Master Thesis

Bielefeld University

Faculty of Technology

Memory-based exploratory 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

September 2015

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

Referee: Dipl.-Inf. Maximilian Panzner

Abstract

Contents

Chapter 1

Introduction

After decades of research a vast amount of powerful tools have been developed for machine learning. A good overview can be found in the book by Bishop [**?**]. 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 way the researchers were able to create better classifiers for image recognition (e.g. [**?**]) and better controllers for complex movements (e.g.[**?**]).

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 because of the stability-plasticity-problem [**?**]. More precisely because of the phenomenon of catastrophic forgetting [GMX+13].

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 multi-purpose robots. However, the biggest 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, suitable tools cannot be training in advance. Furthermore, even if one would attempt to train specialized parts for all kinds of problems, acquiring sufficient and accurate training data beforehand is infeasible at best. Therefore, instead of trying to train the machine beforehand, it might be better to provide it with the means to adapt to new situations on its own while it is encountering them. Instead of learning on previously recorded training data, the robot adapts its model to the continuous stream of data while it is already using what it learned beforehand. 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 usually experienced. The usual approach is 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 on prior knowledge 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 it. For this thesis a memory based learning approach was chosen in order to solve these problems. Due to their one-shot learning ability, memory based methods can produce good prediction results from very little training data.

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. It is for these reasons object interactions make an ideal target for online self adaptation.

This thesis provides two concepts that provide incremental learning of pushing interactions between an actuator 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 primitives as been performed. The inverse model provides an action primitive that is used to reach a specified target configuration within the environment.

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

1. Update themselves incrementally during the interaction
2. Allow prediction and planning of simple object interactions
3. Allow lifelong learning

This thesis is structured as follows: In chapter 2 an overview over memory based learning as well as recently proposed models for learning object interactions for robotics is given. Two concepts were developed in order to fulfill the stated goals. Their general idea is described in 3 before concrete implementation details are given in chapter 4. These models are then evaluated with regards to their prediction and planning performance in chapter 5. Finally, this thesis concludes with a discussion of these results in chapter 6.

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 [**?**] where they 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. Maybe the most prominent example for this is the k-Nearest Neighbor (k-NN) regression [**?**]. 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 $N(\vec{x})$):

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

These instances are also often called prototypes, if only a few instances are used to represent a subspace in the input space. Since the closest instance is determined by similarity, the used features and metric is usually critical for these methods [**?**]. This dependence on careful preprocessing of the used features and metric can be considered the biggest disadvantage of these methods. However, there has also been a lot of research in automatically adapting the used metric while training in order to better fit the data, e.g. by Hammer

et al. [**?**]. Unfortunately, online metric adaptation often needs a lot of iterations before yielding satisfying results.

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 also lead to increased query and update times, since these scale linearly with the number of stored instances. Geva et al. [GS91] showed already 1991 that the number of required instances can be reduced depending on the used algorithm. Extending the area of influence of the given instances, for example by using Local Linear Maps (LLMs) [RMS92], 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 that memory based models usually do not suffer from the catastrophic forgetting phenomenon.

For the implementation of the concepts developed in this thesis, an adaptation of the Growing Neural Gas (GNG) using LLM as output function is used as underlying regression model. The Adapted Instantaneous Topological Map (AITM) is explained in section 4.4.2 in detail.

2.2. Learning object interactions

Because of the importance for robotics, a lot of research is being done regarding object interactions. However, most systems are trained in an offline fashion [NOT$^+$08, MMO$^+$12, KP14]. Out of these, the works of Moldovan et al. and Kroemer et al. are most notable: Moldovan et al. use a probabilistic approach to learn affordance models useful for object interactions in [MMO$^+$12]. Affordances, as introduced by Gibson [Gib14], 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 between contact distributions between objects in order to classify different kind of interactions in [KP14]. In their work an interaction is for example if one object supports another object or if an object can be lifted given a certain grasp configuration. The approach of Kroemer et al. also works with multiple objects since only the contact distributions between two objects were considered. Unfortunately, their approach is limited to binary

predictions of predefined interaction classes. The robot still needs to learn a suitable forward model. Furthermore, their proposed sampling approach showed poor performance and is likely to become infeasible in unconstrained complex environments.

Learning object interaction incrementally has also been done: In [LWA15] Levine et al. achieved great results in learning policies of complex dynamics such as assembling plastic toys. They require much fewer training iterations than earlier policy search approaches, i.e in [**?**]. 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.

The most similar work to this thesis both in terms of setting and in their approach has been done by Lau et al. in [LMI11]. 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, here the entire circular mobile robot, around or towards the object is not part of the learned model but provided instead. Furthermore, while they describe their approach to work with position and orientation, they only provide results for position.

Chapter 3

Concept

Instead of using only a single regression model that is constantly updated and queried, the robot needs some sort of architecture in order to deal with the continuous information stream. 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 learned. This chapter presents the two concepts that were developed in order to fulfill these tasks. 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, 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 model does not necessarily know the effects of the primitives. In this case, the model needs to not only learn about the pushing interactions, but also about the forward model of its own actions.

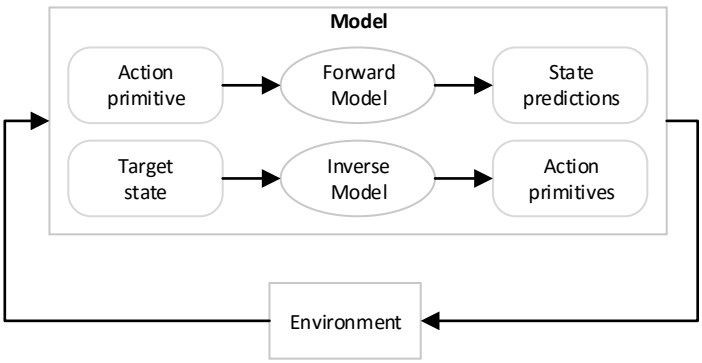

Figure 3.1.: Overview of the problem. The model includes a forward model in order to make predictions given the current environment state and a certain action primitive. When given a target state the inverse model provides suitable action primitives to reach the desired state.

Apart from the forward model, the robot is also 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 model must learn how it can influence the objects through its actuator. The target configurations will rarely be reachable within a single action primitive but the inverse model should 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.

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

3.2. Modeling pairwise interactions

The first approach tries to find distinct subspaces in the interaction space between two objects. In this context, the 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 objects 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.

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**. The general idea is to store these episodes as past knowledge similar to the approach in case based reasoning [Kol14]. 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:

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

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 $||a, b||_e$ 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 planning. 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 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 episodes.

As mentioned above, the idea of this concept is to split the interaction space into subspaces and train local models for each of these subspaces. 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**. Each collection trains their own local forward and inverse model which is explained in section 3.2.1. An **Abstract Collection Selector** is needed in order to choose the most appropriate abstract collection in a given situation.

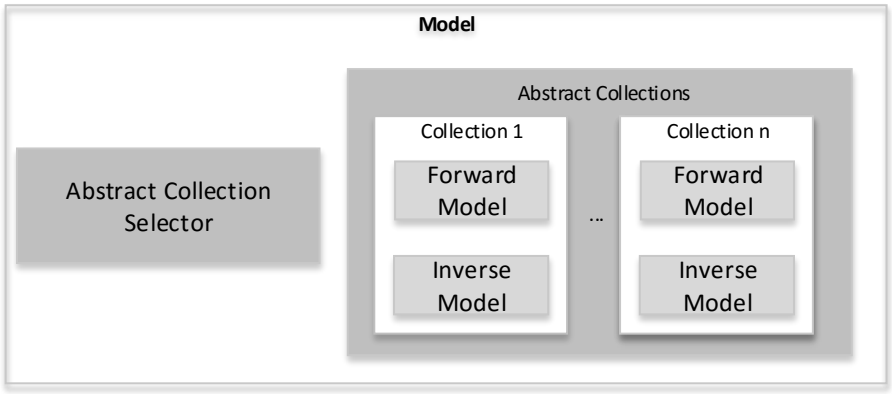

Figure 3.2.: Overview of the model in interaction space. The different abstract collections contain forward and inverse models of distinct subspaces of the interaction space. The selector is needed in order to select the correct submodel for prediction.

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, mainly the position of an object changes in the pushing subspace. However, in order to separate the episodes like that, accurate labels are required. Such a classification is not possible 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.

The clustering of episodes suggested here is based on the set of features that changed between the previous state and the resulting state while 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, $Pre(f), Post(f)$, 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.

Each different set is represented by its own abstract collection AC_i. The idea is that, while not necessarily holding semantic meaning directly, these collections correspond to different interaction scenarios. Depending on the features used, some abstract collections 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 abstract collections can be created: No feature changes, either one of the two features changes and both change at the same time. In this example the abstract collection 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 abstract collections 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 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. Prediction with the Abstract Collection Selector

Prediction within this architecture requires several steps. 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. The selector needs to estimate which Abstract Collection is most likely responsible for the next interaction. In order to make this estimation, any suitable classifier, e.g. a decision tree [**?**], 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 and only those collections are considered, that are already made up of more than ϵ_{min} training examples. This threshold is used to reduce the number of outliers when training the classifier.

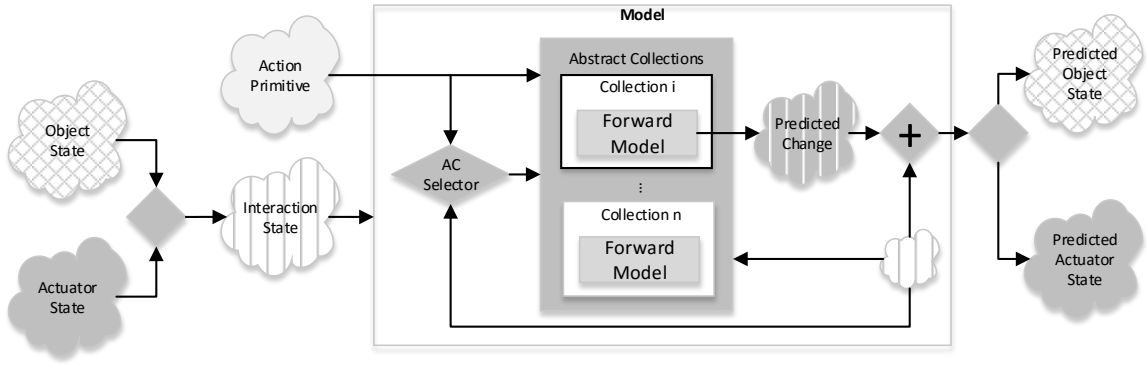

Figure 3.3.: Concept of prediction in the pairwise interaction space with Abstract Collections. First the interaction state is computed. This state is then fed to the model together with the chosen action primitive. The AC Selector uses those to select the responsible Abstract Collection. The collection 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. The actual predicted object states are finally extracted from this interaction state.

After the most likely Abstract Collection has been selected, its forward model can be consulted for the prediction. The local model will also be 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. These predicted changes are then added to features of the given interaction state this Abstract Collection is responsible for. The prediction of the actual object states need to be extracted from this interaction state by reverting the initial transformation.

3.2.3. Planning

In order to get a suitable action primitive given a target configuration, first the difference set between the current situation and the target configuration is computed. Afterwards, the Abstract Collection 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 Abstract Collection is the one that covers most of the changed features in the computed difference set. If multiple alternatives exist, the Abstract Collection responsible for the changes in the most features is chosen. This is because more changing features means that the inverse model has more degrees of freedom to reach the target.

The selected Abstract Collection queries its own inverse model for preconditions that produce a change in the direction of the desired target configuration. Since it might not be possible to move an object directly in the desired direction due to the current position of the actuator, simply returning the action primitive is not sufficient.

The preconditions need to be checked against the currently given configuration interaction state 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 where the preconditions are met.

3.2.4. Theoretical discussion

Using pairwise interaction states in order to represent the interactions between objects is often used in robotics, for example in [**?**]. 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. While this will only involve simple coordinate transformations in most cases, some object attributes might not be as easily transformable. Furthermore, in order to contain all the necessary information about both objects, these interaction states can become high dimensional.

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. Since 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.

An even bigger disadvantage is the fact that this approach can not 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 scene. 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. On top of that, the local models of the Abstract Collections need to be applicable to different objects, which makes them more complicated.

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 gate

The 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 3.4. 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 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 `Gate`. The Gate is used to split the interaction space in two big subspaces. The first subspace represents all scenarios where no actual pushing interactions are taking place. That means that at most the actuator is moving due to some action primitives but no other object is influenced by those movements. 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 they are actually changed. There is no need to train multiple models for each object since the local subspaces each model need to learn are already a lot simpler than the interaction space learned in the previous concept.

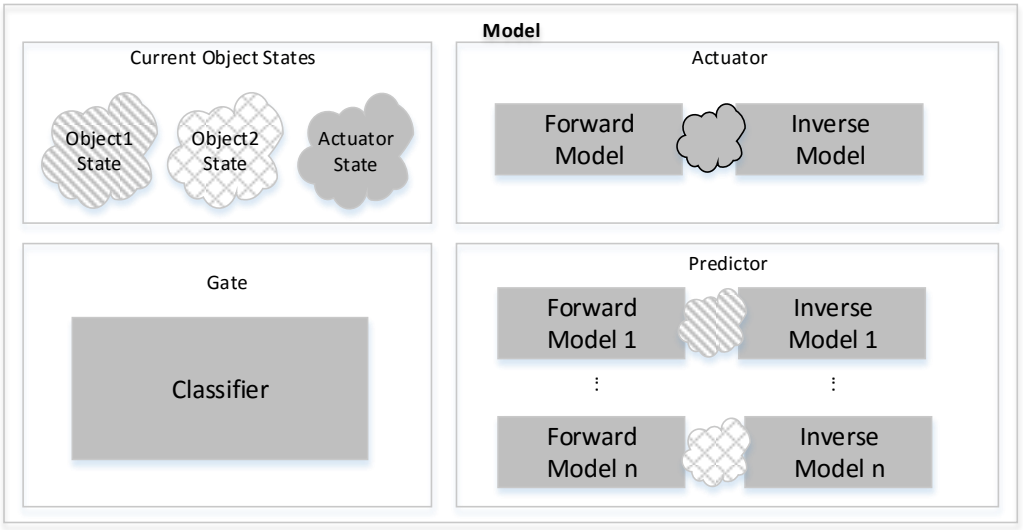

Figure 3.4.: Overview of the concept about object state with gate. The actuator is a special object, that can be influenced directly. It's forward and inverse model might even be provided if known. For the other objects/object groups, the Predictor learns these models. The Gate learns a classifier to distinguish interactions between objects from only single object changes.

In order to allow the local models to only train on this restricted subspace, 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 gate and, if required the actuator's local models are trained. On top of that, the local models responsible for any object whose state changed with the last update, are trained as well.

3.3.1. 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 in order to predict the next actuator state. The general process for predicting general objects is visualized in figure 3.5.

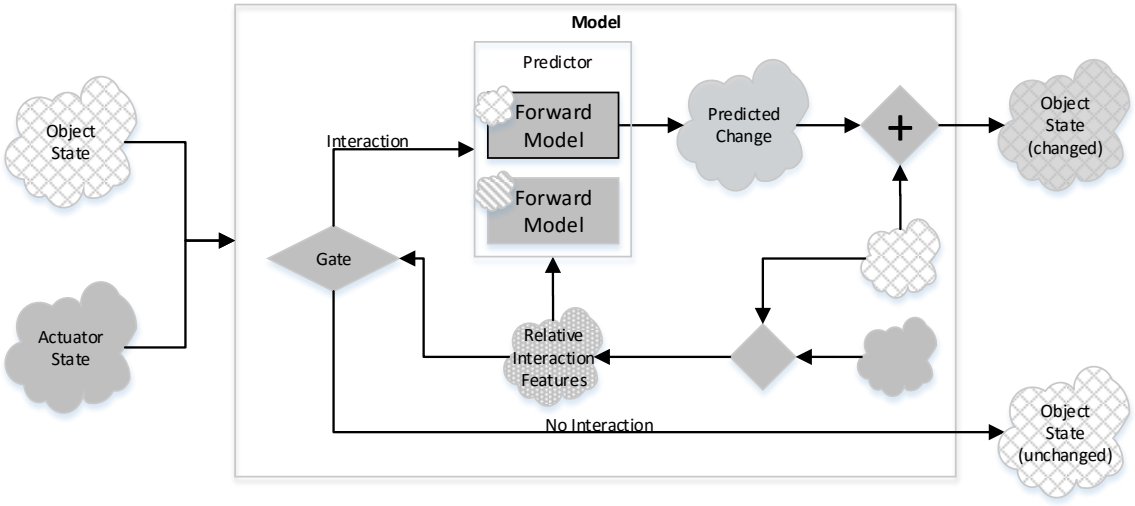

Figure 3.5.: Visualization of the prediction process for non actuator objects in the object space with gating concept. Using the predicted actuator state, relative interaction features are computed. These are used by the gate to determine if an interaction takes place or not. The responsible local forward model is used to predict changes if an interaction is expected. From these changes the actual state prediction is computed.

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 interaction state that was explained in the previous concept. However, since it is not necessary to extract the entire object states for both objects from these relative features, the dimensionality can be a lot lower than before. In fact, depending on the scenario, it may be sufficient to only represent the actuator in the local coordinate frame of the object. The Gate then uses these features to determine if the object will be influenced by the new actuator state. If the gate 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 order to get the actual predicted object state, these changes are added to the current object state.

When more then 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. These predictions can

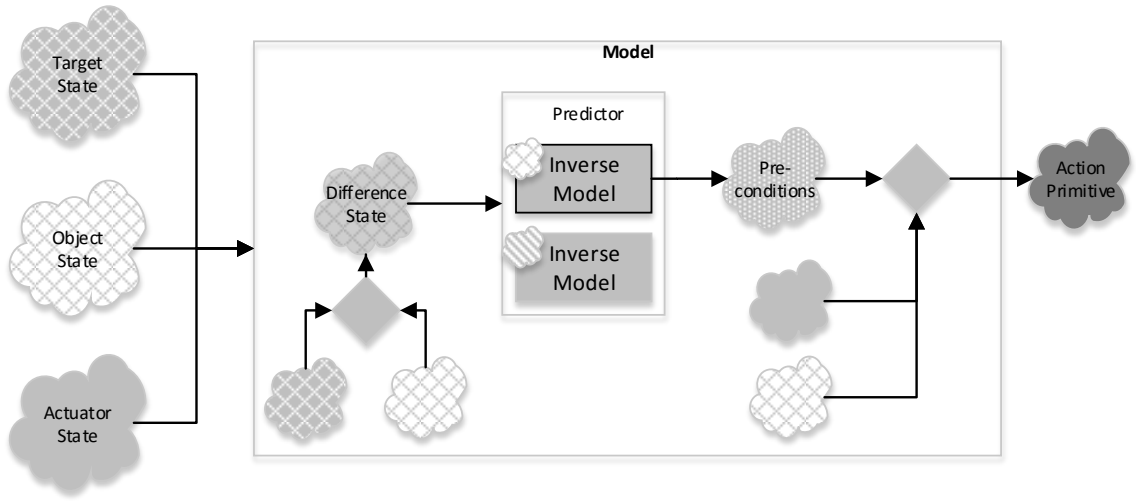

Figure 3.6.: Visualization of the planning process for non actuator objects in the object space with gating concept.

then in turn be used as new "actuators" for interactions between them and other objects.

3.3.2. Planning

Similar to prediction, planning is also 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.

The more interesting process of reaching a target configuration for a non actuator object is visualized in figure 3.6.

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 responsible inverse model. Since only the actuator can be influenced by actions 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 is changed towards 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 can be derived from the local features. In most situations this will however not be the case. 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 a intermediate target.

Considering the current object's position, an action is calculated to move the actuator towards the intermediate target. This might result in a circling movement of the actuator 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 to changes in the environment.

3.3.3. Theoretical discussion

Representing each object by itself has the advantage, that no transformations are required. Furthermore, no object depends on the accuracy of another object's state. However, the biggest disadvantage is that if the gate misclassifies then this concept can easily predict impossible configurations.

Apart from the representation, this concept requires some more assumptions about the scenario then the previous one. 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 actions.

3.4. Time invariant inverse model

Directly searching in the forward regression model for the inverse has several disadvantages in the given scenario: Firstly, the difference state that is computed can have features magnitudes 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. Secondly, 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 at the cost of an increase in another feature. 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.

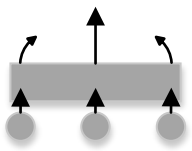

Figure 3.7.: Visualization of the precondition grouping in the proposed time invariant inverse model. All three pushing scenarios indicated by the three balls, will be grouped together since they all result in the same positional change.

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 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: It is called time invariant inverse model because, 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 of the changes. Furthermore, in order to avoid the problem of feature weighting feature importance, each feature is considered separately. Preconditions are grouped together according to the direction they change a certain feature in. An example of this grouping is visualized in figure 3.7.

The figure shows three different scenarios or preconditions in the form of the three balls that represent the actuator. In the forward model each precondition is tightly coupled to its resulting change. In this example the left situations results in exactly the small positional change upwards and a 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 present in multiple prototypes if it caused multiple features to change.

Each prototype creates a weighted average[1] of the preconditions for its feature and direction. The preconditions are weighted by their contribution towards the feature. 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 first reduce the feature with the biggest difference. The prototype for the selected feature and its direction is then queried for the 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

- Actual architectures resemble concepts closely
- Crux lies in the regression model (AITM)

In order to evaluate the developed ideas, prototypes of the two concepts described in the previous section have been implemented in Python. Both prototypes are developed as such that they can be evaluated in the same way in the next chapter. The implementations try to follow the presented concepts very closely. As such the general data flow is the same as was presented in the figures 3.3, 3.5 and 3.6. Relevant decisions and assumptions made specifically for the implementation are highlighted and explained in this chapter.

Section 4.1 starts with describing common design decisions made 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 such as 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 on what features are actually used. In fact the models do not require any knowledge about what the features represent for prediction and none is provided. 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 model does need to have some partial knowledge about the features when it comes to planning as will be further explained in the corresponding sections of 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 available at each update step from the environment in the given setting.

Furthermore, a unique identifier, the shape and size of the objects are known and can be used to compute additional features for the model. What kind of features are computed from this information is a design decision of the user of these models. Because of the dependence on the used metric in the underlying regression model, using more features might worsen the performance of the models.

4.1.2. Action primitives

In the scope of this thesis, the action primitives allow to set the two dimensional velocity of the actuator. Although, velocities are set, the models assume that an action is selected and performed at every iteration. 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. In this case a two dimensional vector is used representing the x- and y-velocity respectively. The velocities are given according to the global coordinate system of robot.

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 AITM, explained in section 4.4.2, is used for all regression and classification models used in both prototypes unless otherwise mentioned.

4.1.4. Using local features and predicting differences

Both prototypes only use local 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 model and 4.3.1 for the object state model. Using local features assumes that similar interactions behave identical regardless where in the environment they take place. 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 a constant effect on the objects. The environment used for the evaluations, described in section 5.1.1, fulfills these conditions since it only affects the objects through gravity and friction, which are constant throughout the entire environment.

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 differences can still be used. However, in this case, the used input features need to contain the information about the influence of the environment or the current object configuration. 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 also contains the components visualized in figure 3.2. From these components, three different parts are trained: The Abstract Collection Selector trains a classifier that selects any of the n learned Abstract Collections. Within each Abstract Collections, there exist a local forward model which performs the predictions. Additionally, the Abstract Collections contain an inverse model, which return 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 with the AITM. For the reasons explained in section 3.4, the special inverse model is used.

Algorithm 1 explains the steps that are performed each time new information is received from the environment.

```
Input: New worldstate ws
Input: Used action a
Data: Last worldstate ws_old
Data: Set of Abstract Collections L
Data: Abstract Collection Selector ACS
1  foreach interaction state i ∈ ws do
2      i_old = extractInteractionState(ws_old, i) newEpisode = Episode(i_old, a, i)
3      S = computeChangedFeatures(newEpisode)
       /* Check if a corresponding AC already known                    */
4      if S ∉ L then
5          newAC = AbstractCollection(S)
6          L = L ∪ newAC
7      end
8      AC = L_S                          /* The Abstract Collection responsible for S */
9      update AC with newEpisode
10     update ACS with i_old, a and AC
11 end
```

Algorithm 1: Prediction of the update steps in the pairwise interaction model.

The changed feature set S is basically 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.

When updating an Abstract Collection both its forward and inverse model need to be updated. The forward model as well as the inverse model are trained using the combination of the old interaction state i_{old} and the action primitive a as input and the difference vector d between the old interaction state and the new one as desired output. The exact training of the AITM is explained in section 4.4.2 while the training of the inverse model is explained in section 4.4.1.

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

When used for the prediction the model follows the process visualized in figure 3.3 and described in section 3.2.2. Planning 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.

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 are present in the scene which allows fairly simple interaction states as described in table 4.2.

Furthermore, the interaction state contains the information required for the transformations. Specifically, the matrix T 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.

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 world state is simply a collection of all interaction states in the environment. At each update from the environment, a new Worldstate is computed from the information provided. 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

[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.

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.: This table shows exemplary features used to represent one interaction state. Relative positions and velocities refer to the coordinate system of the reference object.

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.

The interaction states are computed as follows: In this case 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:

$$T = \begin{pmatrix} \cos(\alpha_1) & -\sin(\alpha_1) & px_1 \\ \sin(\alpha_) & \cos(\alpha_1) & py_1 \\ 0.0 & 0.0 & 1.0 \end{pmatrix} \qquad T^{-1} = inv(T) \tag{4.1}$$

px_1, py_1 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 Abstract Collections 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 analog if needed. In order to get the global orientation, the reference object's orientation simply needs to be added to the predicted orientation.

4.2.2. Episodes

Episodes are used to store past experiences. The idea comes from case-based-reasoning [Kol14] 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 Abstract Collections use these episode 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. This does however 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 Abstract Collections extract the features differences 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* for $Post^*$.

4.2.3. From target object state to action primitives(?)

[Subject to change]

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 set. The secondary object is the actuator however the actuator 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 difference vector 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.

Using this reduced difference vector, the model needs to find the Abstract Collection that is responsible for changes in the features of interest. Since these features are only a subset of the entire interaction state, there will usually not be an Abstract Collection that is responsible exactly for these features. Instead there will be several Abstract Collections, that are responsible for these features as well as other features.

The greedy strategy of the developed inverse model does not work well with the space segmentation performed by the Abstract Collections. Instead of training local inverse models in each Abstract Collection as described in the concept, this implementation only uses one global inverse model, that is trained with all experienced interactions. Since the proposed inverse model does not scale with the number of training examples, but is only influences by the number of possible sign combinations (see section 4.4.1 for details), this approach does not impact the model's performance. This global inverse 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 user defined interaction states provide the ability to compute an action primitive that lets the actuator circle around the reference object at a fixed distance. Using this circling action allows the actuator to move towards the desired position without influencing the object.

Once the distance is below a threshold of 0.1[2], the model assumes that the actuator is on the correct side of the object an can move directly towards the desired position. This can be done by simply following the direction between the target position and the current position.

Once the actuator has come close[3] 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.

[2]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 5.1.1.

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

4.3. Object space with gating function

The realization consists mainly of the parts visualized in figure 3.4. Depending on whether the actuator models are to be learned as well three to fife different parts need to trained in this model: The predictor takes on a similar role to the Abstract Collections 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. The gate 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 computed by computing the change between the previous object state and the current one.

Algorithm 2 summarizes all steps performed at each update step.

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

```
1  newActuator = extractActuator(ws)
2  foreach object state o ∈ ws do
3  |  o_old = extractObject(ws_old, o)
4  |  relFeatures = computeRelativeFeatures(o, newActuator)
5  |  change = computeLocalChange(o_old, o)
6  |  hasChanged = || change || > 0
7  |  updateGate(relFeatures, hasChanged)
8  |  if hasChanged then
9  |  |  updatePredictor(relFeatures, change)
10 |  end
11 end
12 updateActuator(newActuator, a)
```

Algorithm 2: Algorithm summarizing the steps performed by the object state model at each update from the environment.

Extracting the actuator or an object state is simply an attribute lookup in the worldstate. The changes are computed by first transforming the new object state to the coordinate frame of the old object state before taking the difference of both vectors.

When the object state has changed, the predictor is updated with the relative interaction features as input and the change as output. The predictor can determine the object by

the identifier in the relative features in order to train only the local forward and inverse models responsible for the current object group.

Finally, the actuator is updated. This update trains the local forward and inverse model of the actuator, if no predefined ones were provided. All update calls also update the information available in the last worldstate ws_{old}. Consequently, at the end of the update routine, the old world state and the current worldstate are identical.

When used to predict the next world state, the process visualized in figure 3.5 and described in section 3.3.1 is used for every object in the current world state. 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. The predicted actuator state is then used to compute the relative interaction features with the current object. This feature vector is then used to query both the gate and if required the object's local forward model in the predictor. The local models in the predictor do not know about the action primitives at all as a consequence of this sequential process.

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

4.3.1. Used features

Similar to the interaction model, this model also introduces a Worldstate. This worldstate is also 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 also an object, is regarded separately in the worldstate. Since the model directly predicts the new object states, no finalization is required. This model uses different kind of feature representations. The object states describe the specific features of each object and basically represent what is provided by the environment. The provided 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 object 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

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.: Table showing the different dynamic features used to represent objects.

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

require any knowledge about the features of the states 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 above. However, they do not need to represent the reference object since this model assumes, that the global configuration of the reference object does not influence the interaction. Therefore it is sufficient to model the actuator relative to the reference object, with additional information that might be useful for the learner. The relative interaction features are summarized in table 4.4.

The Closing feature c is computed as described in equation 4.6 and visualized in figure 4.1.

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

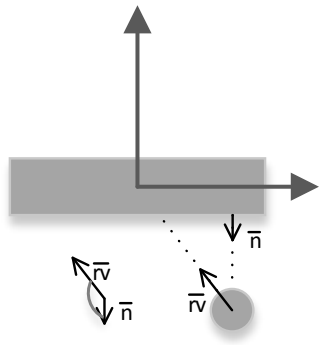

Figure 4.1.: Visualization of the closing feature. The gray half circle shows the angle whose cosine is basically computed for the feature.

where $\vec{n}$ represents the normal from the reference object towards the second object and $\overrightarrow{rv}$ represents the non normalized relative velocity vector of that second object. This equates to the cosine between these two vectors weighted by the magnitude of the relative velocity. This feature is minimal when the second object is moving directly towards the reference object. 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 object in homogeneous coordinates.

The distance is computed 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 corner-edge distance. In case of round objects, such as the actuator, the center is considered as a corner and the radius is reduced from the computed distance.

4.3.2. From target object state to action primitive(?)

[Subject to change] The overall process of computing a suitable action primitive that allows push an object towards a given target configuration is quite similar to the one in the previous model 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. Furthermore, finding the responsible local inverse model is a lot easier in this model due to the separation by object group instead of changing interaction features.

The preconditions are received as visualized in figure 3.6: First the difference vector between the current object state and the target state is computed. 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 p_{cond} need to be extracted from these preconditions. Afterwards, p_{cond} can be compared with the local actuator position p_{cur}, extracted from the current relative interaction features between the object and the actuator. If these two positions are too far apart[4] 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 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 p_{cond} the object might be influenced in a negative way.

If these actuator is close but not close enough[5] the direction towards p_{cond} can be used as action primitive. However, it might be that some part of the object is still in the direct way to the target position. 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 block. Otherwise, the same circling movement as described above is performed.

[4]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 5.1.1.

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

Once the actuator actually is close enough to 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 model, the action primitives from training are not known to the inverse model here. However, the relative interaction features include the relative velocity of the actuator. In the given scenario, this velocity equates to the action primitive, since the action primitives control the velocities. The local velocities need to be transformed to the global coordinate frame, using the transformation matrix T as explained above.

4.4. Used technologies

4.4.1. Time invariant inverse Model

The 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) the feature changes. The idea behind this inverse model is that, for each feature and each direction of the feature change, the average precondition that results in such a change is computed. These precondition depend on the actual model. In case of the interaction model, this is the combination of interaction state and action primitive, while the object state model used the relative interaction features. However, as described above, in both cases can this average be used to derive an action primitive.

The implementation of this inverse model consists of two parts: The network and its nodes. For each changing feature, the network experiences during training, a new node is created, representing the change in the given direction of that feature. In the example in figure 3.7, three nodes are created: One for the position change upwards and two for the orientation change since both directions are experienced. While all three situations are used for the position node, the orientation nodes will only be trained on one example each.

The averaging is performed by weighting each input by the magnitude of the change in the feature represented by each node. In the given example, 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.

Consider the orientation node for positive rotation 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 input features 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

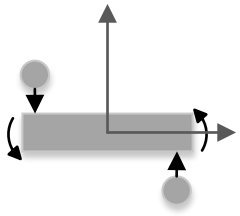

Figure 4.2.: Visualization of the averaging problem for point symmetric features. The circles represent the actuator while the rectangle represents a block object. Both pushing scenarios result in the same direction for orientation change of the object, however averaging the relative x and y positions would result in an invalid precondition.

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

Table 4.5.: Table showing the sign combinations for the example in figure 4.2.

of the object. Not only is that position not reachable, it also would not produce the desired change in orientation.

In order to prevent this, multiple averages are computed for each feature. Basically, the nodes store a separate average for each combination of signs in the input data. This model uses three different sign possibilities: +, - and 0[6]. Going back to the example in figure 4.2, assume that the following four input features are used: relative x position, relative 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 node. More specifically, for each different combination the nodes 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

[6]The sign is considered 0 if the feature $||f|| < \epsilon_{Node}$ is. 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.: Table showing the sign combinations for the example in figure 3.7.

x velocity 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.

When the node is asked to return the averaged preconditions, it tries to find up to two valid combinations. These two combinations would ideally represent two valid alternatives, as in the example above. First of all, the node tries to merge equivalent combinations. This is done, by making a special assumption about features with a 0 sign: If a feature has been experienced to be close to 0 for a given node, 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 3.7. Since a 0 does not oppose either a + or a −, all three combinations can be merged together.

The actual merging is described in algorithm 3. 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 changed above a certain threshold when training the nodes. That way, all combinations that have been experienced less often then the average number of experiences over all combinations are filtered out.

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

If only one combination remained after the merging process, the preconditions from that combination are returned. Otherwise, the node sorts the resulting combinations based on their average contribution. This can easily be computed by dividing the total weight by

```
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 occurence numbers N*
usefulKeys = filterInvalidCombinations(L, N)
sortedKeys = sortByNumZeros(usefulKeys)
while sortedKeys not empty do
    currentKey = sortedKeys[0]
    tmpWeight = 0
    tmpSum = 0
    tmpNumber = 0
    foreach key in sortedKeys do
        if isEquivalent(currentKey, key) then
            tmpWeight += W[key]
            tmpSum += I[key]
            tmpNumber += N[key]
            remove key from sortedKeys
        end
    end
    newKey = recomputeCombination(tmpSum, tmpWeight)
    W* = tmpWeight
    I* = tmpSum
    N* = tmpNumber
end
```

Algorithm 3: Description of the merging process for combinations within a node.

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[7].

From the two combinations with the biggest contribution, two preconditions are computed as described in algorithm 4.

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: average_1
Output: 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  |  |  average_1[i] = (pre_1i + pre_2i) / (w_1 + w_2)
7  |  |  average_2[i] = average_1[i]
8  |  else
9  |  |  average_1[i] = pre_1i / w_1
10 |  |  average_1[i] = pre_2i / w_2
11 |  end
12 end
```

Algorithm 4: Description of the averaging process within the nodes if at least two combinations are present.

The nodes return two results where possible since there are cases, such as the example in figure 4.2, where at least two different options are valid. When wanting to rotate the object, both preconditions visualized in the figure are valid options. Which one should be preferred needs to be determined by the network.

The inverse model 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 nodes 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.

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

In order to avoid those problems, the network uses a greedy strategy of reducing the feature with the biggest difference[8]. While the features are not normalized the biggest numerical value might not correlate to what humans regard as a big distance, this approach works quite well considering the model does not have any knowledge about the features characteristics. Once the feature with the biggest difference has been chosen, the corresponding node returns up to two possible preconditions. The same is done for the second biggest feature. By comparing the norm between the preconditions it is possible to determine which precondition for the first feature is closer to the precondition of the second feature. By choosing the closer one, the network ensures that the actuator uses a favorable alternative with respect to the next feature difference that is to be reduced. 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.

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) which will be called Adapted Instantaneous Topological Map (AITM) throughout this thesis. The Instantaneous Topological Map (ITM) [Joc00] is an adaptation of the GNG [F+95] 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 [RMS92]. 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 net 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 closes to the given input $\vec{x}$. In order to reduce the effect of the metric problem when

[8] In order to avoid oscillations between different features and preconditons, 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

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 [Buh00]. 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 $\vec{x}$:

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

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:

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

The initial matrix A is a zero matrix with proper dimensions. The learning rates η_{in}, η_{out} and η_A are meta parameter 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 $\vec{w}^n_{out}$.

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 adapted ITM inserts new nodes into the network if the current network output varies too much from the target output, i.e if:

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

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

$$d = \begin{cases} \lfloor \log_{10}(||\vec{y}^{\alpha}||) \rfloor - 1 & \text{if } ||\vec{y}^{\alpha}|| > 0 \\ -k & \text{otherwise} \end{cases} \tag{4.14}$$

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. 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 $d = 0$ which results in a threshold of 1. 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.

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. 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:

$$\forall c \in N(n) : \text{If } (\vec{w}^{n}_{in} - \vec{w}^{s}_{in}) \cdot (\vec{w}^{c}_{in} - \vec{w}^{s}_{in}) < 0 \text{ remove connection (n,c)} \tag{4.15}$$

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

Chapter 5

Evaluation

This chapter describes the different evaluation scenarios that were used to test the trained model as well as present the achieved results in each of these scenarios. The presented tests were designed to measure the forward models as well as the inverse models performances. The first scenario, called the "Push Task Simulation", tests the forward models of the different concepts, while the second test scenario, the "Move to target", tests the inverse models.

5.1. Simulation

The model is supposed to learn object manipulation through interaction. Instead of on a real robot the model has been developed and tested in a simplified simulation. For this work gazebo [KH] V.2.2.3 was used with the physics engine Open Dynamics Engine (ODE) [S+05]. This simulation and version were chosen because this thesis was initially planned as a part of a bigger project in the Citec of Bielefeld University where this software was used as well.

5.1.1. World and robot actions

The model was developed with a toy-world consisting of only a sphere, which is regarded as actuator, and a rectangular block as can be seen in figure **??**. Since the focus of this work is to learn simple object interactions, the possible actions of the robot were limited to moving the actuator freely in two dimensions.

Figure 5.1.: Overview of the used world and objects. The sphere represents the robot's actuator and the rectangular block represents the primary object.

Command	Meaning
Move Command	Sets the velocity for the actuator
Pause	Pauses the simulation
Unpause	Continues the simulation
Reset	Resets the world to starting configuration
Set Pose	Places a specified object at a certain position

Table 5.1.: Overview of all the commands implemented in the model.

5.1.2. Communication

In order to communicate with the simulation, a plug-in was written, that publishes the properties of all objects using google protobuf [**?**]. These messages are received and unpacked by a custom python interface that handles the communication with the simulation from the model's side. The plug-in also interprets the commands coming from the interface. A list of all implemented commands can be found in table 5.1

5.2. Push Task Simulation

This task is designed to test the accuracy of the forward models by decoupling them from the open world during successive predictions.

5.2.1. Scenario description

In this test, the actuator uses a constant action in order to first move towards the block and then push it. The actuator starts at different starting positions on a line parallel to the block so that the distance along the action axis always starts at 25cm between the two centers. The block is orientated so that it's main axis is perpendicular to the action axis.

The model is first trained using the open loop. This means that the model receives all information about the world at each timestep before making the prediction for the next timestep. This training is done for a set amount of "runs". A run starts when the actuator performs the first action from the starting position and ends after a fixed number of

Figure 5.2.: TODO Push Task Simulation. The left side shows one exemplary start configuration, while on the right the corresponding end configuration can be seen. The darker objects (dark blue and black) represent the actual block and actuator, while the transparent objects symbolize the predicted objects.

iterations or if the actuator travels a predefined distance. An example starting and end configuration can be seen in figure 5.2.

All starting positions except the first three are chosen randomly, but using the same seed for all models and configurations. The first three are chosen so that the actuator touches the object on the outer left, the outer right side and directly at the center. This has been done to ensure that the models have seen all general interactions for when they are evaluated with only 3 runs. The random start positions are chosen in such a range, that the actuator can also pass the object without touching it, by sampling starting positions between -0.35 and 0.35. Another seed is set after the training is done, so that all models are tested on the same starting positions, regardless of the number of training examples.

The number of iterations used here is 150 when using the model at 100Hz. During these iterations the actuator moves approximately 75cm. The maximal distance is set to 1m when using the model with 10Hz. The actual action is the same for both frequencies and is that to 0.5m/s upwards towards the block from the starting position.

5.2.2. Evaluation criteria

This task tries to evaluate the precision of the different models. In order to measure this precision, the predicted actuator position is compared with the actual actuator position. Since at least the block can also rotate around the third axis, the orientation needs to be considered as well when measuring the prediction performance. Since it is hard to find a metric that adequately combines point differences with angular differences, the orientation is indirectly measured. In addition to the center position of each object, two key points are defined that are located at the edges of the block as can be seen in figure 5.3. This way the euclidean distance can be used to measure the prediction accuracy of both the object's position as well as their orientation. The three distances are then averaged to compute the final difference score s:

$$s = \frac{1}{3} \sum_{i=1}^{3} ||p_i^{pred} - p_i^{actual}||$$

where $p_i^{pred}, pred_i^{actual}$ represents the i's predicted and actual reference point, including the center, respectively.

Figure 5.3.: TODO Image of the reference points of the block object. The two selected edges, along with the center of the block, make up the three points that are used to compare the predicted position with the actual position.

These scores are computed at each timestep and accumulated. Furthermore, the final difference, at the last timestep of each run is recorded separately. The results show the averaged results for the final difference, the accumulated differences and the mean difference over 20 test runs. The mean difference is computed by dividing the accumulated difference by the number of predictions, the model made in each run, which is equal to the number of iterations-1. For the first iteration, there has not been a prediction that can be compared yet.

5.3. Move to target

5.3.1. Scenario description

5.3.2. Evaluation criteria

5.3.3. Results

Chapter 6

Conclusion

- Discuss evaluation results
- Compare models (model configurations) with each other
- Compare to related work where possible (especially with [LMI11])
- Discuss if goals are met, or to what extend they are met
- Discuss model limitations (i.e. what will it never be able to do) (consider including possible extensions/solutions directly with the limitations))
  - Extrapolation to unseen interactions
  - Reliance on "velocity" ?
  - Interaction vector required
  - Current representation does not account for local differences in the world (only relative features are considered, the absolute position in the world is not taken into account)
  - Inverse model/circling can get stuck on loop in rare occasions, due to numeric problems

6.1. Limitations

6.2. Future work/possible improvements

Chapter 7

Bibliography

[Buh00] Buhmann, M. D.: Radial basis functions. In: Acta Numerica 9 (2000), 1, 1–38. http://dx.doi.org/null. – DOI null. – ISSN 1474–0508

[F+95] Fritzke, Bernd u. a.: A growing neural gas network learns topologies. In: Advances in neural information processing systems 7 (1995), S. 625–632

[Gib14] Gibson, James J.: The Ecological Approach to Visual Perception: Classic Edition. Psychology Press, 2014

[GMX+13] Goodfellow, Ian J. ; Mirza, Mehdi ; Xiao, Da ; Courville, Aaron ; Bengio, Yoshua: An empirical investigation of catastrophic forgeting in gradient-based neural networks. In: arXiv preprint arXiv:1312.6211 (2013)

[GS91] Geva, S. ; Sitte, J.: Adaptive nearest neighbor pattern classification. In: Neural Networks, IEEE Transactions on 2 (1991), Mar, Nr. 2, S. 318–322. http://dx.doi.org/10.1109/72.80344. – DOI 10.1109/72.80344. – ISSN 1045–9227

[Joc00] Jockusch, Ján: Exploration based on neural networks with applications in manipulator control. (2000)

[KH] Koenig, Nathan ; Howard, Andrew: 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 Bd. 3 IEEE, S. 2149–2154

[Kol14] Kolodner, Janet: Case-based reasoning. Morgan Kaufmann, 2014

[KP14] Kroemer, O. ; Peters, J.: Predicting object interactions from contact distributions. In: Intelligent Robots and Systems (IROS 2014), 2014 IEEE/RSJ International Conference on, 2014, S. 3361–3367

[LMI11] Lau, Manfred ; Mitani, Jun ; Igarashi, Takeo: Automatic learning of pushing strategy for delivery of irregular-shaped objects. In: Robotics and Automation (ICRA), 2011 IEEE International Conference on IEEE, 2011, S. 3733–3738

[LWA15] Levine, Sergey ; Wagener, Nolan ; Abbeel, Pieter: Learning Contact-Rich Manipulation Skills with Guided Policy Search. In: arXiv preprint arXiv:1501.05611 (2015)

[MMO+12] Moldovan, Bogdan ; Moreno, Pablo ; Otterlo, Martijn van ; Santos-Victor, José ; De Raedt, Luc: Learning relational affordance models for robots in multi-object manipulation tasks. In: Robotics and Automation (ICRA), 2012 IEEE International Conference on IEEE, 2012, S. 4373–4378

[NOT+08] Nishide, Shun ; Ogata, Tetsuya ; Tani, Jun ; Komatani, Kazunori ; Okuno, Hiroshi G.: Predicting object dynamics from visual images through active sensing experiences. In: Advanced Robotics 22 (2008), Nr. 5, S. 527–546

[RMS92] Ritter, Helge ; Martinetz, Thomas ; Schulten, Klaus: Neural computation and self-organizing maps. An introduction. 1992

[S+05] Smith, Russell u. a.: Open dynamics engine. (2005)

Acronyms

k-NN k-Nearest Neighbor. 5, 7

AITM Adapted Instantaneous Topological Map. 6, 25–27, 42, 44

BN Bayesian Network. 6

GNG Growing Neural Gas. 6, 25, 42

ITM Instantaneous Topological Map. 42, 44

LLM Local Linear Map. 6, 42

Appendix A

Eidesstattliche Erklrung

Ich versichere hiermit, daß ich die vorliegende Arbeit selbständig verfaßt und keine anderen als die im Literaturverzeichnis angegebenen Quellen benutzt haben.

Alle Stellen, die wörtlich oder sinngemäß aus veröffentlichten oder noch nicht veröffentlichten Quellen entnommen sind, sind als solche kenntlich gemacht.

Die Zeichnungen oder Abbildungen in dieser Arbeit sind von mir selbst erstellt worden oder mit einem entsprechenden Quellennachweis versehen.

Die Arbeit ist in gleicher oder ähnlicher Form noch bei keiner anderen Prüfungsbehörde eingereicht worden.

Bielefeld, den XX. Monat 20XX

Nachname, Vorname