a change in the direction of the desired target configuration. Only returning the corresponding action

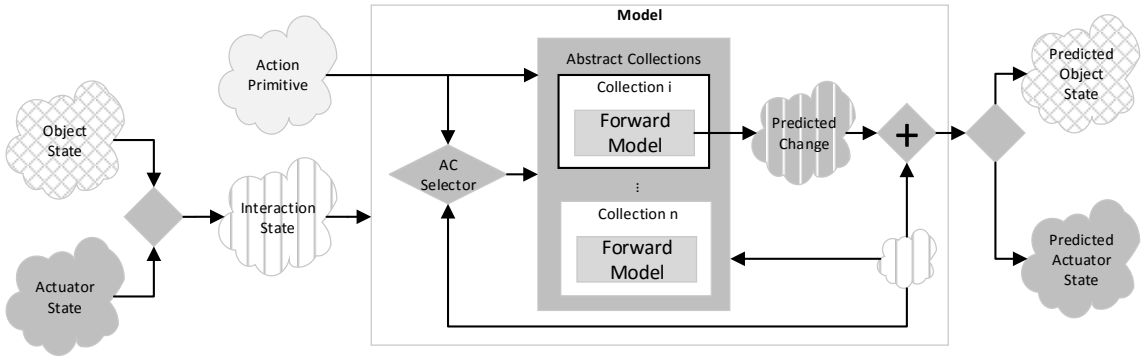

Figure 3.3.: Concept of prediction in the pairwise interaction space with Abstract Collections (ACs). First the interaction state is computed. This state is then fed to the model together with the chosen action primitive. The Abstract Collection Selector (AC Selector) uses those to select the responsible AC. The collection then uses its forward model to predict the changes in the interaction state. These changes are added to the given interaction state in order to compute the resulting prediction. Finally, the actual predicted object states are extracted from this interaction state.

primitive is not sufficient, since it might not be possible to move an object directly in the desired direction due to the current position of the actuator.

The preconditions need to be checked against the current configuration of the target object and the actuator. If these preconditions are mostly fulfilled an action primitive can be extracted, otherwise an intermediate target needs to be determined. This intermediate target represents a configuration for the actuator where the preconditions are met. The actuator might be required to circle around the object in order to reach the intermediate target without influencing the object in a negative way.

3.2.4. Theoretical discussion

The two core ideas of this concept are the use of a pairwise representation and splitting the interaction space. The splitting is done in order to allow the training of simpler local regression models. Furthermore, using local models avoids having to compare to all previously seen training examples. However, this does require the Abstract Collection Selector in order to first select the correct local model. Using a global regression model, for example for the inverse model, is obviously possible as well, especially if the used model does not suffer from the complexity of the used interaction state and the continuous training. Global models have the benefit of not having to determine the responsible Abstract

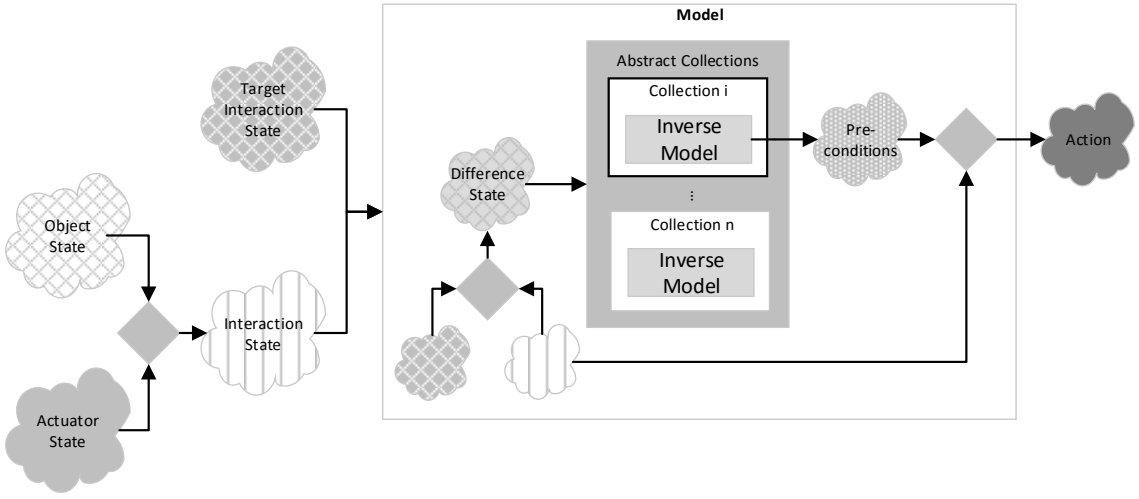

Figure 3.4.: Concept of computing suitable action primitives in order to reach a given target configuration within the interaction state concept. The suitable AC is determined by the Difference State. An action primitive is computed from the preconditions returned by the inverse model.

Collection.

Using pairwise interaction states in order to represent the interactions between objects is often used in robotics, for example in [35, 41]. Its advantage is that it represents both objects involved in an interaction at the same time. This means that only one regression model needs to be trained in order to make predictions about both objects. In theory this should also make it less likely that impossible configurations are predicted such as solid objects being inside of each other. Furthermore, such an interaction state can contain all the necessary information about the objects in one representation.

The downside of this approach is that the actual object states need to be inferred from these interaction states. When using only differential features to construct the pairwise interaction state, this will only involve simple coordinate transformations. Some other object attributes might not be as easily transformable though. Furthermore, in order to contain all the necessary information about both objects, these interaction states can become high dimensional. All computations from object to interaction states and vise versa need to be performed outside this proposed model and need to be predefined.

Although the coupling of the two objects can have its advantages as stated above, the dependence between the objects can also be a problem when predicting. Any erroneous prediction will immediately affect two objects. Furthermore, when one of the two objects

is represented relative to the coordinate frame of the other object, all predictions are also made with regard to the reference object's coordinate frame. In case the state of the reference object is not accurate, for example because of a sensor malfunction, the predictions for both objects will be influenced by this error. This can easily be imagined when considering errors in the reference object's position. Since the second object's position needs to be computed based on the reference object's position, the error will immediately be reflected in the predictions for both objects.

An even bigger disadvantage is the fact that this approach cannot easily be extended to multiple objects. In environments with at least two objects and an actuator at least three pairwise interaction states are necessary: One for the first object and the actuator, one for the second object and the actuator and at least one for both objects. While one can argue for and against using the actuator as the reference object for the general case, depending on the actual scenario, this decision is not trivial between two objects. Even if one finds suitable reference objects, there is still a problem with the extraction of the actual object states after prediction. All objects are part of at least two interaction states in this situation. Therefore, at least two extracted predictions exist for all objects in the environment. Unfortunately, these predictions are likely to be different from another. It is therefore required to compute a final prediction for each object which is not trivial in the general case. Furthermore, the presented concept does not even know about object states, meaning that the final computation would need to be performed outside of the concept. It is also unclear if an AC should only be responsible for the interaction subspace between two specific objects or if its local model should represent the subspace for all object pairs. In most cases, training individual ACs for each object pair is likely to be more successful because similar actions will result in different interactions for different interacting objects.

Regarding the Abstract Collections: Unfortunately, no guarantees can be given about how well the interaction space is split. Depending on the used features, as well as the actual interaction scenarios that are encountered, some abstract collections might cover most of the interaction space.

3.3. Object space with gating function

The intuitive alternative to modeling the interaction space, is to model each object and perform predictions for each object separately. The general components of this architecture are visualized in figure 3.5.

The idea is to train local forward and inverse models for each object, or object group in the Predictor. Instead of distinguishing between different interaction scenarios as in

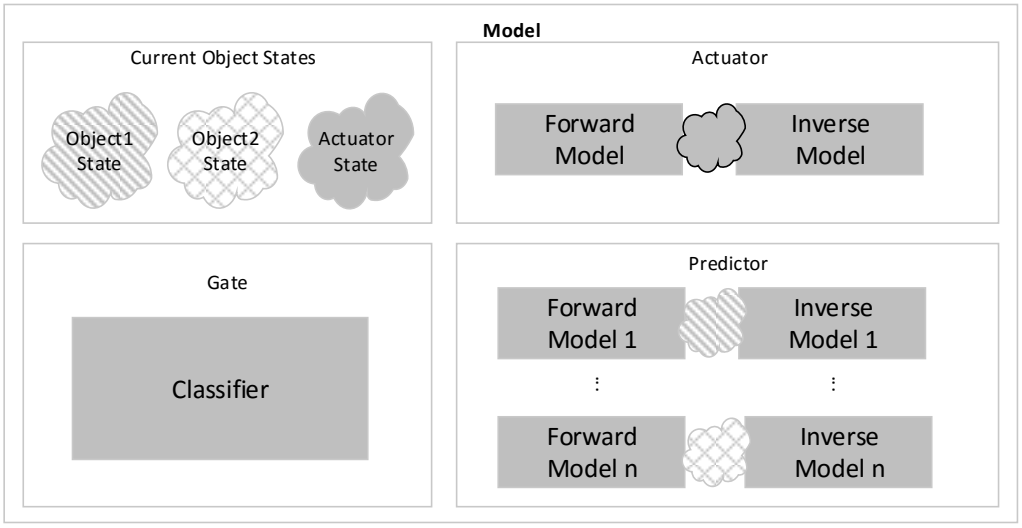

Figure 3.5.: Overview of the concept using object state with a gating function. The actuator is a special object, that can be influenced directly. It's forward and inverse model might even be provided if known. For the other objects/object groups, the Predictor learns these models. The gating function (Gate) learns a classifier to distinguish interactions between objects from situations where only the actuator changes.

the previous concept, objects that behave differently are distinguished. An object group can be considered a collection of objects that are similar in the way they behave during interactions. Therefore they can be represented by a single local model. One example would be two identically shaped block objects that only have different colors. In order to detect if two objects should be grouped together or not, one can start with training separate local models and compare their outputs. If two models appear to be similar, they can be merged together. Ideally, a similarity measure for objects is provided. Following the assumption that similar objects should behave similarly, such a measure would allow immediate grouping of objects.

Apart from the difference in representation, the other big difference in the two concepts is the introduction of the gating function (`Gate` in the figure). The gating function is used to split the interaction space in two big subspaces. The first subspace represents all scenarios where at most the actuator is moving due to some action primitive but no other object is influenced by the actuator. In the given context, this means that no actual pushing interactions are taking place. Consequently, the other subspace contains all the scenarios where an object is influenced. When making predictions for the first subspace, it is sufficient to make predictions about the next state of the actuator. No knowledge about

the behavior of the objects is required which also means that the local models do not need to be trained for these scenarios. The local object models only need to be trained on the scenarios where an object's state actually changes. This makes the local subspaces each model needs to learn already simpler then the interaction space learned in the previous concept. Therefore, multiple models for each object are not required.

In order to allow the local models to only train on this restricted subspace, this concept makes an additional assumption about the environment: This idea assumes that object states do not change without any interaction. The only exception is the Actuator. Here, the actuator is treated as a special kind of object with its own forward and inverse model. These models can be provided beforehand or learned online.

The current state of each object, including the actuator, is always updated when new information is received from the environment. At each update the gating function and if required the actuator's local models are trained. On top of that, the local models responsible for any objects whose states have changed with the last update, are trained as well.

Due to the significance of the gating function, this concept and its implementation will often be referred to as gating concept/model in the remainder of this thesis.

3.3.1. Object state prediction

Because of the assumption that objects other than the actuator cannot change without any interaction, the way prediction is performed differs between the different object types. The actuator simply queries its own forward model with the selected action primitive as input in order to predict the next actuator state. The general process for predicting the next state of other objects is visualized in figure 3.6.

First the next actuator state is predicted as mentioned above using the selected action primitive. This predicted state is then used together with the current state of the object that is to be predicted to compute Relative Interaction Features. These relative features are similar to the pairwise interaction state that was explained in the previous concept. However, since it is not necessary to extract the entire object state for both objects from these relative features, the dimensionality can be a lot lower than before or additional higher level features can be used. In fact, depending on the scenario, it may be sufficient to only represent the actuator in the local coordinate frame of the object. The gating function then uses these features to determine if the object will be influenced by the new actuator state. If the gating function predicts no interaction between the object and the actuator, the current object state is returned. On the other hand, if an interaction is predicted, the Predictor uses the local forward model responsible for the current object. The local model predicts the changes in the object states instead of the final states. In

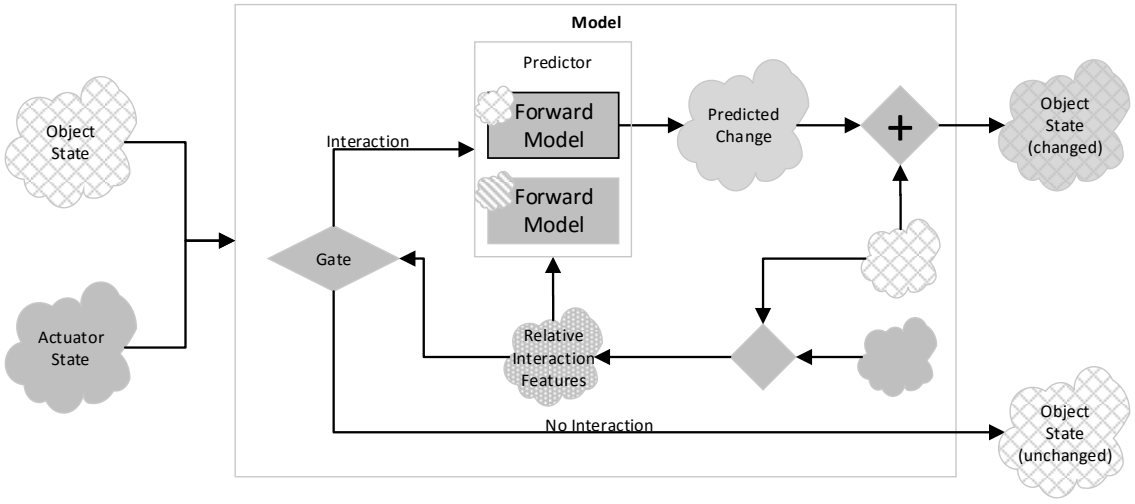

Figure 3.6.: Visualization of the prediction process for non actuator objects in the gating concept. Using the predicted actuator state, relative interaction features are computed. These are used by the gating function to determine if an interaction takes place or not. The responsible local forward model is used to predict changes in the object's state if an interaction is expected. From these changes the actual state prediction is computed.

order to get the actual predicted object state, these changes are added to the current object state.

When more than one object other than the actuator is present, the same process can be repeated. In order to predict interactions between two non actuator objects, a prediction chain is performed. First the actuator is predicted as described above. Afterwards, the objects that are directly influenced by the actuator are predicted. The predictions of these objects can then in turn be used as new "actuators" for interactions between them and other objects. However, additional problems arise as explained in section 3.3.3 below.

3.3.2. From target object state to action primitive

Similar to prediction, computing a suitable action primitive is performed differently depending on what kind of object is supposed to reach a given target configuration. In the case that the target object is the actuator, its inverse model is queried for an action primitive with the target configuration as input.

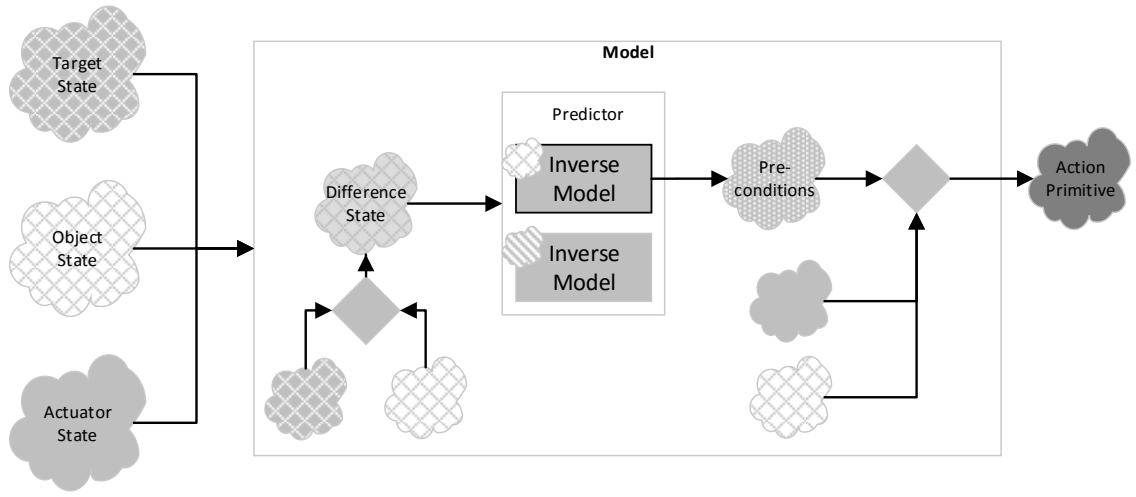

Figure 3.7.: Visualization of the process of computing action primitives to move non actuator objects towards a target configuration in the gating concept. Using the Difference State as input, the local inverse model responsible for the target object are queried for Preconditions. The resulting action primitive is computed from these preconditions.

The more interesting process of moving towards a target configuration for a non actuator object is visualized in figure 3.7.

First the current difference state is computed from the given target configuration and the current object state. This difference state is then used to query the inverse model responsible for the target object. Since only the actuator can be influenced by action primitives directly, these local inverse models do not directly return action primitives. Instead they return preconditions in the form of the relative interaction features. These features represent a situation where the object state changes in the direction of the target configuration.

The current actuator state is then compared to these preconditions. If the actuator is already in a configuration where it meets these preconditions, an action primitive is derived from the local features. This requires that the local features contain information that can be used to compute an action primitive. However, in most situations the actuator configuration will not meet the preconditions. The given preconditions will often require the actuator to be in a different position relative to the object that is to be moved. In these cases the suitable position for the actuator is used as an intermediate target.

Considering the current object's position, an action is calculated to move the actuator

towards the intermediate target. It might be necessary to perform a circling movement around the object. Moving the actuator directly towards the intermediate target position might result in moving the object in a more unfavorable position and is therefore avoided. Since these steps are performed at every timestep, the model can quickly adapt itself to changes in the environment.

3.3.3. Theoretical discussion

Representing each object by itself has the advantage, that no transformations outside of the concept are required. Instead all objects are represented for themselves within the model. Consequently, any noise in any object does not directly influence another object's state directly. Noisy data in one object might still influence the prediction of another object, for example if the noise leads to an erroneous classification by the gating function. However, the effect is not as unavoidable as in the pairwise interaction concept.

The big disadvantage of this approach is that impossible configurations can be predicted more easily. This can happen due to an inaccurate prediction made by either the actuator's or the object's forward model. An impossible configuration is even bound to be predicted when the gating function misclassifies that no interaction takes place when it should. In this case the actuator is predicted to move into the other object.

Extending this concept to multiple objects is easier in theory since each object can be considered separately. However, training the gating functions with multiple objects is difficult. When one object's state changes during an update from the environment, the model needs to determine which other object caused this change in order to train the gating function on the correct relative interaction features. Depending on the situation, this causal determination can be very difficult and hardly be done without additional prior knowledge.

Apart from the representation, this concept requires an additional assumption about the scenario compared to the interaction state concept. The assumption that object states can only change through interactions with an actuator also means that an object does not continue sliding after it has been pushed. This restricts the possible weights of the objects and speeds of the actuator. A possible solution that was not pursuit within this thesis would be to predict the forces that operate on each object instead of the resulting object state. Using the forces, each object could train its own forward model which predicts the next state based on these forces. This however requires the robot to be able to detect or estimate these forces. Furthermore, predicting forces is often more complex then predicting static attributes such as position or orientation.

3.4. The Averaging Inverse Model

In order to compute suitable action primitives given a target configuration, both concepts require inverse models. Directly searching in the forward regression model for the inverse has two disadvantages in the given scenario:

1. The difference state that is computed can have features orders of magnitude greater than any change the local models have been trained with. In fact any target configuration, that requires multiple timesteps to reach, is already outside the range of the changes the forward models can have seen. The inverse model would need to extrapolate into unknown regions in these cases.

2. The features in the target state are not necessarily normalized. Therefore, the model does not necessarily know if a reduction of the difference in one feature is better than in another. This especially becomes a problem once only actions can be found that decrease the difference in one feature dimension at the cost of an increase in another feature dimension. Consider the example where a target position and orientation is given for an object. There might be situations where the object needs to be turned away from the target orientation in order to reduce the distance to the target position. In this case the orientation would need to be more or less ignored when using the inverse model to search for an action.

Furthermore, the output of the inverse model does not need to be as precise as the predictions of the forward model. This is mainly due to the fact that the inverse model is used to reach a target configuration interactively. In the context of this thesis, there is no need to provide a sequence of precise action primitives that will then be executed blindly. Instead, as highlighted in figure 3.1, the model is tightly coupled with the environment. This means that the model is constantly updated and can adapt to changing situations. It is therefore sufficient for the inverse model to provide the means to make a step in the direction of the target configuration. In the context of the two concepts presented here, the means are not directly action primitives but rather preconditions that need to be fulfilled in order to be able to influence an object in a certain way.

In order to avoid the mentioned problems, a special inverse model is proposed that is trained separately from the forward model: The model is called **Averaging Inverse Model (AIM)** due to its nature of computing average preconditions for certain effects. Unlike a forward model, it does not focus on the actual quantity of the feature changes within one timestep. Instead, the inverse model focuses mainly on the direction, or the sign, of the changes. Furthermore, in order to avoid the problem of weighting feature importance, each feature dimension is considered separately.

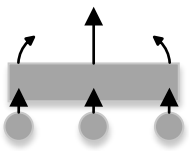

Figure 3.8.: Visualization of the precondition grouping in the proposed Averaging Inverse Model. The arrows above the block symbolize the change in the object's state corresponding to the actuator situations below the block. All three pushing scenarios indicated by the three spheres, will be grouped together since they all result in the same upward positional change.

For each feature dimension, the model collects all the preconditions that result in a positive and negative change in that dimension separately. Here preconditions means a representation of the situation in which the change occurred. An example of this grouping is visualized in figure 3.8.

The figure shows three different scenarios or preconditions in the form of the three spheres that represent the actuator. In the forward model each precondition is tightly coupled to its resulting change. In this example the left situation results in the small positional change upwards and a negative change in orientation. The same tight coupling is present with the other two situations. In the inverse model proposed here, all three situations are grouped together in one `prototype` responsible for a positional change upwards. Other prototypes exist for the two directions of change in orientation. This way one situation will be included in multiple prototypes if it has caused multiple features to change.

Each prototype creates a weighted average[1] of the preconditions for its feature dimension and sign. The preconditions are weighted by their contribution towards the feature dimension. In the example above, the preconditions for the middle scenario receive the strongest weight since they are responsible for the biggest change in position.

When the inverse model is queried given some difference vector to the target configuration a greedy strategy is used to reduce the feature dimension with the biggest difference first. The feature dimension along with the sign of this dimension in the difference vector determine the responsible prototype. The average computed by that prototype can then be used as preconditions that will reduce the distance to the target configuration.

[1] Depending on the used features, a simple average does not work. See the implementation details in section 4.4.1 for details.

Chapter 4

Model realization

The previous chapter introduced two concepts that were develop with the goal of adapting themselves to unknown situations through interaction. In order to evaluate the developed ideas, prototypes of the two concepts have been implemented in Python 2.7. Both prototypes are developed as such that they can be evaluated in the same way in the next chapter. The implementations follow the presented concepts very closely in most cases. As such the general data flow is the same as was presented in the figures 3.3, 3.4, 3.6 and 3.7. Relevant implementation details as well as made assumptions are highlighted and explained in this chapter.

Section 4.1 starts with describing common design decisions that apply to both concepts. Afterwards the relevant implementations details regarding the two prototypes are presented in section 4.2 for the interaction state concept and in section 4.3 for the object state concept. Finally, section 4.4 describes the used technologies, the underlying regression and classification model and the implementation of the inverse model, in detail.

4.1. Basics

4.1.1. Available information about objects

The kind of information that is provided by the environment obviously depends on the sensors that are available to the robot. From the information provided by these sensors, features can be computed. The models themselves are greatly independent of what features are actually used. In fact the models do not require any knowledge about what the features represent for prediction. This allows the exact same model to be used in various settings with different features, for example if additional or different sensors are available.

In order to yield good results, the used features obviously need to have the necessary expressiveness.

Unfortunately, the models do need to have some partial knowledge about the features when it comes to computing action primitives as will be further explained in the corresponding sections for both models.

These prototypes were evaluated using a physics simulation, further explained in section 5.1. From this simulation the information in table 4.1 is provided for each object at each update.

Feature	Description
x Position	Global x position of the object
y Position	Global y position of the object
Orientation	Global orientation around the z-axis of the object
x Velocity	Global x velocity of the object
y Velocity	Global y velocity of the object

Table 4.1.: Summary of all information about objects that is provided by the environment at each update step.

Furthermore, a unique identifier, the shape and size of the objects are known and can be used to compute additional features. What kind of features are computed from this information is a design decision by the user of these models. Since the underlying regression model is memory-based, using more features might result in worse performances. This is even more likely, if the features are not normalized or no suitable metric is used.

4.1.2. Action primitives

Action primitives are the collection of possible low level actions a robot can perform directly. Higher level actions are usually composed of multiple action primitives. For example, the action walking might be composed of the action primitives raise leg and lower leg.

In the scope of this thesis, the action primitives are defined by the simulation used for the evaluations in the next chapter. In this case a two dimensional action primitive is used, representing the x and y velocity of the actuator. The velocities are given according to the global coordinate system of the robot. In general, the nature of the action is not all that important to the models as long as it can be represented as a vector. However, the models assume that an action primitive is selected and performed at every iteration. This is also

true for the used velocity action primitives, even if the velocity does not change by using the same action primitive again.

The used simulation only accepts velocities with a norm not higher than $0.5\frac{m}{s}$. The implementations make sure to scale all computed action primitives accordingly. While moving towards a target, the implementations scale the action primitives to a norm of $0.3\frac{m}{s}$ in order to not push the objects too far.

4.1.3. Used regression and classification model

As already stated in the introduction and motivated in chapter 2, this thesis uses a memory-based approach to learn the interactions. For this end, a special adaptation of the GNG has been developed which can be used for regression as well as classification tasks. This adaptation, called Adapted Instantaneous Topological Map (AITM) and explained in section 4.4.2, is used for all regression and classification models used in both prototypes unless stated otherwise.

4.1.4. Using local features and predicting differences

Both prototypes only use local (differential) features as inputs for their trained regression and classification models. Local features mean that all features are represented relative to some reference object's coordinate frame. The exact computation of the features is explained in sections 4.2.1 for the interaction state model and 4.3.1 for the gating model. Using local features assumes that similar interactions behave identical regardless of where they take place in the environment. Only the local relations between the involved objects determine the outcome of the interaction. For the given task of pushing interactions this assumption will hold true as long as the environment only has constant effects on the objects. The environment used for the evaluations, described in section 5.1, fulfills these conditions since it only affects the objects through gravity and friction, which are constant in the given scenario.

The great advantage of this approach is that the interactions can be learned regardless of the object's actual configuration in the environment. In fact training data from all configurations can be used to learn a certain interaction. Furthermore, this approach provides free generalization to any configuration in the environment without the need of additional training data.

In more realistic environments, where this assumption does not hold true, the local features can still be used. However, in this case, additional features containing the information

about the influence of the environment or the current object configuration might be required. Depending on the actual scenario and its dynamics, this might require absolute global features which offer a lot less generalization.

An additional consequence of these local features is that the forward models of both concepts can only predict the changes instead of resulting features. As already stated in the two concept descriptions, the actual predictions for the interaction state and the object state are computed by adding these predicted changes to the current states. Without knowledge about the actual position in the input data, the actual position after an interaction cannot be predicted. However, this is not a problem but rather a benefit. Changes are limited in their size compared to the resulting features. Consider the feature position: An object cannot change its position from one timestep to another arbitrarily, but rather only by a certain maximum amount. When predicting changes, the learner's output only needs to cover the subspace defined by this maximum amount instead of the entire space. Furthermore, this approach of only predicting the changes can also be applied when using global input features. The models themselves are therefore not restricted by this.

4.2. Modeling pairwise interaction

The implementation of the pairwise interaction model contains the components visualized in the overview figure 3.2. The Abstract Collection Selector (ACS) trains a classifier that selects any of the n learned ACs. Within each ACs, there exist a local forward model which performs the predictions. However, instead of training local inverse models for each AC, one global inverse model is used in this implementation. The reasons for this decision are explained in section 4.2.3. Just as the local ones would, the global inverse model returns preconditions that result in a specific change of the interaction state. The forward model as well as the selector both use the same underlying mechanism in the Adapted Instantaneous Topological Map (AITM). For the reasons explained in section 3.4, the special Averaging Inverse Model (AIM) is used for the global inverse model.

As mentioned before, the model is constantly updated with new information provided by the environment. The steps that are performed to update the model each time new information is received are explained in algorithm 4.1.

The changed feature set S is computed according to equation 3.2[1]. The structure and used features within the interaction state are explained in section 4.2.1 while the episodes are explained in section 4.2.2.

[1]Since local features are used, both *Pre* and *Post* need to be transformed to the same coordinate frame, see section 4.2.2 for details.

Algorithm 4.1 Overview of the update steps in the pairwise interaction state model.

Input: New worldstate ws
Input: Used action *act*
Data: Last worldstate ws_{old}
Data: Set of Abstract Collections L
Data: Inverse Model IM
Data: Abstract Collection Selector ACS

```
1  for all interaction state i ∈ ws do
2      i_old ← extractInteractionState(ws_old, i)
3      newEpisode ← Episode(i_old, act, i)
4      S ← computeChangedFeatures(newEpisode)
5      if S ∉ L then                          ▷ Check if a corresponding AC already known
6          newAC ← AbstractCollection(S)
7          L ← L ∪ newAC
8      AC ← L_S                               ▷ The Abstract Collection responsible for S
9      update AC with newEpisode
10     update IM with newEpisode
11     update ACS with i_old, act and AC
```

When updating an AC its forward model needs to be updated. The forward model is trained using the combination of the old interaction state i_{old} and the action primitive *act* as input and the difference vector $\vec{d}$ between the old and the new interaction state as desired output. The exact training of the AITM is explained in section 4.4.2.

The global inverse model is trained on the same features as the forward model. Details regarding the training process are explained in section 4.4.1.

The ACS is also trained on the same combination of i_{old} and *act* as input, but it uses the identifier of the responsible AC as desired output. Since the selector also uses the AITM as classifier, the exact update rules are explained below.

When used for prediction the model follows the process visualized in figure 3.3 and described in section 3.2.2. Computing suitable action primitives on the other hand is more complicated and requires additional knowledge about the features. While the general process follows what is explained in 3.2.3, the process of providing relevant interaction states as targets and extracting relevant action primitives from the returned preconditions is explained in section 4.2.3.

Feature	Description
Id 1	Identifier of the reference object
Id 2	Identifier of the second object
Local x Position	Local x position of the reference object
Local y Position	Local y position of the reference object
Local Orientation	Local orientation of the reference object
Relative x Position	Relative x position of the second object
Relative y Position	Relative y position of the second object
Relative Orientation	Relative orientation of the second object

Table 4.2.: Features used to represent one interaction state. Relative positions and velocities refer to the coordinate system of the reference object.

4.2.1. Used features

The pairwise interaction model basically only uses the interaction state and the action primitive vector as feature vectors. The actual composition of the interaction state depends on the objects that need to be represented. For this thesis, only simple objects (e.g. spheres or rectangular cubes) are present in the scene which allow fairly simple interaction states as described in table 4.2.

As already mentioned in section 4.1.4 differential features are used. Apart from the features listed in the table, the interaction state contains the information required for the transformations. Specifically, the matrix T used to transform from the local coordinate frame back to the global frame, its inverse T^{-1} and the orientation of the reference object in the global coordinate frame are stored.

The interaction states are computed as follows: All features are computed by transforming the global features of both objects given by the environment to the local coordinate frame. Consider two objects o_1 and o_2 with positions $\vec{p}_1$ and $\vec{p}_2$ and orientations α_1 and α_2 respectively. First the transformation matrices T and T^{-1} are computed from the reference object's position and orientation:

$$T = \begin{pmatrix} \cos(\alpha_1) & -\sin(\alpha_1) & p_{x1} \\ \sin(\alpha_1) & \cos(\alpha_1) & p_{y1} \\ 0.0 & 0.0 & 1.0 \end{pmatrix} \qquad T^{-1} = inv(T) \tag{4.1}$$

p_{x1} and p_{y1} correspond to the x and y dimensions of the position $\vec{p}_1$. With the help of the transformation matrix T^{-1} it is possible to compute the local positions, e.g.:

$$\vec{p}' = T^{-1} \times \vec{p}_1^* \tag{4.2}$$

where $\vec{p}_1^*$ is the homogeneous vector of $\vec{p}_1$. Since $\vec{p}'$ is also in homogeneous coordinates, only the first two components are used for the interaction state. The local and relative orientations are computed by subtracting the orientation of the reference object from the given orientations. In fact by doing this, all local fields in the interaction state will be zero after the computation. However, these fields are still required. The ACs predict the change in the current interaction state. In order to extract the predicted object states from the interaction state, changes in the reference object need to be predicted as well.

The object states are extracted by using the inverse transformation. Since the structure of the interaction states are known outside of the model, the predicted local position $\vec{q}$ can easily be extracted. Using the homogeneous coordinates $\vec{q}^*$ of $\vec{q}$, the prediction for the actual object's position $\vec{p}_{pred}$ can be computed:

$$\vec{p}_{pred}^* = T \times \vec{q}^* \tag{4.3}$$

Velocities can be computed analogously if needed. In order to get the global orientation, the reference object's orientation simply needs to be added to the predicted orientation.

The model assumes, that these interaction states come directly from the environment. Transformations from and to the actual object states need to be performed outside of the actual model. This is achieved by introducing a **Worldstate**. From the point of view of the model, this worldstate is simply a collection of all interaction states in the environment. At each update from the environment, a new worldstate is computed from the provided information. The model predicts a new worldstate by making predictions about all interaction states that are included in a given worldstate. Since one is usually more interested in the actual object states, the model finalizes the worldstate after all interaction states have been predicted. This finalization extracts the object state predictions from the predicted interaction states.

4.2.2. Episodes

Episodes are used to store past experiences. The idea comes from case-based-reasoning [39] where past experiences are used to reason about new problems. An episode is made up of three parts:

1. The initial interaction state Pre that was given before an action was performed.
2. The action that has been performed.
3. The resulting interaction state $Post$.

As explained in the concept, these experiences can be stored and looked up at query time in order to make predictions. However, as highlighted before, the performance of such an operation deteriorates over time as the number of stored experiences increases. Instead, the ACs use these episodes as training data for the local regression model. Apart from those three parts, all episodes compute a difference vector between the initial and resulting interaction state. However, since the model is working with local coordinates, it is important that both states are transferred to the same coordinate frame before computing the difference. As explained above, the features in all interaction states are always computed relative to the reference object's coordinate frame. As such the positional information of the reference object will always be zero in a freshly computed interaction state. However, this does not mean, that the reference object has not moved within an episode. The model is interested in learning to predict how a given interaction state changes after a special action is performed. Therefore, the difference vector in an episode needs to represent the differences relative to the initial interaction state. This is achieved by first transforming the features of the $Post$ state back to global coordinates via T_{Post} and then transforming them to the local coordinate frame of the Pre state using T_{Pre}^{-1}. For the position of the reference object p_{ref} (in homogeneous coordinates) this can be computed as follows:

$$\vec{p}'_{ref} = T_{Pre}^{-1} \times T_{Post} \times \vec{p}_{ref} \tag{4.4}$$

When transforming the orientation of the $Post$ state, first the global orientation of $Post$'s reference object is added before subtracting Pre's global orientation. All transformed features from $Post$ are again collected in a vector $Post^*$ where the features are organized in the same way as in a normal interaction state.

Afterwards the difference vector can simply be computed by:

$$\vec{d} = Post^* - Pre \tag{4.5}$$

During training, the ACs extract the feature differences that they are responsible for from $\vec{d}$ and use those as target output of the local regression model. Finally, the set of changing features S can then again be computed as stated in equation 3.2 while substituting $Post$ with $Post^*$.

4.2.3. From target object state to action primitives

In most cases the robot will have to push a certain object to a given specification. It usually does not matter where the actuator is after the target is reached. Therefore, it would be best if a target object state could be provided instead of a target interaction state. However, the model itself does not know about object states. Therefore, this prototype implements a method to convert a given object state to an interaction state, where only the reference object is correctly set. The actuator is used as secondary object, however the actuator's features are not required. Instead, the features representing the reference object are remembered in order to only focus on these features when trying to reduce the distance to the target interaction state.

Once a target interaction state has been computed, the current interaction state between the reference object of interest and the actuator can be retrieved. Using the target interaction state as resulting state, an episode with an empty action is computed. The episode provides the local difference vector[2] between the current situation and the target configuration. Only the features remembered when constructing the target interaction states are considered from this difference vector. Effectively, these features correspond to the local differences in the object states.

Following the concept provided in section 3.2.3, this reduced difference vector would be used to find the AC that is responsible for changes represented in the vector. Having said that, since only changes of a subset of the entire interaction state are considered, there will usually not be an AC that is responsible exactly for these changes. Instead there will be several ACs, that are responsible for changes in these features. In theory, the AC responsible for the most feature changes should be chosen in this situation.

Unfortunately, the developed inverse model does not work well with the segmentation performed by the ACs. The main reason for that is, that the inverse model trains separate prototypes for each feature dimension. Each prototype tries to learn the preconditions that have the most influence on the given feature dimension. However, a single feature dimension is often represented in multiple ACs. Therefore, each local inverse model would only experience a subset of the preconditions responsible for changes in a specific feature dimension. When the AIM is trained only on these local parts, the prototypes do not learn the preconditions well.

On top of that, the AIM is a model with close to constant update and query times[3]. Therefore, the main reasons for training multiple local models do not apply to this inverse model.

[2]Here local means that the difference vector is computed in the current interaction state's coordinate frame.

[3]The number of prototypes and averages that are learned per prototype are independent of the number of training examples. See section 4.4.1 for details.

It is for these reasons, that this implementation deviates from the concept in this point and only trains a single global inverse model. This global inverse model is queried just as a local one would be and returns suitable preconditions given a difference vector.

Once preconditions have been retrieved from the inverse model, the current situation is compared to these preconditions. Most importantly, the current actuator position needs to be similar to the actuator position in the preconditions. If the distance between these two positions is too great, the actuator is circled around the object. Unfortunately, this requires some world knowledge about the objects. In the given implementation, the interaction states defined by the user provide the ability to compute an action primitive that lets the actuator circle around the reference object at a fixed distance (see Appendix A.1 for details). Using this circling action the actuator is able to move towards the desired position without influencing the object.

Once the distance is below a threshold of 0.1m[4], the model assumes that the actuator is on the correct side of the object and can move directly towards the desired position. This can be done by simply following the direction between the target position and the current position.

As soon as the actuator has come close[5] enough to the position defined by the preconditions, the actual desired action primitive can be computed. The preconditions already contain local action primitives that were encountered during training. These can be transformed to the global coordinate frame using the transformation matrix T from the current interaction state as described above.

In case no preconditions could be retrieved, for example because the inverse model has not seen training examples that produces changes in the required directions, the model would need to explore. Real strategic exploration is not developed in this thesis, which is why this implementation uses a random action primitive in the general direction of the object in this case. This can lead to the model getting stuck which is why for the purpose of this thesis both models assume, that all required changes have been experienced before a target should be reached.

4.3. Object space with gating function

The realization of the second prototype consists mainly of the parts visualized in figure 3.5. Depending on whether the actuator models are to be learned as well, three to five different parts need to be trained in this model: The predictor occupies a similar role to the ACs

[4]The threshold heavily depends on the given environment and the size of the objects. This threshold has been empirically determined and worked well in the evaluation environment described in section 5.1.

[5]This implementation uses the threshold of 0.01 to determine when it is close enough.

in the previous model. For each distinct object group, a local forward model is trained using the relative interaction features (described in section 4.3.1) as input and the changes in the object states as output. Just as the interaction model, the inverse model is also trained on the same input and output data each time new information is available from the environment. In this implementation, object groups are simply divided by the object identifier. The gating function trains a classifier in order to predict if an actuator object influences another object. The classifier is trained, using the relative interaction features as input. The output, i.e. if an interaction took place or not, is determined by computing the change between the previous object state and the current one. If no actuator models are predetermined, both the local forward and inverse model for the actuator need to be learned as well.

Algorithm 4.2 summarizes all steps performed at each update step.

Algorithm 4.2 Summary of the steps performed by the gating model at each update from the environment.

Input: New worldstate ws
Input: Used action *act*
Data: Last worldstate ws_{old}
Data: Predictor
Data: Gate

```
1  newActuator ← extractActuator(ws)
2  updateActuator(newActuator, act)
3  for all object state o ∈ ws do
4      o_old ← extractObject(ws_old, o)
5      relFeatures ← computeRelativeFeatures(o_old, newActuator)
6      change ← computeLocalChange(o_old, o)
7      hasChanged ← ||change|| > ε_change
8      updateGate(relFeatures, hasChanged)
9      if hasChanged then
10         updatePredictor(relFeatures, change)
```

At the beginning of each update step, the current and last states for the actuator as well as the objects is retrieved from the current and last worldstate respectively. Extracting the actuator or an object state is simply an attribute lookup in the worldstate.

The actuator is updated first. If no local forward model is predefined, an AITM is updated with the used action primitive as input and the change in the actuator's state as output.

Afterwards each object is considered separately. A difference vector change is computed between the old object's state and the new one. Since the local models want to predict local

changes, the change in an object's state needs to be computed with respect to the coordinate frame of the object before the update. This requires the features to be transformed as described in section 4.3.1.

Furthermore, the relative interaction features between the old object state and the new actuator state need to be computed. The new actuator state is used because the model uses the predicted actuator state when making predictions instead of the current one. Therefore, the model needs to train using the actuator state that results from the last action primitive, instead of the old actuator state.

The computed change vector is used to determine if the current object has actually been influenced by the actuator by comparing the norm of the change vector with ϵ_{change}. This threshold should be set according to the noise level in the received information. This implementation uses $\epsilon_{change} = 0$.

When the object state has changed, the predictor is updated with the relative interaction features as input and the change vector as output. In this implementation the predictor determines the object group by the identifier in the relative features in order to train only the local forward and inverse models responsible for the current object group. Ideally, the model would be able to automatically determine object groups through interaction and comparing their behavior.

When used to predict the next world state, the process visualized in figure 3.6 and described in section 3.3.1 is used for every object in the current worldstate. As mentioned in the concept, prediction is a sequential process in this model:
First the next actuator state is predicted by querying its local forward model given the selected action primitive. Afterwards, the predicted actuator state is used to compute the relative interaction features with the current object. This feature vector is then used to query both the gating function and if required the object's local forward model in the predictor. As a consequence of this sequential process, the local models in the predictor do not know about the action primitives at all. This has the benefit that the same process and the same representations can be used to make predictions about object-object interactions, provided the problem with training the gating function that was mentioned in section 3.3.3 is solved.

As in the other model, computing action primitives is more complicated and requires additional knowledge about the features. The steps required to extract action primitives useful to reach a given target are explained in section 4.3.2

Feature	Description
Id	Unique identifier of each object
x Position	Global x position in the environment
y Position	Global y position in the environment
Orientation	Object rotation around the z-axis of the global coordinate system

Table 4.3.: The different dynamic features used to represent objects in the gating model.

4.3.1. Used features

Similar to the interaction model, this model also introduces a Worldstate. This worldstate is computed at each update from the environment. In this case, the worldstate simply collects the states of all objects in the environment. The actuator, although technically an object, is regarded separately in the worldstate. Since the model directly predicts the new object states, no finalization is required. The object states describe the specific features of each object and basically represent what is provided by the environment. This implementation can use dynamic features such as velocities, but does not require them. Table 4.3 summarizes the basic features that are used to represent object states.

While additional information, such as information about the shape of the objects, is available, the model itself does not require it. However, when computing the relative interaction features, at least information about the shape is required. Furthermore, the model stores not only the current state of each object but also the previous one. This is required in order to make finite difference estimations about the objects dynamics such as velocity. In case the dynamics are directly provided by the environment, this previous state does not need to be stored. However, omitting the velocities and only estimating them when needed reduces the output dimensionality of the regression models. This is because the model predicts changes in all features that it experiences. Since the model does not have or require any knowledge about the features for prediction, a selective prediction is not possible. Therefore, it is beneficial to reduce the number of features that need to be predicted.

The different regression and classification models are trained using relative interaction features as input. As mentioned in the concept description, these features are similar to the interaction state described in the pairwise interaction model above. However, not all information about both objects needs to be expressed. This is because, each object's information is available separately. Furthermore, as mentioned before, the model assumes that the global configuration of an object does not influence an interaction. Therefore it is sufficient to model the actuator relative to the reference object. Additional information

Feature	Description
Id 1	Identifier of the reference object
Id 2	Identifier of the second object
Distance	Closest distance between the two objects
Closing	Describes how much the objects are moving towards each other
Relative x Position	Relative x position of the second object
Relative y Position	Relative y position of the second object
Relative x Velocity	Relative x velocity of the second object
Relative y Velocity	Relative y velocity of the second object

Table 4.4.: The different relative interaction features used to make predictions about interactions in the gating model. Relative positions and velocities refer to the coordinate system of the reference object.

that might be useful for the learner is also provided. The relative interaction features are summarized in table 4.4.

The Closing feature c is visualized in figure 4.1. Its computation is given by equation 4.6.

$$c = \vec{n} \cdot \overrightarrow{rv} \tag{4.6}$$

$\vec{n}$ represents the normal vector from the reference object towards the second object and $\overrightarrow{rv}$ represents the non normalized relative velocity vector of that second object. The dot product equates to the cosine between these two vectors weighted by the magnitude of the relative velocity. This feature is minimal when the objects are moving directly towards one another. A positive closing value on the other hand indicates that the objects are moving away from each other. When the feature becomes 0 it indicates that the distance will not change. As mentioned above, the relative velocity is estimated by the finite difference of the current and last position.

The other relative features are computed by transforming their global counterparts to the coordinate system of the reference object. Equation 4.7 shows this exemplary for the position:

$$\overrightarrow{relPos} = T^{-1} \times \overrightarrow{gPos} \tag{4.7}$$

where T^{-1} is the transformation matrix computed from the reference objects orientation and global position as defined by equation 4.1. $\overrightarrow{gPos}$ is the position vector of the second

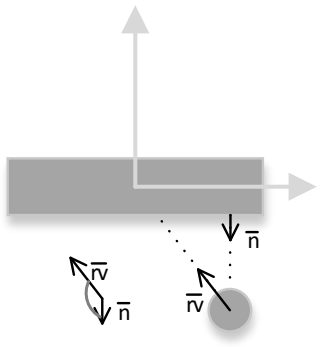

Figure 4.1.: Visualization of the closing feature. The gray half circle shows the angle whose weighted cosine is computed by the feature.

object in homogeneous coordinates. The same transformation is required when computing the change in an object's state using the old object's state as reference.

The distance is calculated by computing the distances from all corners of one objects to all edges of the second object. The distance between the two objects corresponds to the closest of these corner-edge distances. In case of round objects, such as the actuator, the center is considered as a corner and the radius is subtracted from the computed distance.

4.3.2. From target object state to action primitive

The overall process of computing a suitable action primitive that allows to push an object towards a given target configuration is quite similar to the one in the previous model that is explained in section 4.2.3. However, due to the differences in representation, the process is actually easier in this model. First of all, since the model works directly with the object states, the target representation does not need to be changed. A specified target object state can directly be used by the model. Furthermore, for each object group only one inverse model exist, which directly avoids the problems with multiple local inverse models mentioned in section 4.2.3.

The preconditions are computed as already visualized in figure 3.7:

First the local difference vector[6] between the current object state and the target state is calculated. Afterwards, the local inverse model responsible for the given object is queried for the preconditions given this difference vector. The actual computation of these preconditions is explained in section 4.4.1.

Once the preconditions have been computed, the same analysis as in the other model needs to be performed. These preconditions are in the form of the relative interaction features described above. Unlike in the prediction case, the model needs to have knowledge about what some of the features represent. Specifically, the information about the local actuator position $\vec{p}_{cond}$ needs to be extracted from these preconditions. Afterwards, $\vec{p}_{cond}$ can be compared with the local actuator position $\vec{p}_{cur}$. $\vec{p}_{cur}$ is extracted from the current relative interaction features between the object and the actuator. If these two positions are too far apart[7] a circling action is performed. Unfortunately, this needs to be provided by the user since the model has no information about the shape of the objects or what circling means. In this implementation, each object state, that needs to be defined by the user anyways, provides a method that computes an action that circles the actuator around the object in a fixed distance. The algorithm used to compute the circling action is described in Appendix A.1.

The model uses a circling action instead of directly moving towards the desired position, because the target position might be on another side of the object. When moving the actuator straight towards $\vec{p}_{cond}$ the object might be influenced in a negative way.

If the actuator is close but not close enough[8] the direction $\vec{d} = \vec{p}_{cond} - \vec{p}_{cur}$ can be used as action primitive. However, regardless of how fine the "closeness" threshold is set, the object might still be in the way of this direction in some cases. In order to avoid, moving the object in an unintended way, the direction is only followed if the gate predicts that the next action will not influence the object. Otherwise, the same circling movement as described above is performed.

Once the actuator is close enough to $\vec{p}_{cond}$, the actual action primitive needs to be extracted from the preconditions. While the action primitives were directly included in the preconditions in the interaction state model, they are not included in the preconditions for this model. However, the relative interaction features include the relative velocity of the actuator. Since the action primitives control velocities in this scenario, the direction of this relative velocity can be used to construct an action primitive that points in the same direction. Afterwards, the action primitive needs to be transformed to the global coordinate frame, using the transformation matrix T as explained above.

[6] Local mean here that the difference vector is computed with respect to the current object's coordinate frame.

[7] This implementation considers an euclidean distance of 0.1 as too far apart. This threshold has proven to be suitable in the environment presented in section 5.1.

[8] This implementation considers a euclidean distance of < 0.01 as close enough.

As mentioned in the interaction state model above (section 4.2.3), in case no precondition could be returned by the AIM, exploration would be required. However since this was not a focus of this thesis only a random action in the general direction of the object is used which can lead to the model getting stuck. Therefore, it is assumed that the inverse model as been trained sufficiently before a target configuration is to be reached.

4.4. Used technologies

4.4.1. The Averaging Inverse Model

The Averaging Inverse Model (AIM) was suggested in the previous chapter in order to deal with the problems involved in computing the input from a given output. In this setting, the problem of extrapolation is the most important one since the models are supposed to reach any target instead of only ones that are reachable by performing a single action primitive.

This special inverse model is designed to learn typical preconditions that result in a certain change in the features. It is less important how strong the change is, but rather in what direction (positive or negative) a feature dimension changes. The idea is to compute the average preconditions for each feature dimension and each direction this dimension changes in. This assumes, that similar situations result in similar changes, for example if a block is pushed from the left it moves to the right.

These preconditions depend on the actual model. In case of the interaction state model, the combination of interaction state and action primitive is used while the gating model uses the relative interaction features. However, as described in sections 4.2.3 and 4.3.2, this average can be used to derive an action primitive in both cases.

The implementation of this inverse model consists of two parts:

- A network
- Its nodes (prototypes)

Network:

The network is the main part of the inverse model. It contains and manages the different prototypes. As explained in the sections for both models, the inverse model is trained on the preconditions and the difference vector. The difference vector $\vec{d}$ represents either the changes in an interaction or an object state depending on the used model.

For each actually changing feature dimension within $\vec{d}$ the network creates up to two prototypes. One for positive and one for negative changes within that dimension. That

way a prototype represents changes in a given direction (positive or negative) of one specific feature dimension.

In the example in figure 3.8 three different pushing interactions between the actuator and an object are seen. When the network is trained on these situations, three prototypes are created: One for the position change upwards and one for each rotation direction. Since all three situations result in an upward positional change, the corresponding prototype averages the preconditions of all three situations. On the other hand, the two orientation prototypes will only be trained on one situation each.

The network is queried to provide preconditions that reduce a given difference vector. Ideally, preconditions that reduce the difference in all features at the same time would be returned. However, averaging the preconditions returned by different prototypes is prone to produce invalid preconditions. Furthermore, depending on the environment, it might not always be possible to reduce all features at the same time.

In order to avoid invalid preconditions, the network uses a greedy strategy of reducing the feature dimension with the biggest difference[9]. Since the feature dimensions are not normalized, the biggest numerical value might not correlate to what humans regard as a big distance. However, this approach works quite well in practice, considering the inverse model does not have any knowledge about the feature's characteristics.

Once the feature dimension with the biggest difference has been chosen, the corresponding prototype returns up to two possible preconditions as explained below. The prototypes are asked to return two preconditions where possible in order to allow greater flexibility of the network:
The network tries to determine preconditions that reduce the difference in the biggest feature dimension while also setting up future movements in order to reduce the next biggest difference. Consider the example in figure 4.2 where two valid alternative preconditions are visualized: If the block should not only be rotated counterclockwise but also moved upwards, the position below the block is more favorable. This is because, in order to reduce the positional difference, the preconditions averaged over the situations visualized in figure 3.8 would be used. On the other hand, if the block should be moved to the right, the other situation should be preferred.

In order to do that the network queries the preconditions for the second biggest feature dimension as well. By comparing the (euclidean) norms between the four preconditions it is possible to determine which of the two preconditions for the first feature dimension is closer to the preconditions of the second feature dimension.

[9]In order to avoid oscillations between different features and preconditions, the network remembers the selected feature until the sign in the difference flips or the feature difference $||f|| < \epsilon_{Index}$. Again this threshold depends on the expected change of features in the environment.

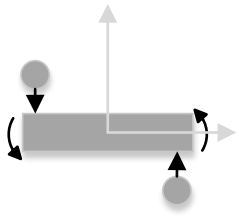

Figure 4.2.: Visualization of two alternative preconditions for a positive rotation of the block. The circles represent the actuator while the rectangle represents a block object. The gray arrows symbolize the local coordinate system of the block. Averaging these preconditions in the prototype would result in an invalid precondition.

In case both alternatives for the first feature are deemed equally good, the first preconditions are used since these correspond to a greater change in the desired feature dimension.

Unfortunately, this strategy, does not always avoid choosing the suboptimal preconditions. Since the preconditions for the second biggest feature dimension can only ever be evaluated based on the current situation, wrong choices might be made. Consider the same example as above, but the block is supposed to be turned by 180°. In that case, the local positional difference, the inverse model tries to reduce, will change its sign by turning the block around. This means that in order to move the block upwards in the global sense after turning it around, the block needs to be pushed downwards regarding its local coordinate system. In this case, the alternative above the block in figure 4.2 would have been the better choice.

If the network is queried before it has been trained sufficiently, there might not be a prototype for the feature dimension with the biggest difference. In that case the network tries to reduce the second biggest feature dimension and so forth. If no prototype can be found for the remaining differences, the network cannot return any preconditions which it indicates by returning a null value.

Prototype:

Within a prototype, all collected preconditions are averaged by weighting each input by the magnitude of the change in the feature dimension in $\vec{d}$ that the input caused. In the example in figure 3.8, the middle scenario receives the biggest weight, since it changes the position more then the outer situations.

While this average works great in the given example for the positional node, it does not work for all features as can be seen in figure 4.2.

Feature	Combination 1	Combination 2
x position	+	-
y position	-	+
x velocity	0	0
y velocity	+	-

Table 4.5.: The sign combinations of preconditions in the example in figure 4.2.

Consider the orientation prototype for positive rotations around the z-axis as shown in the figure. The magnitude of the change is identical in both situations, meaning that normally, the mean of both preconditions should be used as the average. In the given situation, this results in invalid preconditions, where the actuator would be supposed to be in the center of the object. That position is not only not reachable, but it also would not produce the desired change in orientation.

In order to prevent this, multiple averages are computed in a prototype. Basically, the prototypes store a separate average for each combination of signs in the input data. This model uses three different sign possibilities: `+`, `-` and `0`[10]. Going back to the example in figure 4.2, assume that the following four input features are used: relative x and y position and relative x and y velocities of the actuator. All feature are represented in the object's coordinate frame. Table 4.5 shows the sign combinations that are experienced in the two given situations.

In this case, averages for both possible combinations are updated in the prototype. More specifically, for each different combination the prototype stores the sum of the contributions, the sum of all inputs as well as the number of times this combination was experienced.

Unfortunately, noisy data will lead to the creation of multiple combinations that are basically equivalent. For example it might be, that due to some motor or sensor noise, a small x velocity $> \epsilon_{Node}$ was detected in one of the two situations. In that case a third combination with an x velocity sign different from `0` is created as well. As long as this velocity is very small, this combination would still be valid. In the worst case noise can create invalid combinations. An example would be if the sensors pick up a change in the object's orientation although the actuator moved away from the object. Since the model cannot know what is a valid training example and what is invalid noise, all combinations are created equally.

[10]The sign is considered 0 if the feature $||f|| < \epsilon_{Node}$. This threshold depends on the expected change of the features.

Feature	Combination 1	Combination 2	Combination 3
x position	+	0	-
y position	-	-	-
x velocity	0	0	0
y velocity	+	+	+

Table 4.6.: The three sign combinations for the example in figure 3.8 where the block is pushed at three different positions from below.

When the prototype is asked to return the averaged preconditions, it tries to find up to two valid combinations for the network. These two combinations would ideally represent two valid but different alternatives, as in the example above. In order to create these averages, the following steps are performed:

1. Merge equivalent (valid) combinations.
2. Select up to two merged combinations with the highest contribution.
3. Compute averages from these two combinations.

1) Merging:

The prototypes try to merge equivalent combinations. This is done, by making a special assumption about features with a `0` sign: If an input feature has been experienced to be close to `0` for a given prototype, then it should be possible to average over all occurrences of this feature. In this sense, combinations are considered equivalent and are merged, if they do not have any opposing signs at any of their input features. Table 4.6 summarizes the sign combinations experienced from the three scenarios in the first example, presented in figure 3.8. Since a `0` does not oppose either a `+` or a `-`, all three combinations can be merged together. Depending on the used features and the amount of noise in the data, special care needs to be given when merging combinations. In the example above, the resulting combination should ideally represent a x position close to 0 since that would result in the biggest positional change. In case the weights of either combination 1 or combination 3 is significantly bigger than the other, the average over all three situations would result in an x position different from 0. In order to further avoid such undesired effects, the merging process can be made more strict by only combining combinations that are similar even in their `0` signs. This implementation allows only one feature dimension to differ after the opposing sign check in order to better preserve `0` sign feature dimensions.

The actual merging process is described in algorithm 4.3. The sign combination of a given precondition is used as `key` in order to reference the corresponding weights, the sum of preconditions and the number of occurrences of that sign combination. In order to avoid

merging invalid combinations, the combinations are first filtered. This model assumes, that valid combinations are experienced more often than invalid ones that are results from noise. Especially, since small noise can also be filtered out by only considering feature changes above a certain threshold when training the prototypes. That way, all combinations that have been experienced less often then the average number of experiences over all combinations are filtered out. The resulting combinations are checked for their equivalence by iteratively comparing the combination with the most `0`s to all other combinations until all have been processed.

Algorithm 4.3 Description of the merging process for combinations within a node.

Input: List of combinations L
Input: Hashmap of combination weights W
Input: Hashmap of combination input sums I
Input: Hashmap of combination occurence numbers N
Output: Hashmap of merged combination weights W*
Output: Hashmap of merged combination input sums I*
Output: Hashmap of merged combination occurrence numbers N*

```
1  usefulKeys ← filterInvalidCombinations(L, N)
2  sortedKeys ← sortByNumZeros(usefulKeys)
3  while sortedKeys not empty do
4      currentKey ← sortedKeys[0]
5      tmpWeight ← 0
6      tmpSum ← 0
7      tmpNumber ← 0
8      for all key in sortedKeys do
9          if isEquivalent(currentKey, key) then
10             tmpWeight ← tmpWeight + W[key]
11             tmpSum ← tmpSum + I[key]
12             tmpNumber ← tmpNumber + N[key]
13             remove key from sortedKeys
14     newKey ← recomputeCombination(tmpSum, tmpWeight)
15     W*[newKey] ← tmpWeight
16     I*[newKey] ← tmpSum
17     N*[newKey] ← tmpNumber
```

After equivalent combinations have been merged, the resulting input features might represent a new sign combination. This combination is computed and used as the new key for storing the resulting values.

2) Selecting combinations with the highest contribution:

If only one combination remained after the merging process, the preconditions from that combination are returned. Otherwise, the prototype sorts the resulting combinations based on their average contribution. This can easily be computed by dividing the total weight by the number of experiences for each combination. In that case the best two combinations are considered if their average contribution is not too far apart[11]. Otherwise, only the best combination is used as if no alternative exist.

3) Compute preconditions:

From the two combinations with the biggest contribution, two preconditions are computed as described in algorithm 4.4.

Algorithm 4.4 Description of the averaging process within the nodes if at least two combinations are present.

Input: $\overrightarrow{pre}_1$ Input sum of the first combination
Input: $\overrightarrow{pre}_2$ Precondition sum of the second combination
Input: w_1 Total weight for the first combination
Input: w_2 Total weight for the second combination
Input: $\overrightarrow{comp}_1$ Vector containing feature signs for the first combination
Input: $\overrightarrow{comp}_2$ Vector containing feature signs for the second combination
Output: $\overrightarrow{average}_1$
Output: $\overrightarrow{average}_2$

```
1  size ← len(comp⃗_1)
2  for i = 1..size do
3      c_1i ← comp⃗_1[i]
4      c_2i ← comp⃗_2[i]
5      if c_1i == c_2i then
6          average⃗_1[i] ← (pre_1i + pre_2i) / (w_1 + w_2)
7          average⃗_2[i] ← average⃗_1[i]
8      else
9          average⃗_1[i] ← pre_1i / w_1
10         average⃗_2[i] ← pre_2i / w_2
```

The signs of both combinations are compared and the feature dimensions in the preconditions of both combinations are averaged if their signs are identical. The feature dimensions are considered separately otherwise.

Using these two components of the network and its prototypes, it is possible to abstract from the actual quantities in feature changes during training and instead focus in the

[11] In this implementation this means, that the second highest average contribution is not less then half of the highest average contribution.

direction. For the purpose of incrementally reaching target configurations this direction is sufficient in combination with the circling strategies explained in sections 4.2.3 and 4.3.2.

4.4.2. Adapted Instantaneous Topological Map

The underlying regression and classification model that is used throughout this thesis is an adaptation of the Instantaneous Topological Map (ITM) and is therefore called Adapted Instantaneous Topological Map (AITM) throughout this thesis. The Instantaneous Topological Map (ITM) [42] itself is an adaptation of the GNG [43] algorithm to create topological maps. Instead of GNGs global update rules for inserting new nodes in the map, the ITM uses local update rules in order to be better suited for correlated inputs. In order to be applicable to classification and regression, the ITM was further extended by an output function using the idea of LLM [17]. In order to extend a topological map with an output function, each node represent the corresponding output vector $\vec{w}^i_{out}$ along its input vector $\vec{w}^i_{in}$. The LLM extends each node further with a local linear mapping A^i. This matrix is used to improve the function approximation within each Voronoi cell. With this, the output of each node given an input vector $\vec{x}$ is computed by equation 4.8:

$$\vec{y}^i(\vec{x}) = \vec{w}^i_{out} + A^i \cdot (\vec{x} - \vec{w}^i_{in}) \tag{4.8}$$

The output function for the network can be computed in multiple ways. The simplest method is to use the output of the winning node, i.e. the output of the node whose input vector $\vec{w}^i_{in}$ is closest to the given input $\vec{x}$. In order to reduce the effect of the metric problem when finding the closest node, the outputs of multiple nodes can also be mixed together. The evaluations in this thesis interpolate the output functions of the two closest nodes:

$$\vec{y}_{net}(\vec{x}) = \frac{1}{k_n + k_s} \cdot [k_n \cdot (\vec{w}^n_{out} + A^n \cdot (\vec{x} - \vec{w}^n_{in})) + k_s \cdot (\vec{w}^s_{out} + A^s \cdot (\vec{x} - \vec{w}^s_{in}))] \tag{4.9}$$

where k_n and k_s are the weights or importance for the nearest and the second node respectively. These weights are computed as follows:

$$\begin{aligned} k_n &= \exp\left(\frac{||\vec{x} - \vec{w}^n_{in}||}{\sigma^2}\right) \\ k_s &= \exp\left(\frac{||\vec{x} - \vec{w}^s_{in}||}{\sigma^2}\right) \end{aligned} \tag{4.10}$$

σ determines the influence radius of each node, just like in radial basis networks [44]. The nearest node n and the second closest node s are determined by comparing the input vectors of all nodes in W with the given input vector $\vec{x}$:

$$\begin{aligned} nearest : n &= \underset{c \in W}{\operatorname{argmin}} ||(\vec{x} - \vec{w}_{in}^{c})|| \\ second : s &= \underset{c \in W \setminus \{n\}}{\operatorname{argmin}} ||(\vec{x} - \vec{w}_{in}^{c})|| \end{aligned} \tag{4.11}$$

During training, the network receives an input-output pair and updates its nodes. First, the two closest nodes *nearest* and *second* are computed as stated in equation 4.11. Afterwards, only the node *nearest* is adapted in accordance to the update rules of the LLM:

$$\begin{aligned} \Delta \vec{w}_{in}^{n} &= \eta_{in} \cdot (\vec{x}^{\alpha} - \vec{w}_{in}^{n}) \\ \Delta \vec{w}_{out}^{n} &= \eta_{out} \cdot (\vec{y}^{\alpha} - \vec{y}^{n}(\vec{x}^{\alpha})) + A^{n} \cdot \delta \vec{w}_{in}^{n} \\ \Delta A^{n} &= \eta_{A} \cdot (\vec{y}^{\alpha} - \vec{y}^{n}(\vec{x}^{\alpha})) \frac{(\vec{x}^{\alpha} - \vec{w}_{in}^{n})^{t}}{||\vec{x}^{\alpha} - \vec{w}_{in}^{n}||^{2}} \end{aligned} \tag{4.12}$$

The initial matrix A is a zero matrix with proper dimensions[12]. The learning rates η_{in}, η_{out} and η_A are meta parameters that need to be determined. In case η_A is set to 0, no linear approximation is learned for each Voronoi cell. This means, that each cell has only the constant output of $\vec{w}_{out}^{n}$. This thesis uses a $\eta_A = 0$ because the differences between $\vec{x}^{\alpha}$ and $\vec{w}_{in}^{n}$ are often very small, which results in numeric instabilities when updating the matrix A.

After the winning node has been updated, the classical ITM algorithm uses local relations between the new input, the winning node and the second node in order to determine if a new node should be inserted or if some node should be deleted. As long as there are no big jumps in consecutive training samples, this approach works quite well. However, when resetting the environment between consecutive training runs, larger gabs can arise. Furthermore, this network has already been extended by an output function which can now also be used during training. This AITM inserts new nodes into the network based on the difference between the current output of the network and the target output:

$$||\vec{y}^{net}(\vec{x}^{\alpha}) - \vec{y}^{\alpha}|| > \epsilon_{ITM} \tag{4.13}$$

[12]Usually one initializes the matrix A at random or with prior knowledge. However, in this thesis a zero matrix is used in order to avoid using invalid linear interpolations while the matrix has not been adapted to the data.

If the output varies too much, a new node at the position of $\vec{x}^{\alpha}$ and output $\vec{y}^{\alpha}$ is created unless its distance to the winning node is too small

$$||\vec{x}^{\alpha} - \vec{w}_{in}^{n}|| \leq \epsilon_{max} \tag{4.14}$$

ϵ_{max} represents the maximum distance nodes should be apart and needs to be chosen to represent the order of magnitude of the norm of the input vectors. If the new input is too close to the winning node although the output was not similar enough, the output of the winning node is adapted with:

$$\Delta\vec{w}_{out}^{n} = 0.5 \cdot (\vec{y}^{\alpha} - \vec{w}_{out}^{n}) \tag{4.15}$$

This ensures, that noisy data does not lead to the creation of multiple nodes with similar input vectors.

The threshold $\epsilon_{ITM} = 10^{d}$ is dynamically computed, based on the order of magnitude d of the target output norm:

$$d = \begin{cases} \lfloor \log_{10}(||\vec{y}^{\alpha}||) \rfloor & \text{, if } ||\vec{y}^{\alpha}|| > 0 \\ \lfloor \log_{10}(||\vec{y}^{\alpha}||) \rfloor - 1 & \text{, if } log_{10}(||\vec{y}^{\alpha}||) = 0 \\ -k & \text{, otherwise} \end{cases} \tag{4.16}$$

When the output has a norm of 0 a fixed threshold 10^{k} is chosen. Ideally k should represent the average order of magnitude of the input. This average can be computed incrementally from the non-zero output norms[13]. The benefit of such a dynamic threshold is that it automatically adapts to different use cases. For example, when the AITM is used for classification, the output values will be class labels in the form of positive natural numbers. In this case d equals -1 which results in a threshold of 0.1 which allows to distinguish conflicting class labels[14]. In regression tasks however, the output values will be real numbers. It is obvious that different thresholds are required for both types of use case. The AITM assumes that the orders of magnitude of the output within one use case are generally rather similar and can be used as an approximation of the desired accuracy.

[13] In this prototypes a fixed $k = 3$ is chosen instead.

[14] The special treatment for this case is done in order to conform to the $>$ in equation 4.13, the case can be omitted if $>=$ is used instead, however testing for the equal case is less robust in the regression case.

With every update the two winning nodes are connected as neighbors. Node deletions are performed just as in the traditional ITM: Second winners are removed if they are too far away from the winning node, i.e.:

$$||\vec{w}_{in}^{n} - \vec{w}_{in}^{s}|| > \epsilon_{max} \tag{4.17}$$

Furthermore, isolated nodes will be deleted. A node is considered isolated if it does not have any neighbors left. Neighbor connections are removed if the second winner in an update can replace a previous neighbor:

$$\forall c \in N(n) : \text{If } (\vec{w}_{in}^{n} - \vec{w}_{in}^{s}) \cdot (\vec{w}_{in}^{c} - \vec{w}_{in}^{s}) < 0 \text{ remove connection (n,c)} \tag{4.18}$$

N denotes the set of neighbors of the winning node n.

All norms that are used here are euclidean norms which might not be ideal depending on the used features. However, since the models should not be provided with additional knowledge about the features, this common choice was made.

Chapter 5

Evaluation

The implementations of the developed concepts were evaluated with regard to the three requirements stated in the introduction:

- Update models incrementally during the interaction
- Predict future object states
- Incrementally reach a given target configuration

In order to evaluate these requirements, the implementations are tested in two different tasks using a physics simulation which is explained in section 5.1. The first task, the Push Task Simulation, tests the forward models of the different implementations. The actual task as well as the results achieved in that task are explained in section 5.2. The second task, Move to Target, tests the implementations' ability to move an object towards a given configuration. This task and its results are explained in section 5.3.

5.1. Simulation and environment

The models are supposed to learn object manipulation through interaction. The developed prototype implementations are tested in a simplified simulation. For this work gazebo [45] version 2.2.3 was used with the physics engine Open Dynamics Engine (ODE) [46]. Gazebo simulates physics one step at a time at fixed time intervals. It is possible to configure the maximum step size for each update step as well as the number of steps to be performed in one second in real time (real time update rate). The simulation time depends on these two values. For this thesis, the standard step size of 0.001 simulation seconds is used. This value represents the timestep which is used to predict the next state of the simulation. The real time update rate, which determines how many steps are performed in one real time

Object	Dimension	Friction coefficients		Mass
		μ_1	μ_2	
Actuator	$0.025m$	0.01	0.01	10kg
Blue rectangular block	$0.5m \times 0.1m \times 0.1m$	0.9	0.9	1kg
Red square block	$0.3m \times 0.3m \times 0.1m$	0.9	0.9	1kg

Table 5.1.: Summary of the dimensions, friction coefficients and masses for the objects in the environment. If only one dimension is given, it represents a radius for a spherical object. Three dimensions correspond to the width, depth and height of an object respectively.

second, is used to control the speed at which the simulation actually runs. This thesis uses an update rate of 500 when the models are to be updated at 100Hz in simulation time. This setting requires the models developed in Python to finish all their updates and queries within 0.02 seconds before the next update. The simulation sends information about the objects every x simulation steps, where x depends on the chosen update rate and step size. When using 100Hz, the simulation publishes the objects' information every 10 steps of physics simulation.

A very simple two dimensional environment is used, only containing a spherical actuator and a rectangular cube object such as the one that can be seen in figure 5.1. In all but one testing scenarios only one of two possible block objects is used mainly due to the reasons discussed in sections 3.2.4 and 3.3.3. The dimensions of all objects as well as their friction coefficients and masses used by the physics simulation are listed in table 5.1. The fairly high friction coefficient for the blocks was chosen in order to better suit the assumption of the object state model. By increasing the friction the blocks are less likely to slide. The high mass for the actuator was chosen so that the used velocities actually move the other object since forces are not set directly. The red block shows less rotatory dynamics due to its shorter edges than the blue block

The used action primitives were chosen based on their ease of implementation in the simulation, since it is quite easy to change the global velocity of an object at runtime. For simplicity the spherical actuator's orientation is fixed to 0 so that the global velocities always correspond to the local coordinate system of the actuator.

All experiments were performed on the same computer using an Intel Core i7-5820K @ 3300 Mhz CPU.

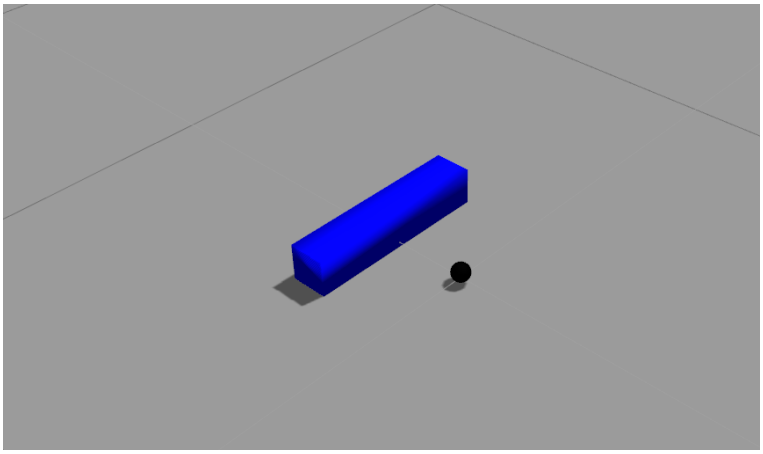

Figure 5.1.: Overview of the used environment. The black sphere represents the actuator, the models can control using action primitives. The blue square represents the object that the actuator interacts with.

5.1.1. Communication

Gazebo uses a client-server architecture. The actual simulation is being performed by the server. This server publishes Google Protobuf [47] messages via TCP/IP. Interested clients, for example a graphical user interface, can register themselves as listener to messages of desired types. It is also possible to send messages to the server in order to influence the simulation.

The server can furthermore be extended by writing custom plug-ins that handle self defined messages or perform some sort of additional computation at each update step of the simulation.

In order for the Python prototypes to communicate with the simulation, an interface on both sides was created.

On the side of the simulation, a custom server plug-in was written. This plug-in publishes the information described in table 4.1 in the realization chapter at the fixed rate described above. The information for all objects is packed into a custom Protobuf message, which is explained in details in Appendix A.2. Basically, this message simply contains a list of object descriptions where each object description contains the information explained in table 4.1. Furthermore, the plug-in receives custom control messages. These messages allow the Python interface to influence the simulation. While the exact messages are explained in the Appendix A.2, table 5.2 gives an overview about the possible commands, that can be send to the simulation.

Command	Meaning
Move Command	Sets the velocity for the actuator
Pause	Pauses the simulation
Continue	Continues the simulation
Reset	Resets all objects to starting configuration
Set Pose	Places a specified object at a certain position and orientation

Table 5.2.: Overview of all implemented commands to influence the simulation.

On the Python side, an interface was written with the help of the module pygazebo[1], which provides Python bindings for the message passing system used by gazebo. This interface handles the messages that are received from and send to the simulation. At each update step, the interface performs four to five actions:

1. Construct a suitable worldstate from the provided information
2. Update the model with the current worldstate
3. Get an action primitive from the model given a target configuration
4. Get a prediction about the next worldstate from the model using the current worldstate and chosen action primitive
5. Send the current action primitive to the simulation

The 3rd step is only performed in the tasks evaluating the inverse model. In the Push Task Simulation a fixed action primitive is predetermined.

Furthermore, the Python interface provides the testing framework of setting up the models and managing training and testing runs as explained in detail below.

5.1.2. Sources of noise

Although the object's attributes can be read accurately from the simulation, some random noise is still present in the data received by the prototypes:

1) The physics simulation and the sensors that record the objects' attributes are run in different threads within the simulation. These threads cannot always be perfectly synchronized which can result in minor differences in consecutive timesteps.

[1]Pygazebo is a Python module developed by Josh Pieper: `https://github.com/jpieper/pygazebo` (Last accessed November 2015).

2) The physics simulation itself might encounter numerical instabilities, especially when computing resting forces. These instabilities might lead to small oscillations within an object's features, such as position. In order to filter out such small noise, all information from the simulation is rounded to 3 decimal places while constructing the worldstate.

3) The strongest variation comes from the fact that the implementation communicates with the simulation via asynchronous messages. All runs start in a configuration where all objects are resting. In the very first update step, the Python interface sends the first action primitive to the simulation, which results in the actuator moving. However, the exact time when the simulation retrieves and executes the action is not deterministic. Depending on the system's load the simulation might have already performed 5 update steps before it receives and executes the action. In this case, the action affects only half of the update steps. In another situation, the action might be executed earlier. From the model's point of view, the same action was performed from the same initial situation, but the resulting changes in object states can differ greatly.

Since this thesis considers a rather simple scenario and these three sources of noise are already present due to the used technologies, no additional artificial noise was introduced to the system. Both models are expected to be flexible enough to learn from the noisy data without fixating on outliers.

5.2. Push Task Simulation

The Push Task Simulation is designed to test the accuracy of the forward model of both concepts.

5.2.1. Scenario description

In this task, the actuator is controlled by a constant action primitive which first moves the actuator towards the blue object and then pushes it. In order to evaluate different interactions, the actuator starts at different starting positions on a line parallel to the object.

The distance between the starting line and the center of the object is 25 cm. With the used speed, the actuator reaches the object in about 35 update steps after the start of a run. Depending on the third issue explained above, this number can vary a little.

After 150 update steps, a run ends and the objects are reset. This means, that the object will be placed at its resting position and the actuator is moved back to the starting line.

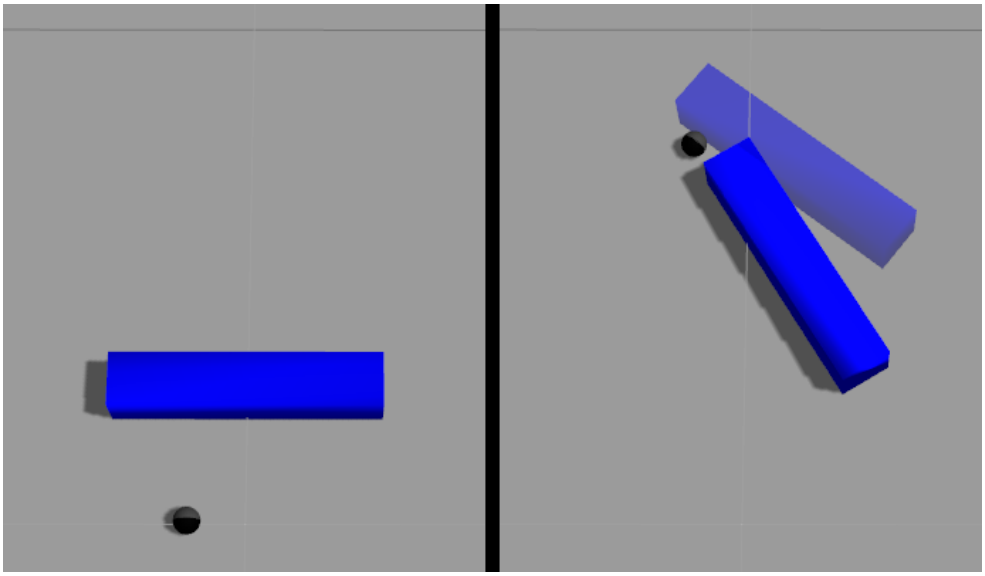

Figure 5.2.: Exemplary start and end configuration of a Push Task Simulation run. The left side shows the start configuration, while on the right the corresponding end configuration can be seen. The opaque objects (blue and black) represent the actual block and actuator, while the transparent objects symbolize the predicted states of the objects.

During a run, the object's status can change in a number of ways. In case the actuator goes past the object, it will not change at all. If it is pushed in the center, it will be translated up to 0.6 m. Furthermore, depending on where the object is pushed, it will be rotated up to 1.02 rad (around 58°) in either direction. The maximum rotation is experienced when the actuator starts around 0.105 m away from the origin.

An example starting and end configuration can be seen in figure 5.2.

As the name suggest this task is designed to evaluate the prediction performance of the models during the pushing interaction. The "simulation" part of the name comes from the fact, that the models will make consecutive predictions based on their last prediction. At the start of each run, the models will make their first prediction based on the current worldstate provided by the simulation. This prediction is then used to construct an appropriate next worldstate which is used at the next update step from the simulation. Furthermore, the predicted states of both the actuator and the object are send to the simulation for visualization as can be seen in figure 5.2.

The models will not get any feedback from the real environment in this setting during testing, which means that they will not be able to adapt their predictions based on the actual interaction. The only exception to this is the adapted experiment mentioned below.

In order to be able to make predictions at all, the model will first perform a number

of training runs. During these training runs, the models are provided with the current worldstate from the simulation in order to train their local models.

In all runs, an action primitive is used that sets the velocity of the actuator to $0.5\frac{m}{s}$ straight upwards. Although only a constant global action is used, the prediction task is still challenging enough:
The constant action primitive does make it easy for the gating concept to learn the forward model of the actuator. However, the local forward model for the object is not as simple because the velocity is not constant in local coordinates but rather depends on the object's orientation. Similarly, in the interaction state concept the used action primitive is also transformed to the local coordinate frame of the reference object.

Different kinds of training scenarios are evaluated in this task which are further explained below when their results are presented.

5.2.2. Evaluation criteria

This task tries to evaluate the precision of the different models. In order to measure this precision, the distance between the predicted and the actual object state is computed at the end of each test run. In this scenario, the models predict up to two quantities: The position and the orientation.

Finding a metric that can adequately combine differences in position and orientation at the same time is difficult. Furthermore, the interpretation of such a measurement is not intuitive. Therefore, the differences in position and orientation are considered separately. The positional difference is computed by the euclidean distance between the centers of the predicted and the actual object. The difference in orientation is given by the absolute difference between the predicted and the actual orientation.

Each training and testing setting is repeated 20 times and the average differences in position and orientation as well as their standard deviations are reported.

5.2.3. Evaluated configuration

The implementations of both concepts offer several possible configurations, ranging from using different features to using hard coded components in the gating concept and choosing different learning rates for the underlying regression and classification models.

In order for both models to be more comparable, the local actuator models as well as the gating function were also learned online in the object state concept, even though it is possible to provide accurate models by hand in this situation.

Since the goal of this thesis was to provide models that adapt to their environment with as little prior knowledge as possible, standard parameters are chosen where possible.

In this evaluation, the implementations are evaluated in the configuration described in the last chapter with meta parameters chosen as indicated by table 5.3. Both models use almost the same set of meta parameters. While the interaction state model specifies ϵ_{noise} to determine the set of changed features and ϵ_{min} to determine the minimum number of episodes within an AC, the gating model uses ϵ_{change} to determine when an object state has changed.

	Parameter	Value
Simulation	Step size	0.001 s
	Real time update rate	500
	Update rate models	100 Hz
	Number decimals	3
AITM	σ	0.05
	ϵ_{max}	0.001
	η_{in}	0
	η_{out}	0
	η_A	0
Episodes	ϵ_{noise}	0.01
Abstract Collection Selector	ϵ_{min}	1
Gating function	ϵ_{change}	0
AIM	ϵ_{index}	0.002
	ϵ_{Node}	0.001
	Number of differing dimensions	1
Move To Target	ϵ_{close}	0.1
	$\epsilon_{closeEnough}$	0.01

Table 5.3.: Summary of the meta parameters used for the presented evaluations. The learning rates η_x are the same for all instances of the AITM.

5.2.4. Results

Generalization on random training data:

The first test evaluates the prediction accuracy against the number of random training runs. The training run start positions were chosen at random in the range of $-0.25\,\mathrm{m}$ to $0.25\,\mathrm{m}$. This means that they span the entire width of the (blue) block objects. For each

number of training runs that were evaluated, 21 test runs were performed with starting positions in the range of −0.35 m to 0.35 m at 0.035 m intervals. This means that 6 test runs went past the object without interacting with it. The averaged results for the blue block predictions over all folds for the interaction state model can be seen in figure 5.3. The averages over all test positions as well as only over the 15 central test positions are shown.

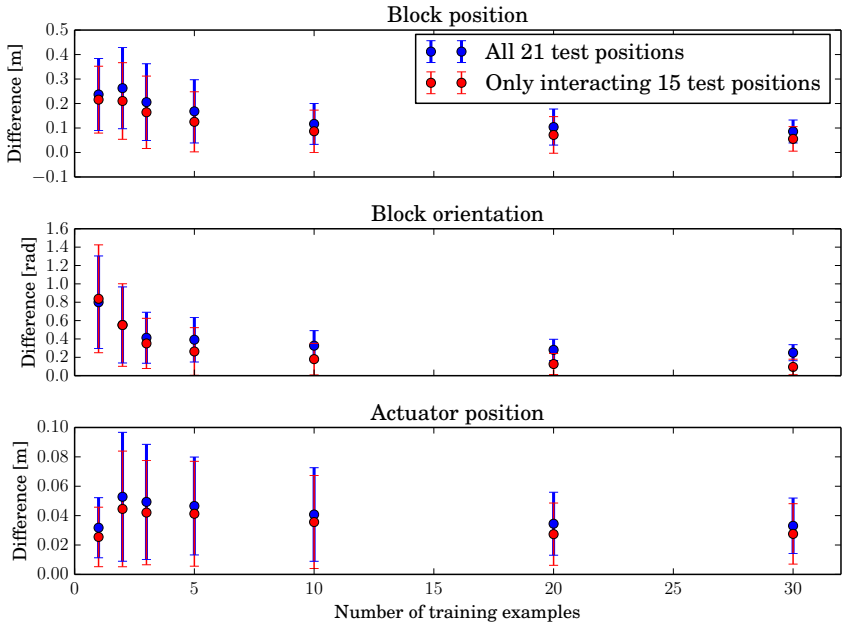

Figure 5.3.: Learning curve for the interaction state concept with the blue block. Difference in position and orientation for the block predictions and differences for the actuator position at the end of a run are shown. Error bars indicate one standard deviation. Blue error bars represents averages over all 21 test positions, while red error bars only averages over the 15 test positions where the block is actually moved.

The positional prediction error starts around 0.24 m with a standard deviation of around 0.15 m when the model is only trained with a single training example. The prediction performance improves to roughly 0.09 m with a standard deviation of around 0.05 m. The predicted orientation error starts at an average of about 0.8 rad with a standard deviation of 0.5 rad but improves to around 0.25 rad with a standard deviation of 0.09 rad when increasing the number of random training examples. The performances for position and orientation do not increase much more after the first 10 training runs (0.12 m +/−[2] 0.08 m for position and 0.33 rad +/− 0.16 rad for orientation). The actuator position predictions errors are ranging between 0.03 m and 0.05 m with standard deviations between 0.02 m and 0.04 m.

[2]The notation +/− is used to as an abbreviation for " with a standard deviation of ".

When only considering the testing positions where the actuator actually interacts with the block (red error bars), predicted errors improve slightly.

The results for the same experiment with the gating model using are presented in figure 5.4.

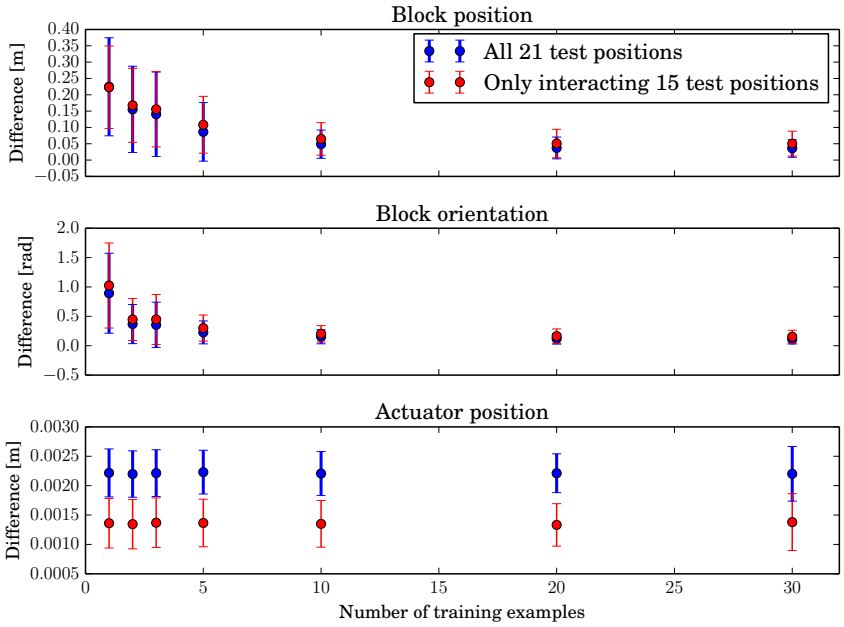

Figure 5.4.: Learning curve for the gating concept using the blue block. Difference in position and orientation for the block predictions and differences for the actuator position at the end of a run are shown. Error bars indicate one standard deviation. Blue error bars represents averages over all 21 test positions, while red error bars only averages over the 15 test positions where the block is actually moved.

The positional prediction error over all testing positions (blue error bars) starts at around 0.22 m with a standard deviation of 0.15 m for only one training run in the object state with gating function model. By increasing the number of random training runs to 30, the positional error reduces continuously to around 0.04 m with a standard deviation of only around 0.03 m. The predicted orientation error starts at around 0.89 rad with a standard deviation of 0.68 rad. After 30 test runs an error of 0.10 rad with a standard deviation of 0.08 rad remains. For this model the performance increases only slightly after having seen 10 training runs for both the error in position (0.05 m +/− 0.04 m) as well as orientation (0.18 rad +/− 0.12 rad). Predictions for the actuator position are fairly constant around 0.002 m with a standard deviation of around 0.0004 m.

Unlike in the other model, prediction errors are slightly worse for the block position and orientation when only considering the 15 testing positions where an actual interaction takes

Train runs	AC Updates	AC Nodes	# ACs	ACS Updates	ACS Nodes
1	59.74 (7.93)	17.58 (13.14)	6.9 (0.45)	149	39.65 (11.49)
2	98.37 (24.63)	28.32 (21.68)	9.58 (1.93)	298	85.45 (17.22)
3	132 (27.26)	28.9 (19.10)	10.37 (2.25)	447	124.35 (22.57)
5	224.84 (29.69)	51.05 (25.84)	11.42 (2.23)	745	196.75 (29.23)
10	453.37 (62.96)	106.21 (61.37)	12.63 (2.98)	1490	372.55 (21.93)
20	876.11 (78.19)	182.53 (94.63)	15.58 (1.09)	2980	740.15 (47.08)
30	1209.89 (146.47)	239.42 (130.38)	16.32 (1.03)	4470	1086.95(76.74)

Table 5.4.: Record of the average number of nodes that resulted from the recorded number of update calls in the regression and classification AITM for the interaction model. Only the number of the AC with the most nodes is used in each repetition is used to compute the averages for the AC Updates and AC Nodes. Numbers in brackets represent one standard deviation.

place. However, actuator prediction are slightly more accurate (average error improves by about 0.001 m).

Memory-based regression methods such as Nearest Neighbor (NN) approaches often suffer from the increasing computational complexity as the number of training samples increases. Table 5.4 presents the average numbers of update calls and the number of nodes in the AITMs used by the interaction state model after training. Since the number of learned ACs varies between the runs, only the number of nodes for the AC with the most nodes is recorded. Only 149 update calls are made at most for each run because the models are not updated after the 150th timestep.

The data in the table indicates, that one AC dominates the space since the number of AC Updates for the AC with the most nodes is significantly bigger than the average number of Update calls divided by the average number of ACs. The ACs appear to keep a node for every 3 to 4 update calls. The ACS, which is trained on every update, also creates a node for about every 4th update.

Similarly, table 5.5 shows the respective number of update calls and nodes for the AITMs used by the gating model.

Interestingly, the number of nodes required for the gating function is rather constant whereas the local forward model in the predictor creates a node between every 4th and 5th update call on average.

The number of nodes learned in the local forward model of the actuator is not included in the table since it only contains two nodes in all runs regardless of the number of training runs.

Train runs	Object Updates	Object Nodes	Gate Updates	Gate Nodes
1	93.5 (16.87)	26.55 (11.01)	149	3.7 (0.46)
2	194.35 (27.43)	39.4 (15.87)	298	5.55 (1.07)
3	287.25 (33.85)	67.4 (20.44)	447	6.4 (1.07)
5	479.6 (42.91)	111.95 (20.46)	745	6.8 (1.07)
10	963.7 (37.04)	219.65 (38.24)	1490	7.75 (0.89)
20	1894.25 (46.3)	438.2 (45.17)	2980	10.4 (6.88)
30	2883.8 (111.4)	603.05 (36.16)	4470	10.2 (4.23)

Table 5.5.: Record of the average number of nodes that resulted from the recorded number of update calls in the regression and classification AITM for the object state model. Values in brackets represent one standard deviation.

Generalization on selected training data:

In order to analyze the information required for generalization, another training scheme was used. This time the models were only provided with three predetermined training runs. The starting positions $-0.18\,\text{m}$, $0.0\,\text{m}$ and $0.18\,\text{m}$ were used in this experiment. These three were chosen in order to provide the models with examples of both possible rotations as well as the lateral movement. The same 21 test runs were performed as before. This time prediction errors at the end of the run are recorded for each starting position separately. Due to the noise explained above, some randomness is still present in the data which is why this experiment was also repeated 20 times and the average errors are reported.

The results for the interaction model are shown in figure 5.5.

The predictions for the test runs that do not actually interact with the block, are rather poor. The model predicts an incorrect change in position around $0.18\,\text{m}$ +/- $0.01\,\text{m}$ and in orientation around $0.77\,\text{rad}$ +/- $0.02\,\text{rad}$ on both sides of the object. Predictions are quite good close to the seen training examples when the block is actually influenced: The positional error is as low as $0.02\,\text{m}$ +/- $0.01\,\text{m}$ around the two outer training positions and slightly higher for the central position ($0.03\,\text{m}$ +/- $0.02\,\text{m}$). Generally the same applies for orientation, but the model does not make a prediction error at the central position. At the outer positions slight prediction errors are made ($0.01\,\text{rad}$ +/- $0.01\,\text{rad}$). The further away from the training positions, the worse the prediction for both position and orientation becomes. Interestingly, the worst positional prediction of $0.07\,\text{m}$ +/- $0.02\,\text{m}$ happens $0.105\,\text{m}$ away from the center, while the worst prediction for orientation of $0.45\,\text{rad}$ +/- $0.01\,\text{rad}$ is seen $0.07\,\text{m}$ away from the center.

It is also interesting to note, that the position and orientation are predicted very consistently (most standard deviations below $0.02\,\text{m}$ and all below $0.02\,\text{rad}$).

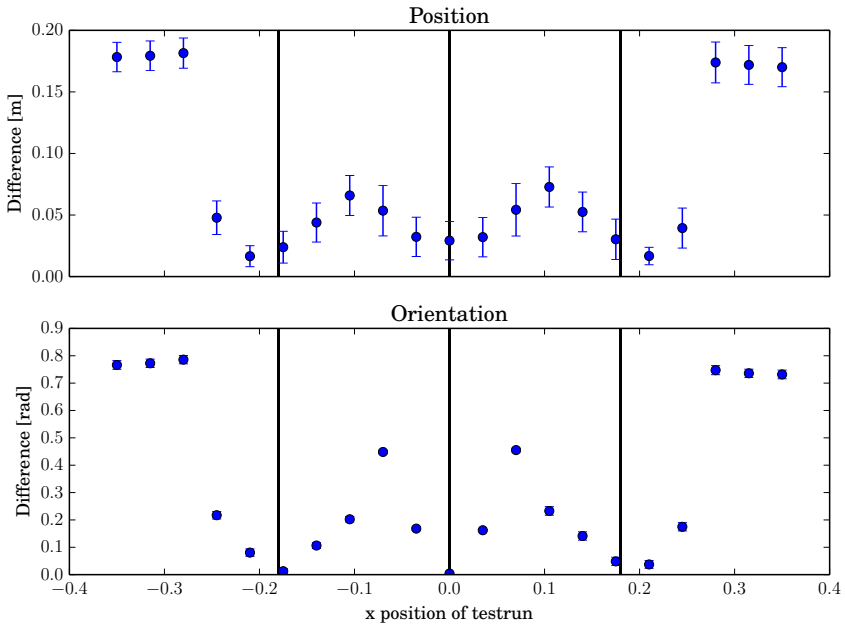

Figure 5.5.: Prediction errors for the interaction model at each of the 21 test positions after seeing three exemplary training examples. The training positions are highlighted by the block lines. Error bars indicate one standard deviation.

The results for the same experiment performed with the gating model are presented in figure 5.6.

The gating model successfully does not make predictions for the test runs that go past the object after seeing the three fixed training runs. However, no predictions are also made for the first interacting test positions on either side which results in constant errors in position of 0.14 m and in orientation of 0.58 rad. For the other test positions that actually interact with the block, the predictions are better for positions close to the training examples as in the other model. The predicted errors are not quite as symmetric as in the other model. The central position has a prediction error of close to 0 with a standard deviation below 0.01 m. The predictions for position are slightly more accurate between the left and the middle training position (ranging from 0.01 m +/- 0.0 m to 0.04 m +/- 0.02 m) when compared to the testing positions between the middle and the right training position (ranging from 0.01 m +/- 0.0 m to 0.07 m +/- 0.03 m). Outside the training positions, the predictions are better for the 17th testing position than for the fifth one (0.09 m +/- 0.04 m at the fifth and 0.06 m +/- 0.01 m at the 17th testing position).

Prediction errors for orientation show a similar trend: In the central position the prediction error in orientation is 0.04 rad +/- 0.02 rad. The same outliers for orientation are found as in the other model. The testing positions 0.07 m away from the center have errors around 0.45 rad +/- 0.04 rad. The remaining errors for orientation in between the training

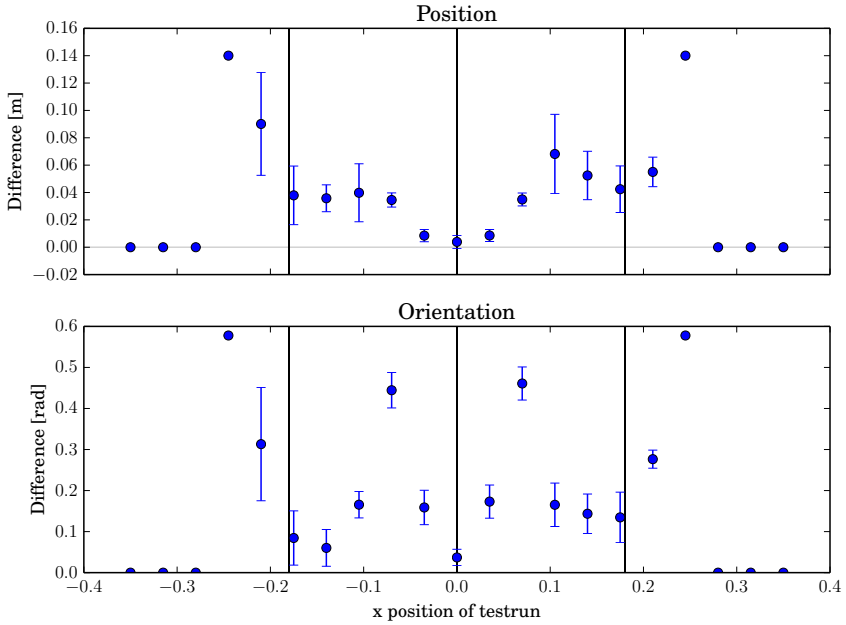

Figure 5.6.: Prediction errors for the object state model at each of the 21 test positions after seeing three exemplary training examples. The training positions are highlighted by the block lines. Error bars indicate one standard deviation.

positions are ranging from 0.06 rad +/- 0.04 rad to 0.16 rad +/- 0.05 rad. As with the positional prediction, the prediction for orientation is slightly worse for the fifth testing position compared to the 17th one, although both have the same distance to the closest training position (0.31 rad +/- 0.14 rad at the fifth and 0.28 rad +/- 0.02 rad at the 17th testing position).

In order to be able to interpret these results, figure 5.7 shows final configurations (predicted and actual) for every 2nd testing positions for both models.

The number of required nodes for this experiment is slightly lower then when learning on random data and show hardly any variance. The interaction state model consistently trains eight ACs of which the biggest one contains around 6 nodes. The ACS requires 72.6 nodes on average. The gating model constantly requires 9 nodes in the gating function and around 58 nodes for the local forward model of the block.

Generalization with constant feedback:

While the previous scenarios test the accumulated prediction error, it is often not required to make accurate predictions 150 steps into the future. Instead it is more important to adapt to changes in the environment quickly. For that reason, a variation of the Push Task Simulation was also evaluated. In this variation, the models are not trained before testing, but will instead continuously receive updates from the environment. That way the

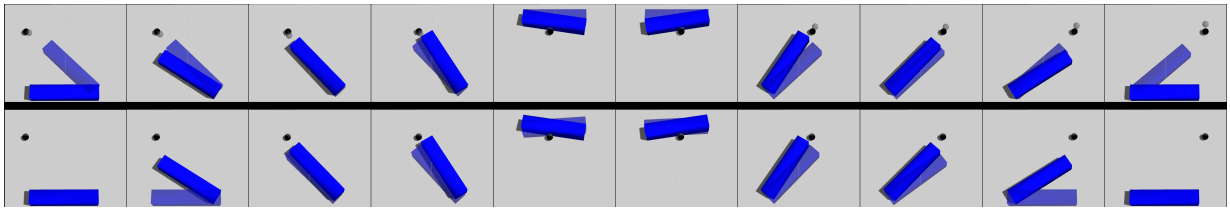

Figure 5.7.: Predicted and actual end positions. Transparent objects represent the predictions. Top row shows results for the interaction model, while bottom row shows the results for the object state with gating function model. Every 2nd of the 21 testing positions is shown. The training positions are right next to the third image on either side as well as between the two central images.

models' online learning capabilities are evaluated. The models will still make consecutive predictions based on their last prediction. Their last predictions are not corrected based on the actual environment and the models do not get or compute any feedback as to how good their last prediction was.

A qualitative evaluation is provided in figure 5.8. Each row in the figure represents one run as described before. The leftmost images show the initial configuration at the start of the run. The rightmost images show the interaction step where the block prediction changed for the last time. The first row shows the first run as performed the gating model. An image sequence for the interaction state model looks almost identical and was therefore omitted. The second row shows the second run performed by the gating model, while the third row shows the same run performed by the interaction state model.

The transparent object represent the last predictions, while the opaque objects represent the actual states of the objects. In order to be able to actually see meaningful differences between consecutive interactions, these images where taken in a setting where the models are only updated every 10Hz, meaning that the models were updated and queried every 0.1 seconds in simulation time instead of the 0.01 seconds used for all the other evaluations. No other adaptation to the models was made.

While no prediction is made for the actuator in the 2nd image of the top row, the actuator is predicted correctly in the 2nd run. The top row continues to lag one frame behind with its predictions. The final block predictions are accurate again. In the 2nd run, the actuator predictions are good. However, the models predict the same rotation as in the first run once the actuator comes close to the object. After a few other frames, the predictions start turning in the other direction, but do not catch up to the actual object. The gating model does not change its prediction for the block anymore after the actual actuator passed the actual block. However, the interaction state model continues to make predictions until it reaches the point shown in the image which leads to a better prediction at the end of the

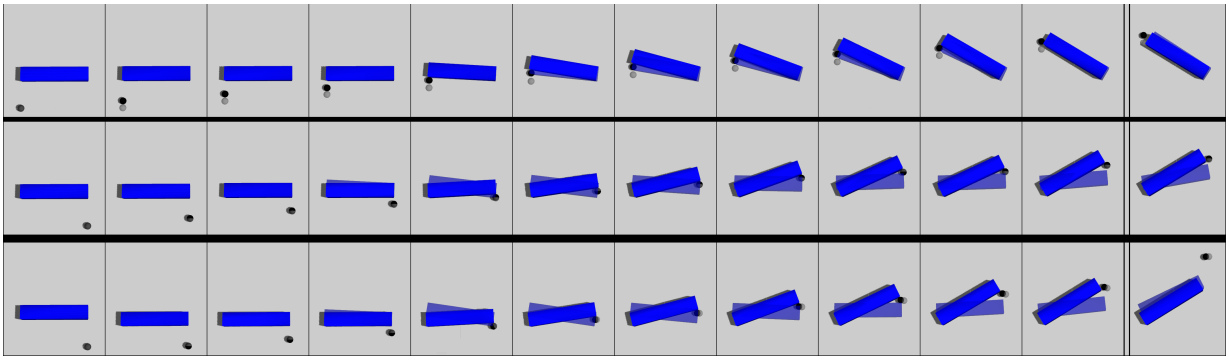

Figure 5.8.: Consecutive interactions in the Push Task Simulation variation. Top row shows images of the first test run, while the second row shows images of the second test run on the other side of the object for the gating model. The third row shows the same second test run performed by the interaction state model.

run.

All three experiments have also been performed using the red object. However, the results are qualitatively very similar and partly better, especially for the experiment with fixed training runs, which is why they are not reported here.

5.2.5. Extension to multiple objects

Environments rarely only contain a single object, however, so far in this thesis, only one object was considered. While it is difficult for the interaction model to handle multiple objects due to the reasons explained in section 3.2.4, the object state model handles objects separately anyways. In this evaluation the red block described in table 5.1 is added to the scene and six fixed training positions (three interacting with each object) are used. The blue block remains at the same position as in the previous experiments, while the red block is located left of it with a center position of $-0.6\,\mathrm{m}$ in x direction and $0.35\,\mathrm{m}$ in y direction. This position ensures that the distance from the starting line to the edge of the block remains the same compared to the blue object. The object is then tested on a fixed set of 21 testing positions ranging from $-0.8\,\mathrm{m}$ to $0.4\,\mathrm{m}$ in intervals of $0.06\,\mathrm{m}$.

The model was not altered in any way for the results presented in figure 5.9.

The gating function often classifies incorrectly for the red object, but the blue object does not seem to be influenced by the new object at all since the results are basically identical to the ones presented in 5.6. Within the training positions for the red object, the predictions are fairly good (below $0.05\,\mathrm{m}$ +/- $0.02\,\mathrm{m}$ for position and below $0.09\,\mathrm{rad}$ +/- $0.06\,\mathrm{rad}$).

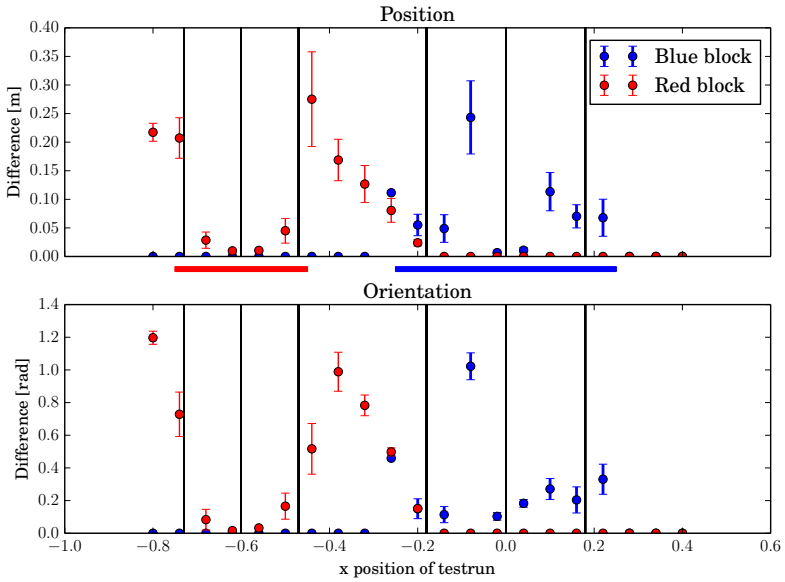

Figure 5.9.: Prediction errors for the object state model with two objects. The training positions are highlighted by the black lines. Error bars indicate one standard deviation. The red and blue bar in between the blocks visualize the size and position of the red and blue blocks respectively.

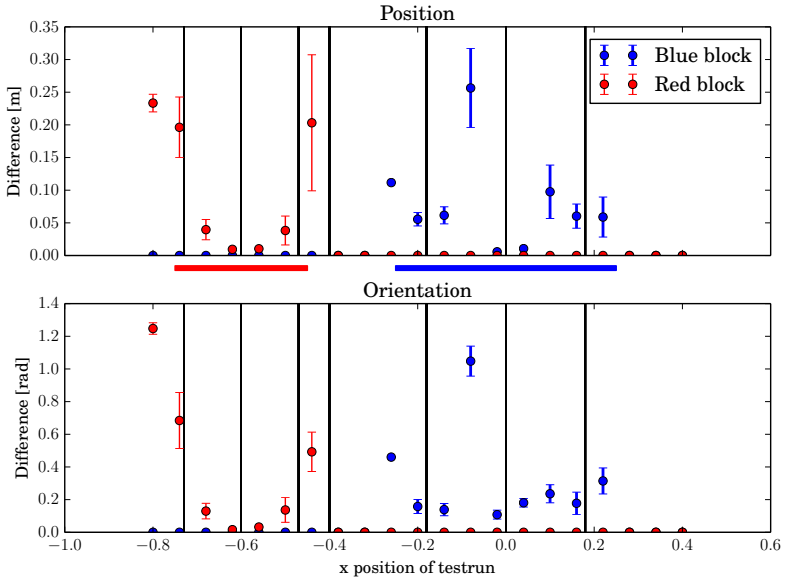

Figure 5.10.: Prediction errors for the object state model with two objects with an additional separating training positions. The training positions are highlighted by the black lines. Error bars indicate one standard deviation. The red and blue bar in between the blocks visualize the size and position of the red and blue blocks respectively.

By introducing a seventh training position in between both objects, the prediction quality of runs that do not interact with the red object improves due to better performance of the gating function as can be seen in figure 5.10. The other predictions remain mostly unchanged through the introduction of the additional training run.

As in the experiment before, figure 5.11 shows the predicted and actual configuration for every second testing positions in order to allow for easier interpretation of these results.

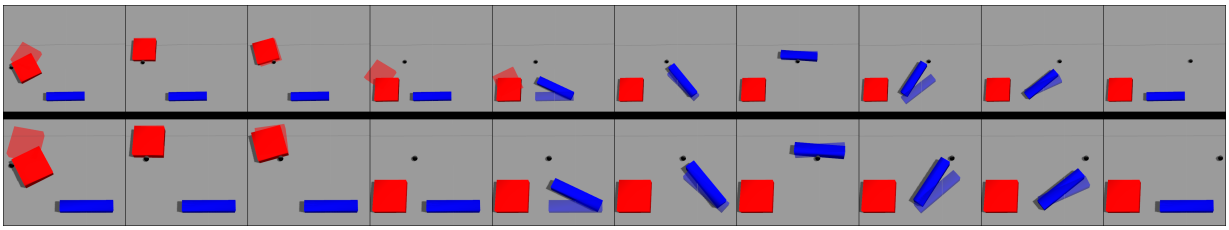

Figure 5.11.: Predicted and actual end positions for the two object scenario. Transparent objects represent the predictions. The top row shows the results after having seen six training runs. The bottom row shows the results for the experiment adding a seventh training run in between both objects. Every 2nd of the 21 testing positions is shown.

5.3. Move to Target

The Move to Target task is designed to evaluate the concept's ability to reach a given target configuration incrementally. As mentioned in the realization chapters of both models, this thesis assumes that the inverse model has been sufficiently trained, i.e. it has seen all required changes in feature dimensions, before a target is specified.

5.3.1. Scenario description

In this task only a target configuration for the blue block object is given to the models. Only the blue block is tested because its dynamics are more complex than that of the red block. This is mainly due to the fact that the red block only rotates when pushed at the corners while the blue block rotates a lot easier.

At each update step the models are updated with the current worldstate. This means that they can adapt their inverse model during testing. Afterwards, the interface queries the model for the next action primitive which is then send to the simulation. This is repeated until the target configuration is reached or a maximum number of steps of 3000 has been performed. The models consider a target configuration as reached, if the norm

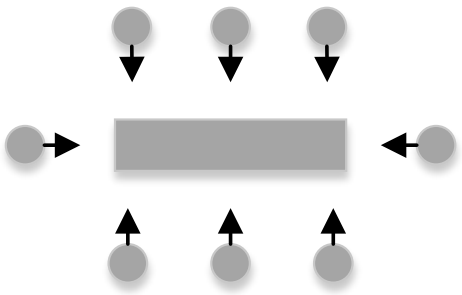

Figure 5.12.: Visualization of the eight training positions for the Move to Target task. The circles indicate the relative starting positions of the actuator, while the arrows indicate the used action primitive.

of the difference vector between the target representation and the current situation is smaller than 0.01. This means, that the combined vector of positional difference as well as difference of orientation must have a norm smaller 0.01. The models do not know about the oscillation of the orientation feature, i.e. that an orientation of $-\pi = \pi$ which can lead to false negatives.

The models are first trained on a fixed set of eight training runs. The set is designed to show important interactions to the models before testing. The eight training configurations are shown in figure 5.12. These training runs ensure that the inverse models can learn reasonable preconditions for all relevant feature changes.

The training runs are performed by placing the actuator in a suitable starting position before applying a constant action primitive directed towards the block. The used action primitives all have a norm of $0.5\frac{m}{s}$.

After training, both the actuator and the block are returned to their initial position. The block is located 0.25 m above the global origin without any rotation, while the actuator is placed directly on top of the global origin. Figure 5.13 visualizes the situation where the actuator just starts moving after training. The transparent blue block symbolizes the target configuration of the block. One edge is colored in order to visualize the block's orientation.

5.3.2. Evaluation criteria

The performance of the inverse model is measured by the number of actions required to reach the target. If the target has not been reached within the allowed number of steps,

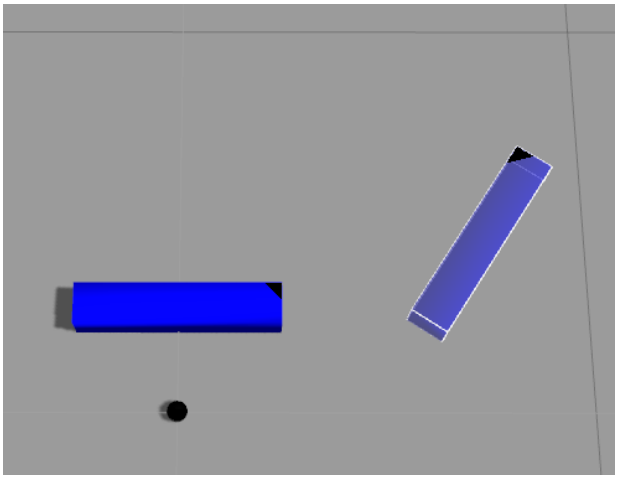

Figure 5.13.: Visualization of the Move to Target task after training. The transparent light blue block indicates the desired target configuration. The blocks have been marked with the black corner to distinguish their orientation.

the remaining differences in position and orientation are recorded.

The models are tested with four targets which are shown in figure 5.14. All target configurations have the same positional difference to the starting position (0.6 m in either x direction and 0.7 m in either y direction) but different orientations:

- Target 1: -2.1 rad
- Target 2: 0.75 rad
- Target 3: 0.0 rad
- Target 4: 3.1415 rad

For each target, the model is trained separately on the same training set in order to ensure comparability between the targets.

The testing of each target configuration is repeated 5 times and the averages over these runs are reported where possible.

5.3.3. Results

The results for the interaction state models are presented in table 5.6 while the results for the gating model are presented in table 5.7.

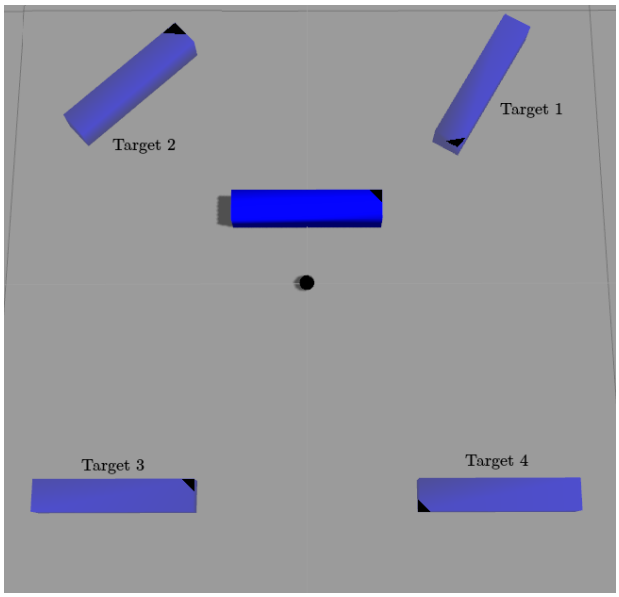

Figure 5.14.: Visualization of the four target configurations for the Move to Target task. The black corners indicate the block's rotation.

Target	Reached	Pos	Ori	# steps
Target 1	3	0.55	0.002	2689.2 (2482)
Target 2	4	0.01	0.023	2430.4 (2288)
Target 3	5	-	-	1377.6
Target 4	2	0.55	2.04	2492.2 (1730.5)

Table 5.6.: Evaluation results for the Move to Target task for the interaction model. Pos and Ori columns represent the remaining error in position [m] and orientation [rad] respectively for the runs that did not reach the target. The number of steps in brackets represents the average over the runs that reached the target.

Target	Reached	Pos	Ori	# steps
Target 1	4	0.01	0.002	2271 (2088.8)
Target 2	5	-	-	1695
Target 3	5	-	-	1147.4
Target 4	5	-	-	1597.4

Table 5.7.: Evaluation results for the Move to Target task for the gate model. Pos and Ori columns represent the remaining error in position [m] and orientation [rad] respectively for the runs that did not reach the target. The number of steps in brackets represents the average over the runs that reached the target.

The gating model reaches the targets in most cases within the allowed number of steps. Only target one was not reached in one run, however the remaining distance was quite small.

No target is never reached, but the interaction model had a few problems reaching the first and fourth targets. Looking at the combined error of position and orientation for the runs that did not reach the first and fourth target in detail revealed that in both cases, the model was not able to push the object straight without changing its orientation. The 2 runs that did not reach Target 1 as well as an exemplary third run that did reach can be seen in figure 5.15. In both failed runs, the model first reduces the difference in orientation successfully. However afterwards, it only manages to reduce the positional difference by rotating the object away from the target orientation. In the successful run on the other hand, the model manages to reduce the positional difference without influencing the objects orientation much.

Snapshots of a successful run towards Target 1 performed by the gating model can be seen in figure 5.16. The image shows how the model first turns the object in the correct orientation. Afterwards the model pushes the object upwards and then towards the right. Finally some minor adjustments regarding the orientation and the position are made.

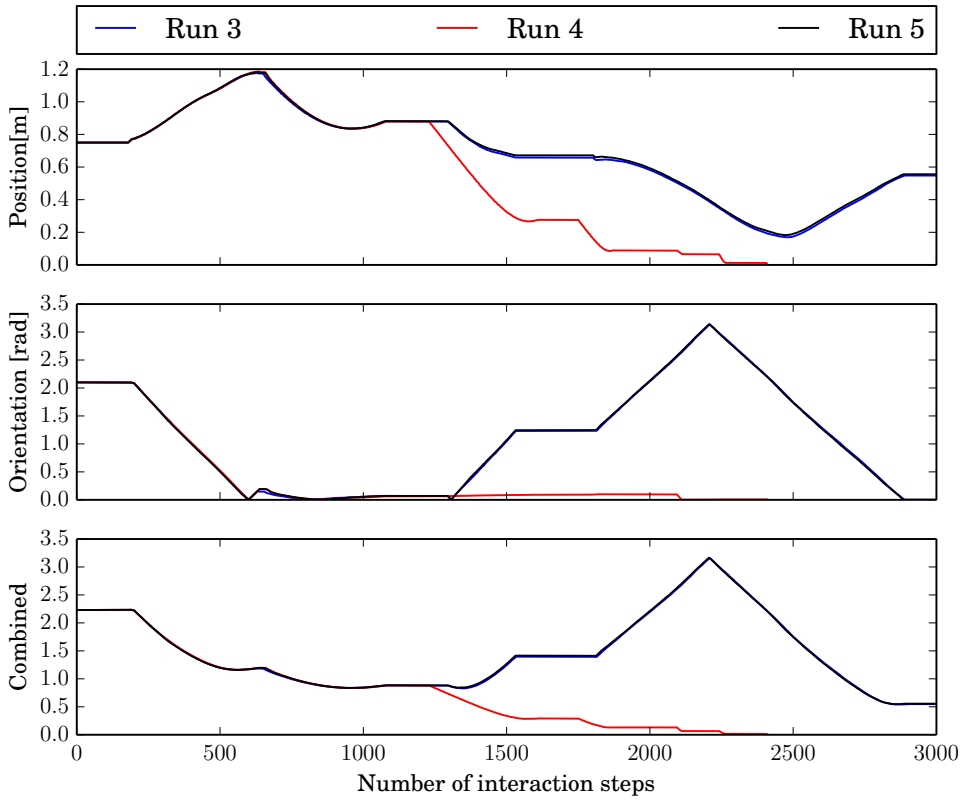

Figure 5.15.: Detailed view of three runs towards Target 1 performed by the interaction state model. The two runs that did not reach the target as well as one successful one are presented. The errors in position and orientation as well as the combined error over the interaction steps are shown.

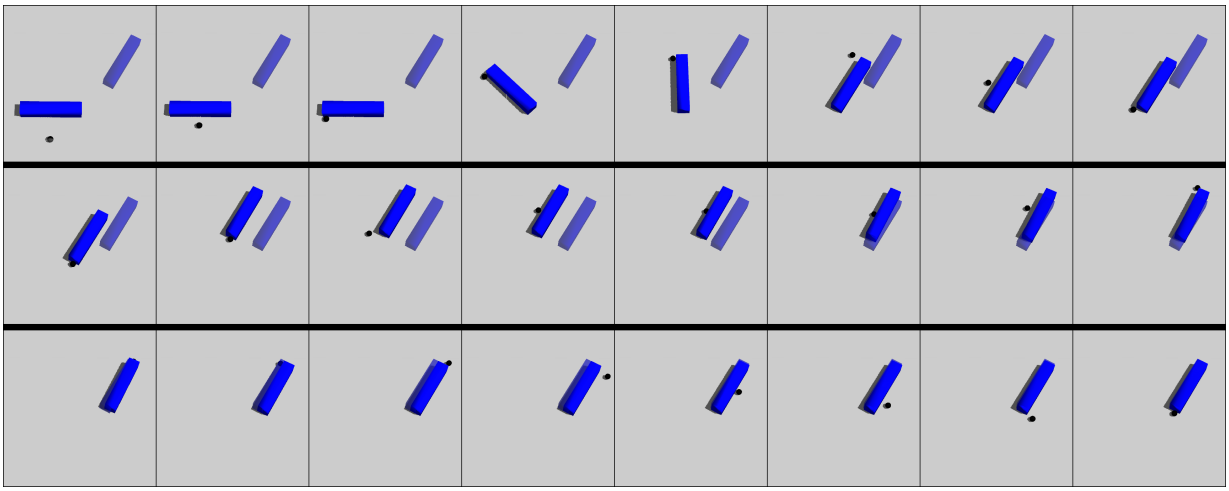

Figure 5.16.: Several snapshots from a run towards Target 1 by the object state model. The run started at the image at the top left and finished at the bottom right. There are around 50–150 updates in between two frames. The transparent block represents the target configuration.

Chapter 6

Discussion

The last chapter presented the results of the evaluations of both developed concepts. Overall it can be argued that both concepts fulfill the stated goal of incrementally learning about simple object interactions. This chapter discusses the results of the evaluations in more detail as well as discuss the developed underlying mechanism in the AITM and the AIM.

6.1. Generalization performance

The first task was designed to evaluate the generalization capabilities of the memory-based concepts. It is important to note, that no domain knowledge outside the stated assumptions was provided to either model for this prediction task. Although the used features were selected by hand, no preprocessing of these features was performed and no knowledge about special feature dimensions was provided. It is highly likely, that the generalization performance of both models could be increased by providing fine tuned distance metrics and/or preprocessing the used features. However, as this kind of knowledge is usually not available to the machine when encountering novel situations, such optimizations were not considered here.

6.1.1. Generalization on random training data

Prediction performances of both models reach acceptable levels for position and orientation considering 150 timesteps are predicted without correction. Both models' prediction for the blocks position are on average less then 10 cm incorrect after having seen 20 random training runs. The gating model even requires only 5 random training runs for this

accuracy. The models predict the rotation of the objects correct up to an average error of around 15° for the interaction, and less then 6° for the gating model.

Overall, the gating model outperforms the interaction state model by requiring less training runs while achieving better prediction results as soon as at least 2 training runs have been seen.

This performance difference is most likely due to the higher complexity of the Abstract Collection Selector (ACS) compared to the gating function. While the gating function only needs to learn a binary classifier, the Abstract Case Selector, needs to choose from a potentially big number of possible local models. Choosing an incorrect model usually results in an inaccurate prediction, which will be accumulated over the course of the test run. Furthermore, the gating function is provided with more information than the Abstract Case Selector, due to the features distance and closing, that are introduced in the relative interaction features.

The better performance of the interaction model after just one training run, is most likely due to the fact, that the gating function needs at least two runs before it works sufficiently well. An incorrect choice by the ACS which predicts a change in the object might still result in a better prediction than predicting no change at all.

The worse performance in predicting the actuator position of the interaction model was expected and is due to the problem of dependency within an interaction state, as already mentioned in section 3.2.4. In the interaction model, the actuator is not predicted directly, but only in combination with the block. That way the actuator's prediction is not only inaccurate if the regression model is not sufficiently trained, but also when the ACS chooses an incorrect AC.

The dedicated regression model in the object state concept does not suffer from these problems and can learn the direct relation between an action primitive and the resulting change in the actuator's state. Furthermore, the evaluation setting makes it very easy for the dedicated forward model since only a constant input is used in order to predict an output that is constant up to the noise explained in section 5.1.2. This also shows in the fact that only the minimum number of two nodes are learned in the forward model for the actuator. The fact that the prediction performance of the local model is as good as it is indicates that the additional input constraint in the AITM is working well to filter out the noise. Adapting the winning node in the case that the predicted output is not good enough when the given input is too close to the winning node, reduces the number of required nodes while also averaging out noise in the data.

The learned number of nodes is concerning when considering lifelong learning. Especially the number of nodes required in the ACS and in the object specific AITM show a concerning trend of scaling linearly with the number of interactions. While the model managed to

finish almost all[1] update and query calls during the experiments within the allowed time of 0.02 seconds, increasing numbers of nodes are bound to impact the models performance in the long run. The intermediate solution would be to implement the models in a more efficient language as well as perform optimizations for the involved computations. To solve this problem, the generalization performance of the AITM would need to be improved or different regression methods would need to be used.

6.1.2. Generalization on selected training data

The evaluations with selected training runs highlight to what extend the proposed models are capable of generalization. Obviously for memory based approaches, the best performance is achieved close to the known positions. However, despite not having seen the situations with the biggest change in orientation, the models are able to interpolate to some extend as can be seen in figure 5.7.

Overall the gating model performs better than the interaction state model, which is most likely attributed to the already mentioned discrepancy in complexity between the gating function and the ACS. However, around the edges of the object, the interaction state model makes better predictions about the orientation. It turns out, that the gating model does not make predictions for the outer testing positions. In this case, the gating function is not sufficiently trained with the provided data. Despite that, the gating function is able to generalize to unseen situations where the object is not influenced by the actuator. Although the ACS does know at least one AC where only the actuator is moving, it is not able to learn the correct selection on the provided training samples. Adding another training run, that passes the object on one side results in correct predictions for the testing runs on that side, but usually not on the other one.

This behavior is to be expected since the memory-based classifier works on similarities rather than correlations. In order to make a prediction, the two most similar known cases are consulted. Since no feature preprocessing or adapted metric is used, the distance in the classifier's input space determines the prediction. The additional features of distance and closing appear to provide enough additional information to the gating model to be able to generalize beyond the distance in the 2d space of the objects' position. Without these two features, the gating concept performs similar to the interaction concept. However, simply adding equivalent features to the interaction state model is not guaranteed to produce better results as it further increases the dimensionality of the feature vector and thus of the input space the regression model needs to learn. Furthermore, the computation of

[1] During the experiment, some runs took slightly longer than 0.02 seconds due to fluctuations in the systems load. However, the number of these cases was far below 0.1% in all experiments and the introduced inaccuracy in the data was considered small additional noise.

these two features requires knowledge about the object's shape which needs to be provided to the gating model, but is not required for the interaction state model.

It is interesting, that both models require less nodes in their AITMs when being provided with more structured training data when compared to the random training runs. The most reasonable explanation for this phenomenon is that the selected training positions result in less dynamic interactions. For example, the positions that produce the biggest change in orientation are not included in these training runs, but are likely to have appeared in random training example.

6.1.3. Generalization with constant feedback

The images in figure 5.8 showcase the one-shot learning capabilities of the memory-based regression and classification models. While both models lag one timestep behind in the first run, they predict changes in the actuator in the second run correctly. The predictions for the block are already quite good in the first run because the models reach the block in the predictions only after they were already updated by the actual interaction. Despite the fact that the rotations are predicted one frame later then in reality due to the late start for the actuator, the actual block predictions are accurate.

The interactions in the 2nd row of the figure are interesting. Since the actuator predictions for both models are accurate, the models have never seen the interaction at the right side of the block before it comes to predicting it. Therefore, it predicts the rotation from the situation that is closest to the current situation. In this case, this results in the wrong prediction for orientation. Furthermore, an impossible configuration of actuator and block is predicted where the actuator is located within the block. The models do not know about physical laws and perform no checks for such situations. However, in the interaction state model the actuator is predicted to be slightly further outside, resulting in less overlap with the block prediction. In the next frames, the models adapt their forward models to the new information that is being provided and start rotating the predicted block in the other direction. The gating model cannot completely compensate the relatively big error. This is because the gating function stops to predict an influence of the predicted actuator on the predicted block. In the interaction state model, the predicted block keeps turning for additional update steps before the ACS starts selecting an AC that does not change the block anymore. While the final prediction of the interaction state model might be closer to the actual block, the gating model managed to learn the interaction more precisely. This is because the gating function correctly predicted that the predicted actuator should not influence the predicted block anymore in the rightmost image of the second row. The already mentioned difference in complexity between the gating function and the ACS is also apparent in this evaluation.

Nevertheless, the images show the one-shot capabilities one would expect from memory-based approaches for both models. If a higher update rate is used, such as 100Hz as in the other experiments, smaller prediction errors will be made. In that case the currently predicted situation remains close to the actual situation, which means that the models can still apply what they have just learned for its next prediction.

6.1.4. Extension two multiple objects

Normal environments contain far more than only a single object. While the robot might perform a detection of a single salient object [48], it generally should be able to deal with multiple objects in its environment. The focus of this thesis was to incrementally learn the interaction with one object, however the results in section 5.2.5 show that the developed object state with gating function model is capable of dealing with multiple objects without being adapted. While the interaction state model suffers from the decision problem between multiple alternative predictions as discussed in section 3.2.4, the gating model does not need to be adapted in order to successfully make predictions for different types of objects. Since the performance of the blue block is not influenced by adding another object to the environment it is likely, that the gating model can deal with an arbitrary number of objects.

However, the results in figure 5.9 indicate a problem with the gating function. More precisely with the AITM when it is used for (binary) classification. The AITM relies on the output error when deciding to add a new node. In the binary classification case, this output will always only be one of two values (0 or 1). The result of that is, that the first nodes that are inserted as prototypes for either case are used to cover a large input space. If not sufficient amounts of diverse training samples are seen, this will lead to poor performances. Ideally, the node deletion scheme would remove nodes far away from the decision border over time, but this does not seem to be the case in this scenario. Adding the seventh training run in between both objects introduced an additional required node in the gating function, which greatly improves the prediction performance for the red block as can be seen in figure 5.10.

Unfortunately, the object state concept is not able to make predictions about object-object interactions. Although, the general idea for object-object interactions is explained in the concept's description (see section 3.3.1), the problem of determining the correct causal relation in order to train an appropriate gating function was not solved over the course of this thesis.

6.2. Reaching target states

The second task was designed in order to evaluate the learned inverse models of the concepts. Both concepts essentially use the same underlying inverse model, which was designed for the challenges that the incremental learning approach entails. For that reason, this section can also be understood as a discussion of the developed inverse model.

The inverse model itself is not provided with any domain knowledge, but does make the assumption about averaging feature dimensions. However, it was necessary to provide the models with information about what some feature dimensions represent in order to successfully reach a given target configuration. Furthermore, it was required to provide a circling action in order to avoid moving blocks arbitrarily.

Both models reach the target configuration most of the time. Target 1 appears to be more difficult for both models compared to the other targets, since it is not always reached and the average number of required steps is significantly higher for the first target than for the other targets. Looking at the test runs for Target 1 in more detail in figure 5.15 revealed that the AIM often selected suboptimal preconditions in the cases, where the target was not, or just barely reached. Two possible reasons for this poor performance come to mind:

1. The merging process produces suboptimal alternatives within the nodes.
2. The network selects the worse of two provided alternatives.

1) Due to numerical inaccuracy as well as noise, some suboptimal combinations can become dominant within a prototype. In this case, the optimal combinations might be lost during the merging process. The likelihood for this is reduced by the stricter merging process described in section 4.4.1, but not completely removed. The mentioned problem of "loosing zero dimensions" appears to be part of the problem in the third and fifth runs shown in figure 5.15. After the difference in orientation has been successfully reduced by the model, it tries to reduce the positional distance. In order to do that the model would need to push the block straight in the center, i.e. the preconditions should indicate the actuators relative position to be around 0, but apparently it pushes slightly away from the center which results in a new difference in orientation. This indicates that the optimal combination was lost in the merging process. The shown run 4 does not suffer from this problem as it manages to reduce the positional distance without impacting the orientation.

2) The greedy strategy of the network compares the two alternative preconditions for the currently biggest feature dimension to the alternatives of the second biggest feature dimension. The alternative that is closest to either of the second alternatives is chosen. This decision was motivated by the idea of planning ahead for the next feature dimension in order to avoid unnecessary circling movements around the objects.

In case some of these alternative are suboptimal (e.g. due to the first problem), the network might choose a bad precondition, even if the other alternative would have been optimal. That is because the suboptimal preconditions might be closer to the preconditions for the second biggest feature dimension. Furthermore, not all feature dimensions in the precondition should be considered when looking at closeness. For example, the preconditions for both models contain the relative actuator velocity (in the interaction state model this is implicitly given in the included action primitive). Different interactions require different velocities in order to produce the desired changes in the object's state. However, since the velocity of the actuator can be changed directly, it should not be considered when trying to choose the precondition that is closest to the preconditions required for the next feature. Preprocessed features or a fine tuned distance metric for preconditions would certainly reduce this problem, but is not available in the given setting as this would provide additional prior knowledge.

Figure 5.15 points out another problem with the used strategy of the network: In order to avoid oscillating between different feature dimensions, the one with the biggest difference is initially chosen and reduced until its sign changes or until the difference is considered small enough. While this does successfully avoid oscillations, it makes the model more vulnerable to errors in the merging process. As can be seen in the figure, the model creates bigger differences in orientation than what was given in the starting configuration while trying to reduce the difference in position. This indicates that the network should pay more attention to the development of other feature dimensions while reducing one.

Another thing to note is that the interaction concept generally requires more steps in order to reach the target than the object state concept. Careful analysis of the testing runs revealed that the used circling decision plays a big part in this. The decision, if the model should employ a circling action or not, depends on the distance to the location determined by the preconditions. In case the actuator simply needs to move to the other side of a corner, no circling action is employed but the actuator is moved directly in the direction towards the location. In this case the actuator would push at the corner of the object. While the gating model uses the gating function in order to determine if this action is safe in the sense that it does not move the object, the interaction state model does not have such a feature. Therefore, the block is often moved in an undesired way which needs to be corrected by additional interactions.

One problem that did not manifest itself visibly in the evaluation lies in the use of circular features such as the orientation. The angle of orientation is given in the range of $[-\pi, \pi]$. The model does not know that turning above π results in negative orientations. It also does not know that the orientation 3.14 and -3.14 are actually very close. When computing the difference vector $\vec{d}$, such a scenario will result in a big difference which the model tries to reduce. In most cases by turning the object essentially by almost 360°. This

problem can only be overcome by providing additional information about the feature dimensions or preprocessing features in a way that differences behave the same for all feature dimensions.

On the positive side, the proposed inverse model does not suffer from the general problem of memory-based approaches of becoming computational expensive the more training data they receive. This is because the AIM stores only a nearly constant amount of values in order to compute the averages for each feature dimension. The number of possible sign combinations that are stored separately is limited by the size of the superset of sign combinations. Furthermore, the inverse model successfully allows to deduce action primitive suitable to reduce the distance to a given state. This is also true for distances far greater than anything the model has seen during training. In that sense, the proposed inverse model is capable of enormous extrapolation.

The inverse model does not provide an action plan to reach a given target. Actual action sequences would need to be planned by higher level components of the robot. However, the provided forward and inverse model should provide the required tools in order to formulate plans successfully. In a sense, the models currently do just that by analyzing the returned preconditions, defining intermediate targets for the actuator and computing a path (circling) in order to reach these intermediate targets without collision. Due to the interactive nature of this setting, these plans are never kept for more than a single update step but rather recomputed each time an action primitive is queried in order to adapt to new situations.

6.3. The Adapted Instantaneous Topological Mapping

Both developed implementations use the same underlying regression and classification model in the form of the AITM. Being an adaptation of the GNG and similar in its output to a k-NN it does suffer from the typical problem of memory-based approaches when it comes to its dependence on the used metric and unnormalized features.

Both implementations do not really adapt the created nodes within the network, but rather rely on the topological mapping for the most part. The only adaptation they use is in the additional check of the input distance when deciding to insert a new node. The main reason for not using learning rates to adapt already learned nodes, is that the learning rates would need to be fine tuned to the given situation. Since the idea of this thesis is to develop and test models that adapt to unknown situations, reducing the number of tuned meta parameters is desired. In fact, having to choose the parameters listed in table 5.3 is already questionable when considering the original goal of autonomous self adaption.

Another reason for not using learning rates, especially for the matrix A, is that it can require a lot of training iterations before a stable matrix is found. In order not to influence early predictions, the matrix needs to be initialized as a zero matrix of suitable dimensions. Depending on the learning rate and the changes in the data, updating a zero matrix can easily lead to instabilities. Since the models constantly rely on the regression method, it is better to use a constant output per node instead of adding noise through an incorrect linear interpolation. Especially, since the method already interpolates between the output of two nodes. In scenarios where more training data and more prior knowledge is available the amount of required nodes is likely to decrease significantly when using positive learning rates.

The presented results of the required nodes indicate, that the AITM in its current form does not generalize well enough for lifelong learning. Ideally the relative number of required nodes should decline with increasing amounts of training data, however this was not the case at least for the first 30 training runs. In order to reduce the number of required nodes, the area of influence of each node needs to be increased. While the linear interpolation matrix A was designed to do that, the problem of reliably training this matrix incrementally with limited amounts of data remains. While outside the scope of this thesis, learning rates that are adapted automatically to the given data might provide a solution to this problem.

Apart from that, the experiment using two objects highlighted some problems with the AITM when used for classification. The dependence on the output alone without adapting the underlying input nodes can result to poor generalization. This indicates that the current update strategy for the AITM should be improved in the future.

Nevertheless, combined with the proposed models the AITM is able to make fairly accurate predictions about a complex environment without preprocessing any features or using special distance metrics. Furthermore, the method was also used successfully as a classifier in both the binary and the general case. Only the output interpolation needs to be turned off in order for the identical method to work for classification instead of regression. Considering the initial goal of self adaptation to unknown environments, such a universal method is very useful. Since the decision if output interpolation should be used or not can be changed at any time by the model, it is theoretically possible to make that decision automatically.

6.4. Concept comparison

This thesis provides two different concepts in order to incrementally learn simple object manipulations. The main differences between the concepts are summarized in table 6.1.

	Pairwise interaction	Object state
World representation	Pairwise interaction states	Individual object states
Actuator representation	Only part of an interaction state	Explicitly modeled
Action primitive	Influence any interaction state	Influences the actuator
Subspace creation	Changing feature sets	Object groups
Interaction separation	None	Gating function
Prediction	Simultaneously	Subsequently starting at actuator

Table 6.1.: Summary of the main differences of the two developed concepts.

Both concepts use the same memory-based regression and classification methods for the actual predictions and mainly differ in the way represent their environment. The pairwise interaction concept uses pairwise interaction states between two objects in order to represent both objects together. The object state with gating function concept on the other hand represents the objects individually and uses these to compute pairwise features when required.

Both concepts split the state space they encounter into local subspaces. While this is biologically inspired [26], the main reason was to reduce the burden on the trained regression models and allow for quicker update and query times.

The gating concept uses a gating function in order to differentiate between interactions that influence another object and those that do not. This greatly reduces the complexity of the local models for each object group. The findings of Johansson et al. also provide some evidence for a similar concept in humans, by highlighting the importance of contact events [25]. The predictions of the gating function can be interpreted as contact predictions in this context.

The gating concept is tailored more specifically to the given situation of object manipulation by explicitly representing the actuator. The interaction state concept on the other hand does not even know of different objects for prediction. This knowledge needs to be provided when trying to reach target states.

In that sense, the interaction state concept is better suited for the initial problem of automatically adapting to unknown environments because it makes less assumptions in its structure about the possible environments. However, the gating concept's assumption about separating the actuator explicitly might be reasonable in the context of robotics, where the robot should at least be provided with basic knowledge about itself.

The results in the previous chapter show that both concepts were able to learn about the dynamics of a simple unknown environment without additional information provided by humans, which was the main goal of this thesis.

Chapter 7

Conclusion

The initial goal of this thesis was the following: Provide simple (memory-based) models that

1. Update themselves incrementally during the interaction
2. Allow prediction of simple object interactions
3. Allow the deduction of action primitives required to reach a given target

in the context of simple object interactions.

In order to meet these goals, the two concepts described in chapter 3 were developed and implemented as described in chapter 4. The first requirement was enforced by the given framework which requires the implementations to be updated and queried quickly. Furthermore, the open loop evaluation in chapter 5 showed one-shot learning capabilities of the underlying regression and classification models. The suggested adaptation of the well known GNG (see section 4.4.2) was successfully used for both regression and classification tasks.

In the Push Task Simulation it was shown, that the second requirement was at least partly fulfilled since prediction up to 150 steps into the future were possible without preprocessing the used features, or using domain specific metrics. Both models successfully predicted future positions and orientations with acceptable precision. Furthermore, the evaluations with predefined training samples have shown that even better prediction performances can be achieved when only a few characteristic training samples are provided.

The results in the Move to Target task show that the third requirement was also fulfilled by reaching different provided targets reliably as long as all possible interaction types have been experienced. The results further show the successful application of the developed AIM described in section 3.4 which allows extrapolation in this dynamic setting. By not

only predicting and reaching positions but also orientations, these concepts outperform the work in [38] at least for simple objects while requiring less training data.

The memory-based approach does suffer from the fact that generalization is only possible in the form of interpolating between experiences. Extrapolation is not possible as the most similar experiences are always used to generate predictions. Furthermore, the interaction state concept cannot deal with multiple objects as it is unclear how to extract an object state from multiple predicted interaction states. The gating concept on the other hand has been successfully used in situations with two objects as long as no object-object interactions need to be predicted.

Overall, this thesis has demonstrated that

- The developed memory-based concepts can be used to incrementally learn about and interact in simple dynamic environments.
- The proposed topological approach can be successfully used for regression and classification tasks without preprocessed features or fine tuned metrics.
- The developed inverse model can be used in interactive settings to retrieve action primitives for extrapolated target configuration.

7.1. Future work

Obviously, the proposed concepts and implementations offer several ways of improvement:

- The developed AITM as well as the AIM are heavily affected by the feature quality. While no additional prior knowledge or preprocessing should be used, it should be possible to extend the developed methods to automatically adapt their distance metric as done in the LVQ [15].
- Alternatively, as already mentioned in the concept of the interaction state model, local optimization such as local feature selection could be employed.
- The developed inverse model sometimes selects suboptimal preconditions, which could be improved using more sophisticated merging strategies.
- The AIM could be extended to optimize multiple feature dimensions synchronously instead of one after another.
- The amount of feature knowledge required to reach a target could be reduced by learning the relations between the different feature dimensions automatically.

- Currently, the models are trained by providing action primitives to try from the outside. The information in the inverse model can be used as a rough estimate for the knowledge the model has already acquired about its environment. By extending on this, potential new prototypes could be evaluated and their preconditions first estimated and then tested in order to provide exploration capabilities to the existing concepts.
- The amount of required meta parameters should be reduced, especially domain specific parameters such as ϵ_{max}. A possible solution might be to use a similar strategy as in the AITM where the order of magnitude of the current output vector is used in order to determine a suitable threshold.
- The models could be extended to work in environments with multiple objects including object-object interactions. For this the problems mentioned in sections 3.2.4 and 3.3.3 need to be addressed.
- The ideas of both models could be combined, e.g. by adding a gating function to the interaction state model in order to reduce the amount of possible Abstract Collections.

The most promising direction for future research regarding the goal of autonomous adaption to unknown environments is the exploration of automatic metric learning. If the used distance metric could be adapted, redundant feature dimensions could be removed automatically. This would allow the use of feature vectors of higher dimensionality making the proposed concepts more suitable to real world tasks and more complex systems.

Chapter 8

Bibliography

[1] L. Liao, Y. Zhang, S. J. Maybank, Z. Liu, and X. Liu, "Image recognition via two-dimensional random projection and nearest constrained subspace," Journal of Visual Communication and Image Representation, vol. 25, no. 5, pp. 1187 – 1198, 2014.

[2] S. Levine and P. Abbeel, "Learning neural network policies with guided policy search under unknown dynamics," in Advances in Neural Information Processing Systems 27 (Z. Ghahramani, M. Welling, C. Cortes, N. Lawrence, and K. Weinberger, eds.), pp. 1071–1079, Curran Associates, Inc., 2014.

[3] C. M. Bishop, Pattern Recognition and Machine Learning (Information Science and Statistics). Secaucus, NJ, USA: Springer-Verlag New York, Inc., 2006.

[4] S. Grossberg, "Nonlinear neural networks: Principles, mechanisms, and architectures," Neural Networks, vol. 1, no. 1, pp. 17 – 61, 1988.

[5] I. J. Goodfellow, M. Mirza, D. Xiao, A. Courville, and Y. Bengio, "An Empirical Investigation of Catastrophic Forgetting in Gradient-Based Neural Networks," ArXiv e-prints, Dec. 2013.

[6] Y. Bengio, "Learning deep architectures for AI," Found. Trends Mach. Learn., vol. 2, pp. 1–127, Jan. 2009.

[7] I. Arel, D. Rose, and T. Karnowski, "Deep machine learning - a new frontier in artificial intelligence research [research frontier]," Computational Intelligence Magazine, IEEE, vol. 5, pp. 13–18, Nov 2010.

[8] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," in Advances in Neural Information Processing Systems 25 (F. Pereira, C. Burges, L. Bottou, and K. Weinberger, eds.), pp. 1097–1105, Curran Associates, Inc., 2012.

[9] D. Silver, Q. Yang, and L. Li, "Lifelong machine learning systems: Beyond learning algorithms," 2013.

[10] A. Carlevarino, R. Martinotti, G. Metta, and G. Sandini, "An incremental growing neural network and its application to robot control," in Neural Networks, 2000. IJCNN 2000, Proceedings of the IEEE-INNS-ENNS International Joint Conference on, vol. 5, pp. 323–328 vol.5, 2000.

[11] G. A. Seber and A. J. Lee, Linear regression analysis, vol. 936. John Wiley & Sons, 2012.

[12] D. Kibler and D. W. Aha, "Learning representative exemplars of concepts: an initial case study," in Proceedings of the Fourth International Workshop on Machine Learning (P. Langley, ed.), (San Mateo, CA), pp. 24–30, Morgan Kaufmann, 1987.

[13] C. Domeniconi, D. Gunopulos, and J. Peng, "Large margin nearest neighbor classifiers," Neural Networks, IEEE Transactions on, vol. 16, pp. 899–909, July 2005.

[14] N. Shental, T. Hertz, D. Weinshall, and M. Pavel, "Adjustment learning and relevant component analysis," in Computer Vision — ECCV 2002 (A. Heyden, G. Sparr, M. Nielsen, and P. Johansen, eds.), vol. 2353 of Lecture Notes in Computer Science, pp. 776–790, Springer Berlin Heidelberg, 2002.

[15] B. Hammer and T. Villmann, "Generalized relevance learning vector quantization," Neural Networks, vol. 15, no. 8–9, pp. 1059 – 1068, 2002.

[16] S. Geva and J. Sitte, "Adaptive nearest neighbor pattern classification," Neural Networks, IEEE Transactions on, vol. 2, pp. 318–322, Mar 1991.

[17] H. Ritter, T. Martinetz, and K. Schulten, Neural Computation and Self-Organizing Maps; An Introduction. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1st ed., 1992.

[18] E. Tulving, "Episodic memory: From mind to brain," Annual Review of Psychology, vol. 53, no. 1, pp. 1–25, 2002. PMID: 11752477.

[19] R. Polikar and C. Alippi, "Guest editorial learning in nonstationary and evolving environments," Neural Networks and Learning Systems, IEEE Transactions on, vol. 25, pp. 9–11, Jan 2014.

[20] C. Diehl and G. Cauwenberghs, "SVM incremental learning, adaptation and optimization," in Neural Networks, 2003. Proceedings of the International Joint Conference on, vol. 4, pp. 2685–2690 vol.4, July 2003.

[21] V. Losing, B. Hammer, and H. Wersing, "Interactive online learning for obstacle classification on a mobile robot," in Neural Networks (IJCNN), 2015 International Joint Conference on, pp. 1–8, July 2015.

[22] D. Liu, M. Cong, Y. Du, and X. Han, "Robotic cognitive behavior control based on biology-inspired episodic memory," in Robotics and Automation (ICRA), 2015 IEEE International Conference on, pp. 5054–5060, May 2015.

[23] D. Liu, M. Cong, X. Han, and Y. Du, "Robotic episodes learning for building cognitive experience map," in Cyber Technology in Automation, Control, and Intelligent Systems (CYBER), 2015 IEEE International Conference on, pp. 1210–1215, June 2015.

[24] J. R. Flanagan, M. C. Bowman, and R. S. Johansson, "Control strategies in object manipulation tasks," Current Opinion in Neurobiology, vol. 16, no. 6, pp. 650 – 659, 2006. Motor systems / Neurobiology of behaviour.

[25] R. S. Johansson, G. Westling, A. Bäckström, and J. R. Flanagan, "Eye–hand coordination in object manipulation," the Journal of Neuroscience, vol. 21, no. 17, pp. 6917–6932, 2001.

[26] M. Kawato, "Internal models for motor control and trajectory planning," Current Opinion in Neurobiology, vol. 9, no. 6, pp. 718 – 727, 1999.

[27] J. R. Flanagan, S. King, D. M. Wolpert, and R. S. Johansson, "Sensorimotor prediction and memory in object manipulation.," Canadian Journal of Experimental Psychology/Revue canadienne de psychologie expérimentale, vol. 55, no. 2, p. 87, 2001.

[28] D. M. Merfeld, L. Zupan, and R. J. Peterka, "Humans use internal models to estimate gravity and linear acceleration," Nature, vol. 398, pp. 615–618, Apr 1999.

[29] A. Bicchi and V. Kumar, "Robotic grasping and contact: a review," in Robotics and Automation, 2000. Proceedings. ICRA '00. IEEE International Conference on, vol. 1, pp. 348–353 vol.1, 2000.

[30] A. Saxena, J. Driemeyer, and A. Y. Ng, "Robotic grasping of novel objects using vision," The International Journal of Robotics Research, vol. 27, no. 2, pp. 157–173, 2008.

[31] I. Lenz, H. Lee, and A. Saxena, "Deep learning for detecting robotic grasps," The International Journal of Robotics Research, vol. 34, no. 4-5, pp. 705–724, 2015.

[32] M. P. Deisenroth, G. Neumann, and J. Peters, "A survey on policy search for robotics," Found. Trends Robot, vol. 2, pp. 1–142, Aug. 2013.

[33] S. Levine, N. Wagener, and P. Abbeel, "Learning contact-rich manipulation skills with guided policy search," CoRR, vol. abs/1501.05611, 2015.

[34] S. Nishide, T. Ogata, J. Tani, K. Komatani, and H. G. Okuno, "Predicting object dynamics from visual images through active sensing experiences," Advanced Robotics, vol. 22, no. 5, pp. 527–546, 2008.

[35] B. Moldovan, P. Moreno, M. van Otterlo, J. Santos-Victor, and L. De Raedt, "Learning relational affordance models for robots in multi-object manipulation tasks," in Robotics and Automation (ICRA), 2012 IEEE International Conference on, pp. 4373–4378, May 2012.

[36] O. Kroemer and J. Peters, "Predicting object interactions from contact distributions," in Intelligent Robots and Systems (IROS 2014), 2014 IEEE/RSJ International Conference on, pp. 3361–3367, Sept 2014.

[37] J. J. Gibson, The Ecological Approach to Visual Perception: Classic Edition. Psychology Press, 2014.

[38] M. Lau, J. Mitani, and T. Igarashi, "Automatic learning of pushing strategy for delivery of irregular-shaped objects," in Robotics and Automation (ICRA), 2011 IEEE International Conference on, pp. 3733–3738, May 2011.

[39] J. Kolodner, "An introduction to case-based reasoning," Artificial Intelligence Review, vol. 6, no. 1, pp. 3–34, 1992.

[40] D. Landgrebe, "A survey of decision tree classifier methodology," Systems, Man and Cybernetics, IEEE Transactions on, vol. 21, pp. 660–674, May 1991.

[41] M. Leordeanu, M. Hebert, and R. Sukthankar, "Beyond local appearance: Category recognition from pairwise interactions of simple features," in Computer Vision and Pattern Recognition, 2007. CVPR '07. IEEE Conference on, pp. 1–8, June 2007.

[42] J. Jockusch, Exploration based on neural networks with applications in manipulator control. PhD thesis, Bielefeld University, 2000.

[43] B. Fritzke, "A growing neural gas network learns topologies," in Advances in Neural Information Processing Systems 7, pp. 625–632, MIT Press, 1995.

[44] M. D. Buhmann, "Radial basis functions," Acta Numerica, vol. 9, pp. 1–38, 1 2000.

[45] N. Koenig and A. Howard, "Design and use paradigms for gazebo, an open-source multi-robot simulator," in Intelligent Robots and Systems, 2004. (IROS 2004). Proceedings. 2004 IEEE/RSJ International Conference on, vol. 3, pp. 2149–2154 vol.3, Sept 2004.

[46] R. Smith, "Open dynamics engine," 2005.

[47] Google, "Protocol buffers." `https://developers.google.com/protocol-buffers/`, 2008.

[48] T. Liu, Z. Yuan, J. Sun, J. Wang, N. Zheng, X. Tang, and H.-Y. Shum, "Learning to detect a salient object," Pattern Analysis and Machine Intelligence, IEEE Transactions on, vol. 33, pp. 353–367, Feb 2011.

Acronyms

k-NN k-Nearest Neighbor. 5, 9, 86

AC Abstract Collection. 14–19, 30, 31, 33–36, 62, 65, 68, 80–82, 91

ACS Abstract Collection Selector. 14, 16, 17, 30, 31, 62, 65, 68, 80–82

AIM Averaging Inverse Model. 25, 26, 30, 35, 43, 62, 79, 84, 86, 89, 90

AITM Adapted Instantaneous Topological Map. 6, 29–31, 37, 50–52, 62, 65, 66, 79–83, 86, 87, 90, 91

BN Bayesian Network. 8

GNG Growing Neural Gas. 3, 6, 29, 50, 86, 89

ITM Instantaneous Topological Map. 50, 51, 53

LLM Local Linear Map. 6, 50, 51

LVQ Learning Vector Quantization. 6, 90

MDP Markov Decision Process. 7

NN Nearest Neighbor. 65

SVM Support Vector Machine. 6

Appendix A

Additional Information

A.1. Circling action

In order to provide action primitives that move the actuator around an object, a circling action was required. The computation of the circling action was designed based on the available features in the gating model. Therefore, the distance between the actuator and the object needed to be computed additionally for the interaction state model. The distance is used in order to be able to avoid the object without explicit knowledge about the object's shape. Some shape information is however required in order to compute the distance. The process is described in algorithm A.1:

Using the distance, the actuator can stay within a save distance of 0.04 m to 0.06 m of the object. Outside this area, the actuator moves straight towards or away from the center of the object. This ensures, that the actuator does not collide with the object while circling. Inside this area, the actuator uses one of the two tangents to the global direction from the actuator to the object. Which tangent to use is determined by the angles of the vectors between actuator-object and target-object with respect to the local x axis of the objects coordinate system. The direction, that reduces the difference the most is chosen. The direction is determined as shown in the function `ComputeTangent`.

Algorithm A.1 Pseudocode for computing a suitable circling action. The resulting tangent needs to be normalized according to the situation.

Input: Distance $dist$ between actuator and object
Input: Local direction $\vec{d}_{local}$ from actuator to object
Input: Global direction $\vec{d}_{global}$ from actuator to object
Input: Local direction $\vec{t}$ from target to object
Output: Velocity vector $\vec{v}$ that moves the actuator around the object towards the target.

```
1  function circling(dist, d⃗_local, d⃗_global, t⃗, v⃗)
2      if dist < 0.04 then
3          v⃗ ← −norm · d⃗_global
4      else if dist > 0.06 then
5          v⃗ ← norm · d⃗_global
6      else
7          v⃗ ← computeTangent( d⃗_global, d⃗_local, t⃗)
8      return v⃗

9  function computeTangent(d⃗_global, d⃗_local, t⃗)
10     tan^x ← −d^y_global
11     tan^y ← d^x_global
12     actAngle ← arctan2(d^y_local, d^x_local)
13     targetAngle ← arctan2(t⃗[1], t⃗[0])
14     angDif ← targetAngle + π - (actAngle + π)
15     if angDif > 0 then
16         if |angDif| < π then
17             return tan⃗
18         else
19             return −tan⃗
20     else
21         if |angDif| < π then
22             return −tan⃗
23         else
24             return tan⃗
```

Message	Field	Type
ActuatorCommand	cmd	Command (NOTHING, MOVE, GRAB, RELEASE)
	direction	Vector3d
ModelState	name	String
	id	uint32
	is_static	bool
	pose	Pose
	scale	Vector3d
	linVel	Vector3d
	angVel	Vector3d
	type	uint32
ModelState_V	models	[ModelState]
WorldState	model_v	ModelState_V
	contact	Contacts

Table A.1.: Summary of custom Protobuf messages including their fields.

A.2. Protobuf messages

The following Protobuf messages were defined in order to communicate with the simulation:

- ActuatorCommand: Allows to send commands for the actuator. Only moving is used for this thesis.
- ModelState: Provides information about a single object. Includes name,identifier, position, orientation, linear and angular velocity.
- ModelState_V: A list of ModelStates in order to transmit information about all objects at the same time
- WorldState: Includes a ModelState_V filled Modelstates for all objects and contacts information. The contact information is not used in the current setting though.

The included information of these messages is summarized in table A.1

Apart from these self defined messages, this thesis also uses some of the already predefined ones from gazebo such as Contacts, Pose and Vector3d. A complete overview of all available messages for the used version can be found at:
`http://osrf-distributions.s3.amazonaws.com/gazebo/msg-api/2.2.1/classes.html`
(last checked December 2015).

Appendix B

Statutory declaration

I declare that I have authored this thesis independently, that I have not used other than the declared sources / resources and that I have explicitly marked all material which has been quoted either literally or by content from the used sources.

Bielefeld, 30 November 2015

(Jan Pöppel)