Using Kernel Perceptrons to Learn Action Effects for Planning

Using Kernel Perceptrons to Learn Action Effects for Planning

Kira Mourão
School of Informatics
University of Edinburgh
Edinburgh EH8 9LW, Scotland, UK
k.m.t.mourao@sms.ed.ac.uk

Ronald P. A. Petrick
School of Informatics
University of Edinburgh
Edinburgh EH8 9LW, Scotland, UK
rpetrick@inf.ed.ac.uk

Mark Steedman
School of Informatics
University of Edinburgh
Edinburgh EH8 9LW, Scotland, UK
steedman@inf.ed.ac.uk

***Abstract*— We investigate the problem of learning action effects in STRIPS and ADL planning domains. Our approach is based on a kernel perceptron learning model, where action and state information is encoded in a compact vector representation as input to the learning mechanism, and resulting state changes are produced as output. Empirical results of our approach indicate efficient training and prediction times, with low average error rates ($<3\%$) when tested on STRIPS and ADL versions of an object manipulation scenario. This work is part of a project to integrate machine learning techniques with a planning system, as part of a larger cognitive architecture linking a high-level reasoning component with a low-level robot/vision system.**

I. Introduction

Artificial intelligence planning systems provide a powerful tool for controlling a cognitive agent's actions in both real-world and artificial domains. A drawback of such approaches is that they require a model of the dynamics of the domain in which the agent will operate. In real-world domains, such models may not be readily available, or may not properly account for unexpected subtleties that arise in the world when a model is constructed *a priori* by hand. An alternative, more desirable approach is to endow the agent with the ability to *learn* from its environment in order to induce a world model, and the effects of its actions, from its experiences.

Using machine learning techniques to learn action models is not a new idea. Prior approaches have applied inductive learning [1] and directed experimentation [2] techniques to data represented in first-order logic, without noise or non-determinism. Other approaches have used schema learning to learn probabilistic action rules operating on discrete-valued sensor data [3]. Also, k-means clustering of equivalence classes, followed by extraction of sensor data features, has been used to train support vector machines (SVMs) to predict deterministic action effects in a given context [4]. Recently, attention has shifted to methods which exploit relational structure in order to improve speed and generalisation performance. For instance, [5] generates and refines rules using heuristic search guided by maximum likelihood, and shows that relational deictic rules are learnt more effectively than propositional or purely relational rules. Similarly, [6] uses a logical inference algorithm to efficiently learn rules in relational environments.

Our approach is based on a connectionist learning model, namely *kernel perceptron learning* [7], [8]. Such methods are particularly useful since they can be shown to provide good performance, in terms of both the training time and the quality of the learnt models. We focus on one aspect of the learning problem in this paper, namely learning the effects of an agent's actions, given a set of actions and their preconditions. Currently, our learning method assumes a fully observable world for training purposes (i.e., complete world state descriptions), however, it can be made much more general. For instance, our approach can be extended to handle noisy data [9], and we believe it can also be used to learn action preconditions and more complex representations.

Since we would like to apply our techniques to real planning systems, we will focus on two different types of action representations commonly used in the planning community: STRIPS actions [10] and ADL actions with conditional effects [11]. We consider deterministic domains with actions that affect a subset of the properties (predicates) that make up the world state. In our approach, we use a representation that makes efficient use of predicates, and follow the approach of [5] where deictic referencing is used to reduce the complexity of the representation. We demonstrate that kernel perceptrons can be used successfully to learn the dynamics of an object manipulation domain, in a manner that is independent of the number of objects in the world, making it suitable for large planning scenarios.

This paper is organized as follows. In Section II we discuss the planning representations we are interested in learning. In Section III we describe kernel perceptrons and how we use them to learn action effects. In Section IV we present the results of our learning experiments. In Sections V and VI we discuss our results and our plans to incorporate these techniques into a cognitive architecture linking a high-level planning system to a low-level robot/vision system.

II. Action representations for planning

The action representations we will use are based on the logical representations typically found in planning systems. A *domain* $\mathcal{D}$ is defined as a tuple $\mathcal{D} = \langle \mathcal{O}, \mathcal{P}, \mathcal{A} \rangle$, where $\mathcal{O}$ is a finite set of world objects, $\mathcal{P}$ is a finite set of predicate (relation) symbols, and $\mathcal{A}$ is a finite set of actions. Each predicate and action also has an associated arity. Predicates of arity 0 are referred to as *object independent* properties, while those of arity at least 1 are *object dependent* properties.

TABLE I

STRIPS ACTIONS FROM AN OBJECT MANIPULATION DOMAIN

Action	**Preconditions**	**Effects**
graspA-table(x)	*clear*(x)	*add*(*ingripper*(x))
	gripperempty	*del*(*gripperempty*)
	ontable(x)	*del*(*ontable*(x))
graspA-stack(x, y, z)	*clear*(x)	*add*(*ingripper*(x))
	gripperempty	*add*(*clear*(y))
	isin(x, y)	*del*(*gripperempty*)
	instack(x, z)	*del*(*isin*(x, y))
		del(*instack*(x, z))

A *fluent* is an expression $p(c_1, c_2, \ldots, c_n)$, where $p \in \mathcal{P}$, n is the arity of p, and each $c_i \in \mathcal{O}$. A *state* is any set of fluents, and $\mathcal{S}$ is the set of all possible states. For any state $s \in \mathcal{S}$, a fluent p is true at s iff $p \in s$. The negation of a fluent, $\neg p$, is true at s (also, p is false at s) iff $p \notin s$.

Each action $a \in \mathcal{A}$ is defined by a set of *preconditions*, *Pre*$_a$, and a set of *effects*, *Eff*$_a$. *Pre*$_a$ can be any set of fluents and negated fluents. We consider two different kinds of action effects, both of which are commonly found in planning domains. In STRIPS actions [10], each effect $e \in$ *Eff*$_a$ has the form *add*(p) or *del*(p), where p is any fluent. In ADL actions [11], each effect $e \in$ *Eff*$_a$ is either a standard STRIPS effect, or a *conditional effect* of the form $C_e \Rightarrow$ *add*(p) or $C_e \Rightarrow$ *del*(p). Here, C_e is any set of fluents and negated fluents, and is referred to as the *secondary preconditions* of effect e. Action preconditions and effects can also be parametrized. An action with all of its parameters replaced with objects from $\mathcal{O}$ is said to be an *action instance*.

Action instances are state transforming. Given a state s and an action instance A, A is *applicable* (or *executable*) at s iff each precondition $p \in$ *Pre*$_A$ is true at s. An applicable action produces a new state s' that is identical to s, but updated with the effects of A as follows: for each $e \in$ *Eff*$_A$, (i) if e is an effect *add*(p) then p is added to s', (ii) if e is an effect *del*(p) then p is removed from s', (iii) if e is an effect $C_i \Rightarrow$ *add*(p) then p is added to s' provided all fluents of C_i are true at s, and (iv) if e is an effect $C_i \Rightarrow$ *del*(p) then p is removed from s provided the fluents of C_i are true at s.

In this paper we will focus on a specific object manipulation scenario, represented as a simple planning domain, where a robot has the ability to grasp, stack, and remove objects from a table environment.[1] For instance, the domain includes actions like *graspA-table*(x) ("grasp object x from the table using grasp type A"), *graspA-stack*(x, y, z) ("grasp object x from object y in a stack with z at its base using grasp type A"), and *putAway*(x) ("put object x away on a shelf"). The domain also includes properties like *gripperempty* ("the robot's gripper is empty"), *clear*(x) ("object x has no objects on top of it"), *ontable*(x) ("object x is on the table"), and *isin*(x, y) ("object x is in object y"). As we'll see in Section V, this domain is motivated by work that aims to link a robot/vision system with a planner, within an architecture where the robot can learn and act [12], [13].

[1]This domain is similar to Blocksworld, but has been extended to include more complex grasping actions and management of limited resources.

Input vector	Corresponding action/predicate	
0	*graspA-table*(*obj1*)	
1	*graspA-stack*(*obj1*, *obj2*, *obj3*)	
0	*graspB-table*(*obj1*)	
0	*graspD-table*(*obj1*)	Actions
0	*putInto-objectOnTable*(*obj1*, *obj2*)	
0	*putInto-stack*(*obj1*, *obj2*)	
0	*putAway*(*obj1*)	
0	*gripperempty*	Object independent
	. . .	properties
0	*ontable*	
1	*clear*	Properties related
0	*isin-obj1*	to grasped object (1)
1	*isin-obj2*	
	. . .	
1	*ontable*	
0	*clear*	Properties related
0	*isin-obj1*	to grasped object (2)
0	*isin-obj2*	
	. . .	
0	*ontable*	
0	*clear*	Properties related
0	*isin-obj1*	to grasped object (3)
0	*isin-obj2*	
	. . .	

Fig. 1. Representation of an action and state as a binary input vector

Table I shows the encoding of two STRIPS actions in our object manipulation domain, *graspA-table* and *graspA-stack*. Actions such as these provide a straightforward representation of the manipulation tasks that can be performed. For instance, if we have a state s defined by the set {*clear*(*obj1*), *clear*(*obj2*), *gripperempty*, *ontable*(*obj1*), *ontable*(*obj2*)}, then the action instance *graspA-table*(*obj1*) is applicable at s. Applying the effects of this action instance to s produces the new state s' = {*clear*(*obj1*), *clear*(*obj2*), *ingripper*(*obj1*), *ontable*(*obj2*)}. We will also consider an ADL version of this domain in our testing.

III. KERNEL PERCEPTRON ACTION LEARNING

The planning actions in the previous section give rise to a simple state transition system, where the application of an action at a state produces a new state. In this model, an action's effects determine the *changes* made to a state during execution. Since a state is simply a set of fluents, the transition between states is simply the difference between two sets of fluents. Our goal in this section is to develop an approach that *learns* these differences between states.

Learning the complete dynamics of a planning domain requires the ability to learn both the preconditions needed to perform an action, and the effects of the action at a particular state. In this paper, we will only focus on the effects problem, and will simply assume that the action set and action preconditions are supplied to our learning mechanism as part of the input. (We also believe our approach extends to the problem of learning action preconditions; see Section V).

The specific learning method we will use is a connectionist machine learning model based on *kernel perceptrons* [7], [8]. Kernel perceptrons obtain reasonable accuracy at acceptable training and prediction speeds, allowing us to use this approach in practical planning applications. Alternative nonlinear classifiers, such as SVMs, can be substantially slower [14] while performance is not guaranteed to be better [15].

In order to effectively use kernel perceptrons, we must consider how best to encode our learning problem in terms of the inputs and outputs of the learning mechanism. We consider each of these problems in turn, as well as the overall operation of our learning approach.

A. Input representation

The input to our learning mechanism uses a vector representation that encodes a description of the action being performed and the state at which the action is applied. For each action in the domain, the vector includes an element that is set to 1 if the action is to be performed, or 0 otherwise. For states, we consider object-independent and object-dependent properties separately. In the case of object-independent properties (e.g., *gripperempty*), the vector includes a single element for each property of the domain, representing the truth value of that property (fluent) at the state being considered: the element is set to 1 if the fluent is true at the state, or 0 if the fluent is false at the state.

For object-dependent properties, we avoid representing all possible fluents, which could lead to very large input vectors. Instead, we consider each property on a per object basis, by representing only those properties of the objects directly involved in the action being applied, and the objects related in some way to those objects. Additionally, a form of deictic representation is used (similar to [5]), where objects are specified in terms of their roles in the action, or their roles in a predicate with another object. For example, in Table I the only object involved in *graspA-table* is the "grasped object" x. In *graspA-stack* the objects include the "grasped object" x, and the related objects "object containing the grasped object" y and "object at base of grasped object's stack" z.

Rather than maintaining a "slot" in the input vector for each possible role, roles are allowed to overlap. The only constraint is that two objects with the same role in the same action in two different instances of the action must always be represented at the same slot in the input vector. Thus, each object is represented by a set of inputs, one for each object-specific predicate (such as *ingripper*), and each relation with another object (such as *isin*). To bind relations to the correct objects, extra predicates are used which relate the current object to one or more other objects, identified by their slot (e.g., *isin-obj1*, *isin-obj2*, etc.). This representation significantly reduces the number of inputs to the learning mechanism, and is dependent on the complexity of the actions and relations between objects, rather than the number of objects in the domain.

The final input vector has the form: ⟨*actions, object-independent properties, object slot 1 predicates, object slot 2 predicates, …, object slot n predicates*⟩. Fig. 1 shows an example of an input vector for an action-state pair. In this case, the action performed is *graspA-stack*. The "grasped object" properties are represented in the object *obj1* slot, while the "object below the grasped object" properties are represented in the object *obj2* slot. Here, $\mathit{clear}(\mathit{obj1})$, $\mathit{isin}(\mathit{obj1}, \mathit{obj2})$ and $\mathit{ontable}(\mathit{obj2})$ are shown to be true. No further object properties are included in the state in this example, and so all the remaining bits are set to 0.

B. Output representation

The output of the learning mechanism is a prediction of the set of domain properties that will change if the given action is performed at the given state. As with the input, this is encoded as a binary vector, with each output representing a state property: the output value is 1 if the property changes and 0 if it does not. As with the input vector, object-independent properties are represented by single elements, while object-specific properties are again represented on a per-object basis in slots. Thus, the output vector has the form: ⟨*object-independent properties, object slot 1 predicates, object slot 2 predicates, …, object slot n predicates*⟩.

C. Learning

The task of the learning mechanism is to learn the associations between action-precondition pairs and their effects, that is, rules of the form $\langle A, \mathit{Pre}_A\rangle \rightarrow \mathit{Eff}_A$. As a result of the form of the planning actions we allow, effects are assumed to be deterministic and disjunctive effects (i.e., effects of the form "either p_1 or p_2 changes") are not allowed. Instead, all effects involve either conjunctions of predicates (in the case of STRIPS) or conjunctions of predicates conditioned on other conjunctions of predicates (in the case of ADL). This means that it is sufficient to learn the rule for each effect predicate separately. Thus, we can treat the learning problem as a set of binary classification problems, one for each (conditional) effect predicate.

A simple, fast, binary classifier that can be used to address our particular learning problem is the *perceptron* [16]. The perceptron maintains a weight vector $\mathbf{w}$ which is adjusted at each training step. The i-th input vector $\mathbf{x}_i \in \{0,1\}^n$ in a class $y \in \{-1,1\}$ is classified by the perceptron using the decision function $f(\mathbf{x}_i) = \mathit{sgn}(\langle \mathbf{w} \cdot \mathbf{x}_i\rangle)$. If $f(\mathbf{x}_i)$ is not the correct class then $\mathbf{w}$ is set to $\mathbf{w} + y\mathbf{x}$; if $f(\mathbf{x}_i)$ is correct then $\mathbf{w}$ is left unchanged. Provided the data is linearly separable, the perceptron algorithm is guaranteed to converge on a solution in a finite number of steps [17], [18]. If the data is not linearly separable then the algorithm oscillates, changing $\mathbf{w}$ at each misclassified input vector.

One solution for non-linearly separable data is to map the input feature space into a higher-dimensional space where the data is linearly separable. However, an explicit mapping leads to a massive expansion in the number of features which may make the classification problem computationally infeasible. Instead, an implicit mapping can be achieved by applying the *kernel trick* to the perceptron algorithm [8]. The kernel trick is applied by noting that the decision function can be written in terms of the dot product of the input vectors:

$$f(\mathbf{x}_i) = \mathit{sgn}(\langle \mathbf{w} \cdot \mathbf{x}_i\rangle) = \mathit{sgn}(\sum_{j=1}^{n} \alpha_j y_j \langle \mathbf{x}_j \cdot \mathbf{x}_i\rangle),$$

where α_j is the number of times the j-th example has been misclassified by the perceptron. By replacing the dot product with a *kernel function* $k(\mathbf{x}_i, \mathbf{x}_j)$ which calculates $\langle \phi(\mathbf{x}_i) \cdot \phi(\mathbf{x}_j) \rangle$ for some mapping ϕ, the perceptron algorithm can be run in higher dimensional space without ever requiring the mapping to be explicitly calculated.

Since in general the problem of learning action effects is not linearly separable, the kernel perceptron is an appropriate choice for this problem. An ideal kernel is one which allows the perceptron algorithm to run over the feature space of all conjunctions of features in the original input space, as this would allow an accurate representation of the exact conjunction of features (action and preconditions) corresponding to a particular effect. Such a kernel is $k(x, y) = 2^{same(x,y)}$, where $same(x, y)$ is the number of bits with the same value in both x and y [19], [20]. As we'll see in Section V, we also believe our approach can be made much more general, letting us relax some of our earlier restrictions.

IV. Empirical results

In this section we describe the results of testing our learning procedure on the object manipulation domain described in Section II. Data was simulated from the domain description, for both a purely STRIPS version of the actions and an ADL version with context-dependent action effects. (For example, the two STRIPS actions in Table I were merged into a single ADL action, along with other changes.) Each case was generated by randomly selecting an action, and setting the inputs for the preconditions required for the action to 1. The action input was set to 1, and all other action inputs to 0. The remaining irrelevant inputs were used to create separate training and testing input data sets. For the training data, half of the inputs in each instance were randomly set to 0 or 1, with the other half all set to 0 (vice versa for the testing data). Outputs were set to 1 if the feature changed as a result of the action and 0 if not. Overall, 3000 training and 500 testing examples were generated with the (strong) assumptions that (i) no noise was present in the training/testing set, and (ii) no irrelevant output data was included in the training examples (i.e., only relevant changes were provided). To determine an error bound on our results, 10 runs with different randomly generated training and testing sets were used. Our testing environment was a 2.4 GHz quad-core system with 6 Gb of RAM. All times were measured for Matlab 7.2.0.294.

The results of our testing are shown in Fig. 2. Overall, the kernel perceptron learnt the training data and performed well on the testing data with a low error rate. Fig. 2(a) shows the error rate for the learnt STRIPS actions, while Fig. 2(b) shows the error rate for the ADL actions. In both cases, the average error dropped to less than 3% after 700 training examples. The standard perceptron error rate, included for comparison, shows significantly worse performance: over 5% error after 3000 training examples. Fig. 2(c) shows the training time for both STRIPS and ADL actions (for 1 bit of the effect vector), while Fig. 2(d) shows the prediction time (for 1 bit of 1 prediction). In both cases, the kernel perceptron method is quite efficient. Perhaps the most surprising result is that there is little difference between the training and prediction times of STRIPS actions, compared with those for ADL actions, at least for our particular testing domain. In general, performance on ADL domains will always take longer than STRIPS domains, particularly when the conditional effect training examples are very dissimilar to the other training examples available. (The ADL problem is slightly more difficult, even in our domain.) Additional testing is needed on more complex domains, to determine to what extent the STRIPS and ADL results remain similar.

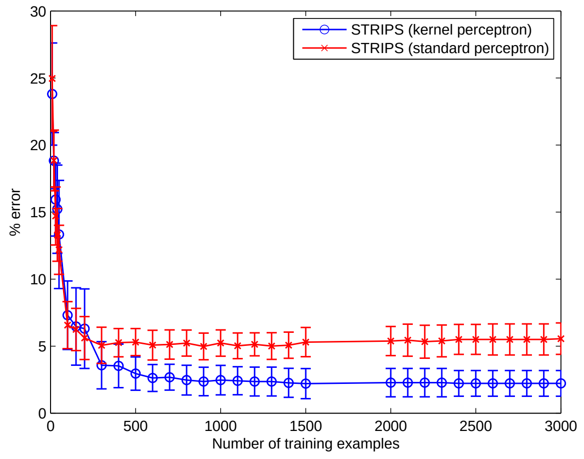
In our implementation, performing training or prediction consists of two steps: calculating the kernel matrix, followed by either the perceptron algorithm loop or the decision function calculation, respectively. The kernel matrix calculation does not vary with the difficulty of the problem. Calculating the kernel matrix for training is $O(n^2)$, where n is the number of training examples. However, only around 700 examples are required to achieve sufficient generalisation for planning in the test domain, corresponding to under 0.25 seconds to calculate the kernel matrix. Similarly for prediction, calculating the kernel matrix is $O(mn)$, where n is the number of training examples and m the number of testing examples. For 700 training examples and 500 testing examples the computation time is also below 0.25 seconds.

For the second step, estimates of $O(n^2)$ for training and $O(n)$ for prediction are valid for the worst case, where the kernel matrix entries for every pair of training examples have to be used in the perceptron algorithm loop (so every training example contributes to the weight vector), and where kernel matrix entries for every training example paired with every test example have to be used for the decision function calculation (again, when every training example contributes to the weight vector). The overall time is also affected by such factors as the number of bits in the input vector (which affects every calculations of the kernel matrix entries), the number of iterations the perceptron algorithm has to make (which affects training), and the number of training vectors which contribute to the weight vector (which affects both training and prediction). For the testing domain, however, we have almost linear training time and constant prediction time in the number of training examples.

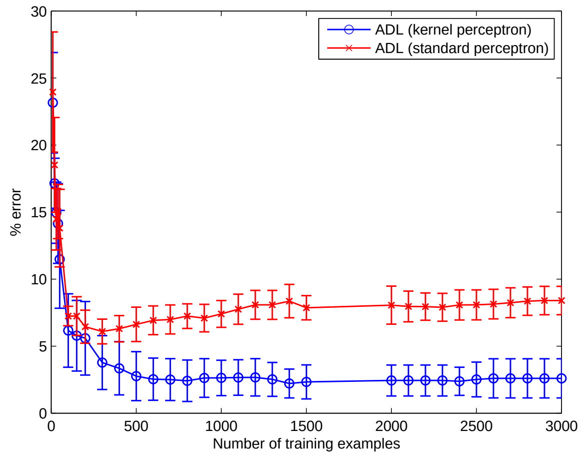
V. Discussion

The results of our experiments show that kernel perceptrons are able to learn the dynamics of a planning domain. In order to test the feasibility of our approach under real planning conditions, we are currently integrating our learning mechanism with the PKS (Planning with Knowledge and Sensing) planner [21], [22]. In this case, the planner uses the network as part of its action model, by querying the network during plan construction to determine action effects at particular states. Since planning systems traditionally use efficient rule-based action models, we must still evaluate the extra overhead resulting from our network-based approach.

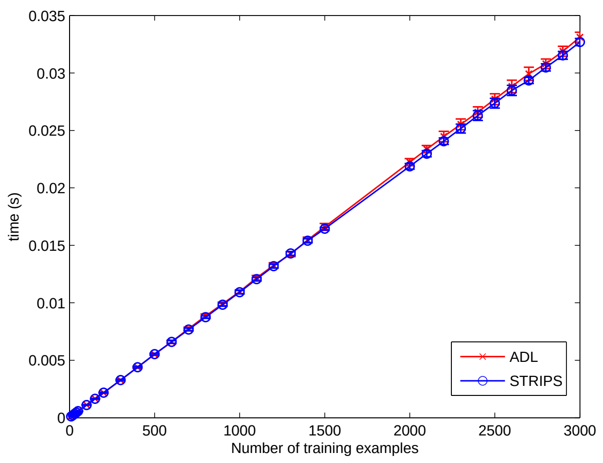
We also envision a more incremental approach to train and use our learning mechanism, as a component of a cognitive architecture of the kind reported in [12], where a low-level robot/vision system is linked to a high-level planning system.



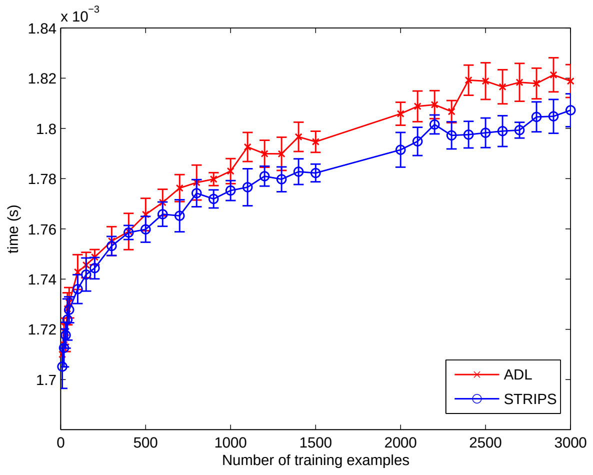
(a) Average error rate for learnt STRIPS actions



(b) Average error rate for learnt ADL actions



(c) Training time for STRIPS and ADL actions



(d) Prediction time for STRIPS and ADL actions

Fig. 2. Results from experiments in the object manipulation domain

Rather than employing a completely offline training phase for our network, we foresee a more interactive approach, where online training data is generated from the robot's initial experiences in its environment, and the planner is able to use early action models to generate plans (albeit, lower quality plans). As more training data becomes available, plan quality should also increase, allowing the robot to learn how better to act. An existing integration project that uses the object manipulation scenario described in this paper, may provide a useful testbed for our learning techniques [13].

Testing in a robot/planning environment will also allow us to investigate the effectiveness of our learning method in practice. For instance, although the error rates for our learnt actions are low ($< 3\%$), it is unclear what effect this will have on the quality of plans constructed for this domain. Since replanning often has to take place in real-world robot domains, having a perfect plan is not always necessary. On the other hand, our method also assumes deterministic outcomes of actions, whereas some actions might better be modelled with probabilistic outcomes in this environment. In this case, we may again be able to use replanning techniques to some extent, but may also have to consider more substantial changes to our approach.

Currently, our approach makes certain assumptions that are not always realistic, especially when data is provided by real-world systems. For instance, we assume that there is no noise in the input or output data, and no irrelevant data in the training outcomes. We can relax the first assumption about noisy data, by using a noise-tolerant variant of the perceptron algorithm, such as adding a margin term [9]. We also believe such techniques can be used to handle irrelevant output data, since by definition such changes behave like noise. (In particular, irrelevant outputs correspond to irrelevant state changes in the action effects.) Otherwise, we run the risk of having perceptrons that fail to converge, or produce error-prone output when trying to predict such cases.

Representationally, there is also an issue with predicting changes to domain properties that are dependent on more primitive properties, as such changes can be wider-reaching than changes to the purely primitive properties themselves.

For example, our object manipulation scenario allows for situations where the robot can combine two existing stacks of objects into one large tower, by gripping the base object of one stack and releasing it at the top of another stack. In our initial planning representation, a predicate $\mathit{instack}(x, y)$ is used to indicate that a block x is in a stack with y at its base. Thus, after combining two stacks, *instack* must be updated for all objects in the "gripped" stack, to reflect the new base object of the single tower. In order to update *instack* correctly using our approach, the output would have to represent all of the objects in both stacks. Currently, only objects directly acted on, or related to directly acted on objects, are represented. Rather than attempt to represent all the objects required, it is easier to treat *instack* as a derived predicate defined in terms of more primitive properties (e.g., *isin* and *ontable*). This is also the approach taken in [5], where derived "concepts" are used in the rule antecedents but only primitive properties are used in the outcomes.

Additional testing is needed to determine the scalability of our approach on more difficult planning domains, although we expect that it should scale well with the number of predicates and actions. We also plan to compare our results to those of other classifiers, such as SVMs. We are currently investigating extensions to our approach to learn more comprehensive, and more complex, action models. For instance, we believe that our kernel perceptron approach could also be used to learn action preconditions, provided it is possible to only represent a small number of objects in the state at a time. Such an extension would require a means of choosing which objects to consider, and may ultimately need to be learnt. An attentional mechanism of some sort may be of help in this task [23], [5]. We also believe that our approach can be adapted to learn more sophisticated action representations, such as those used by PKS to describe knowledge and sensing. Since PKS's representation is based on an extended version of STRIPS/ADL, many of its features are similar to those that can already be learnt by our methods.

VI. Conclusions

In this paper we presented a mechanism based on kernel perceptrons, to address the problem of learning STRIPS and ADL action effects for planning domains. Overall, our approach demonstrated efficient performance on our testing sets, with low average error rates ($< 3\%$) for the learnt action effects. This work is also part of a larger cognitive architecture linking a high-level reasoning component with a low-level robot/vision system. We are currently in the process of integrating our learning method with the PKS planning system, and testing our approach on more complex planning domains including a real-world robot environment [13].

Acknowledgements

This work was partially funded by the European Commission through the PACO-PLUS project (FP6-2004-IST-4-27657) and the UK EPSRC/MRC through the Neuroinformatics Doctoral Training Centre, University of Edinburgh.

References

[1] X. Wang, "Learning by observation and practice: An incremental approach for planning operator acquisition," in *Proc. of the International Conference on Machine Learning (ICML-95)*, 1995, pp. 549–557.

[2] Y. Gil, "Learning by experimentation: Incremental refinement of incomplete planning domains," in *Proceedings of the International Conference on Machine Learning (ICML-94)*. MIT Press, 1994.

[3] M. Holmes and C. Isbell, "Schema learning: Experience-based construction of predictive action models," in *Advances in Neural Information Processing Systems (NIPS) 17*, 2005, pp. 585–562.

[4] M. R. Doğar, M. Çakmak, E. Uğur, and E. Şahin, "From primitive behaviors to goal directed behavior using affordances," in *Proc. of Intelligent Robots and Systems (IROS 2007)*, 2007.

[5] H. M. Pasula, L. S. Zettlemoyer, and L. P. Kaelbling, "Learning symbolic models of stochastic world domains," *Journal of Artificial Intelligence Research*, vol. 29, pp. 309–352, 2007.

[6] D. Shahaf and E. Amir, "Learning partially observable action schemas," in *Proceedings of the National Conference on Artificial Intelligence (AAAI-06)*. AAAI Press, 2006.

[7] M. A. Aizerman, E. M. Braverman, and L. I. Rozoner, "Theoretical foundations of the potential function method in pattern recognition learning," *Automation and Remote Control*, vol. 25, pp. 821–837, 1964.

[8] Y. Freund and R. Schapire, "Large margin classification using the perceptron algorithm," *Machine Learning*, vol. 37, pp. 277–296, 1999.

[9] R. Khardon and G. M. Wachman, "Noise tolerant variants of the perceptron algorithm," *Journal of Machine Learning Research*, vol. 8, pp. 227–248, 2007.

[10] R. E. Fikes and N. J. Nilsson, "STRIPS: A new approach to the application of theorem proving to problem solving," *Artificial Intelligence*, vol. 2, pp. 189–208, 1971.

[11] E. P. D. Pednault, "ADL: Exploring the middle ground between STRIPS and the situation calculus," in *Proc. of Principles of Knowledge Representation and Reasoning (KR-89)*. Morgan Kaufmann Publishers, 1989, pp. 324–332.

[12] C. Geib, K. Mourão, R. Petrick, N. Pugeault, M. Steedman, N. Krueger, and F. Wörgötter, "Object action complexes as an interface for planning and robot control," in *Proceedings of the Humanoids-06 Workshop: Towards Cognitive Humanoid Robots*, Genoa, Italy, 2006.

[13] D. Kraft, E. Başeski, M. Popović, A. M. Batog, A. Kjær-Nielsen, N. Krüger, R. Petrick, C. Geib, N. Pugeault, M. Steedman, T. Asfour, R. Dillmann, S. Kalkan, and F. Wörgötter, "Exploration and planning in a three-level cognitive architecture," in *Proc. of the International Conference on Cognitive Systems (CogSys 2008)*, 2008.

[14] M. Surdeanu and M. Ciaramita, "Robust information extraction with perceptrons," in *Proceedings of the NIST 2007 Automatic Content Extraction Workshop (ACE07)*, Mar. 2007.

[15] T. Graepel, R. Herbrich, and R. C. Williamson, "From margin to sparsity," *Advances in Neural Information Processing Systems*, vol. 13, pp. 210–216, 2000.

[16] F. Rosenblatt, "The perceptron: a probabilistic model for information storage and organization in the brain," *Psychological Review*, vol. 65, no. 6, pp. 386–408, Nov. 1958.

[17] A. B. Novikoff, "On convergence proofs for perceptrons," in *Proceedings of the Symposium on the Mathematical Theory of Automata*, vol. 12, 1963, pp. 615–622.

[18] M. L. Minsky and S. A. Papert, *Perceptrons*. The MIT Press, 1969.

[19] K. Sadohara, "Learning of boolean functions using support vector machines," in *Proc. of Algorithmic Learning Theory,* Lecture Notes in Artificial Intelligence, vol. 2225. Springer, 2001, pp. 106–118.

[20] R. Khardon, D. Roth, and R. A. Servedio, "Efficiency versus convergence of boolean kernels for on-line learning algorithms," *Journal of Artificial Intelligence Research*, vol. 24, pp. 341–356, 2005.

[21] R. P. A. Petrick and F. Bacchus, "A knowledge-based approach to planning with incomplete information and sensing," in *Proc. of Artificial Intelligence Planning and Scheduling (AIPS-2002)*. AAAI Press, 2002, pp. 212–221.

[22] ——, "Extending the knowledge-based approach to planning with incomplete information and sensing," in *Proc. of Automated Planning and Scheduling (ICAPS-04)*. AAAI Press, 2004, pp. 2–11.

[23] D. Kragic, M. Björkman, H. I. Christensen, and J.-O. Eklundh, "Vision for robotic object manipulation in domestic settings," *Robotics and Autonomous Systems*, vol. 52, no. 1, pp. 85–100, July 2005.