Generalised Policies for Probabilistic Planning with Deep Learning

Sam Toyer

A thesis submitted for the degree of
Bachelor of Advanced Computing (Research & Development, Honours)
at the
Australian National University

October, 2017

Declaration

This thesis is an account of research undertaken between February 2017 and October 2017 at the Research School of Computer Science, College of Engineering and Computer Science, the Australian National University, Canberra, Australia.

Except where acknowledged in the customary manner, the material presented in this thesis is, to the best of my knowledge, original and has not previously been submitted in whole or part for a degree in any university.

Sam Toyer
October, 2017

Acknowledgements

I'd like to acknowledge my supervisors, Sylvie Thiébaux and Lexing Xie. Their extensive guidance was essential to the success of this project, and has helped me to grow as a researcher. I am also grateful for their tireless effort commenting on various drafts, and for the exceptional supportiveness which they've shown throughout the year. I would also like to thank Felipe Trevizan for effectively acting as a third supervisor for several months. Felipe's insightful comments and discussion have also shaped the direction of this research, and strengthened its presentation. Sylvie, Lexing and Felipe also deserve thanks for their assistance in distilling this research into a conference submission (Toyer et al., 2017), which is currently under review. Thank you to Felipe in particular for tuning the probabilistic planning baselines which appeared in that paper, and running them on the CSIRO cluster; those baselines appear again as part of the empirical evaluation in Chapter 5. Finally, I want to acknowledge the anonymous conference reviewers whose comments helped improve aspects of both the aforementioned paper and this thesis; Max Wang, for commenting on a draft thesis; the ANU, which helped support my time in Canberra; and my friends and family, for making the past four years a positive experience.

Abstract

Probabilistic planning is the task of deciding which actions an intelligent agent should take in order to achieve a set of goals in a discrete environment. This task requires the agent to anticipate the consequences of its actions far into the future, and also to consider contingencies arising from actions with stochastic outcomes. Typically, this task is solved using a search through the state space of the environment, guided by a heuristic which can identify promising directions in which to expand the search. Planners employing this strategy have successfully been used in robotics, operations research, and other areas. However, their scalability is limited by the fact that neither common search procedures nor popular heuristics are able to improve their performance with experience. In this thesis, we propose a new method for learning policies—that is, mappings from states of an environment to actions—which can provide effective guidance to an agent in a specific planning problem, and which can generalise to other planning problems drawn from the same domain. This can improve scalability by allowing planners to learn to exploit structural features of a domain on small, easily-solved problems, then transfer that knowledge to larger problems which lie beyond the reach of a non-learning planner.

Concretely, this thesis makes three contributions. Our first contribution is a neural network architecture which is specialised to the structure of planning problems in much the same way that a convolutional neural network is specialised to the structure of images. Our second, related contribution is a weight-sharing scheme for the aforementioned neural network architecture which allows weights learnt for one planning problem from a given planning domain to be applied directly to any other problem from the same domain—in other words, our network can generalise. Finally, our third contribution is a scheme for efficiently training the proposed neural network architecture in a supervised manner. Through experiments, we show that the resultant learning-based system is able to learn good policies for complex problems, and in some instances scale far beyond the capabilities of traditional non-learning planners. While we only apply the proposed architecture to the task of learning generalised policies, it is worth noting that it could potentially be applied to learning generalised heuristics, producing vector-space embeddings of the states of planning problems, and so on. We leave exploration of these further applications for future work.

Contents

Chapter 1

Introduction

Planning and learning are key ingredients for constructing intelligent agents. Planning imbues an agent with the ability to choose actions which will help it to achieve some specified goal within an environment. Learning allows the agent to improve its ability to achieve goals as it gains experience. This thesis considers how learning—in particular, learning with neural networks—can be used to make planning more efficient.

1.1 Planning

In this thesis, we are primarily concerned with probabilistic planning (Mausam and Kolobov, 2012), which is a generalisation of the well-studied problem of classical planning (Geffner and Bonet, 2013). Like classical planning, probabilistic planning considers discrete, fully-observable environments with a finite number of states and actions. Both probabilistic and classical planning assume that an agent's interaction with its environment is a sequential process which takes place in discrete time steps. At each time step, the agent observes the current state of the environment in its entirety, and can choose exactly one action to execute. Probabilistic planning differs from classical planning in that actions can be stochastic: applying an action will transition the agent into one of several possible successor states, sampled from a known state transition distribution.

Computationally, planning is extremely difficult. The problem of determining whether a goal is achievable in a purely deterministic problem is already PSPACE-complete (Bylander, 1994). Further, if a sequence of actions to attain a goal does exist then its length could be exponential in the size of the problem description (Porco et al., 2013). The uncertainty inherent in probabilistic planning adds another dimension to this already challenging problem, as agents have to strike a balance between taking short and risky paths or longer but more reliable ones. However, probabilistic planning is just one point on a spectrum of expressiveness and tractability among planning formalisms. The class of probabilistic planning problems which we consider does not include problems with continuous states, partially-observable environments, concurrently executable actions, or various other common characteristics. Such characteristics can be convenient for modelling certain kinds of problems, but make the already-difficult problem of planning even more challenging (Bresina et al., 2002). We consider probabilistic planning over competing formalisms because it offers a reasonable tradeoff between computational difficulty and modelling flexibility.

In addition to computational considerations, we are also motivated by the range of real-world applications which exist for probabilistic planning. For instance, probabilistic planning can be used to uncover the goals of other agents (Alford et al., 2015), to sched-

ule elective hospital admissions (Zhu, 2013), or to help a robot find objects in unknown locations (Trevizan and Veloso, 2013). Further application to larger or more complex problems could be enabled by new planners with higher scalability. Improvements to probabilistic planning methods could also benefit the many existing applications of classical planning, from controlling collaborative robots (Helms et al., 2002) to probing the security defences of a computer network (Obes et al., 2013).

At present, the dominant methods for probabilistic planning all perform some sort of forward-chaining search through state space (Bonet and Geffner, 2003; Hansen and Zilberstein, 2001; Keller and Eyerich, 2012). Further, two of the three dominant methods fall within the paradigm of heuristic search (Bonet and Geffner, 2003; Hansen and Zilberstein, 2001). Starting from a specified initial state, these planners slowly expand a partial graph of reachable states, with the objective of finding reliable paths from the start state to a goal state. In heuristic search planners, expansion of the partial state graph is guided by heuristics, which analyse the structure of a planning problem to estimate how costly it is to achieve an agent's goal from each state. While these search algorithms and heuristics are effective in many problems, they are seldom able to improve with experience. In some domains, this inability to learn from successes and mistakes forces planners to solve similar subproblems over and over again without ever becoming more efficient. This thesis considers one possible strategy for reducing this sort of waste by automatically learning domain-specific knowledge.

1.2 Learning for planning

Learning has already had a long history in planning, with mixed success. For instance, machine learning is often employed to analyse the characteristics of a planning problem and select the most suitable existing planner to solve it. This strategy—which is known as *autoselection*, and is sometimes combined with *autoconfiguration*—has been extremely successful at winning competitions (Coles et al., 2012; Vallati et al., 2015). However, the planners considered by autoselection systems are usually not themselves capable of learning, so autoselection does little to resolve the problem of non-learning planners having to repeatedly solve similar subproblems from scratch. On the other hand, there exist many planners which are able to learn meaningful domain-specific knowledge. This includes learning improved heuristics, learning to decompose problems into more tractable hierarchies of subproblems, and even learning to map states directly to appropriate actions (Jiménez et al., 2012). However, such planners are not yet competitive with those that merely use learnt knowledge for automatic selection or configuration of non-learning planners (Coles et al., 2012).

In parallel with developments in planning, there has recently been an explosion in the popularity of neural networks, under the banner of *deep learning*. Neural networks have long been known as powerful and flexible tools for machine learning. For much of their history, however, neural networks received only limited attention from machine learning researchers. This was in part due to issues of high computational cost, poor data efficiency, poor generalisation to unseen data, and so on (LeCun et al., 2015). The past decade has seen a dramatic reversal of fortunes for neural networks. In the last five years alone, we have seen neural networks obtain state-of-the-art results on machine translation (Wu et al., 2016), image recognition (Krizhevsky et al., 2012), learning to play Atari games (Mnih et al., 2013), and many other tasks.

The technical advances which have enabled promising results in deep learning are manifold. The large body of preexisting literature on neural network architectures for different sensing modalities—for instance, convolutional neural networks for images—has certainly been one enabling factor (LeCun et al., 2015). On the other hand, the cumulative impact of smaller innovations has also been an important factor. Generalisation of learnt knowledge has been improved using techniques like dropout (Srivastava et al., 2014) and batch normalisation (Ioffe and Szegedy, 2015). Computational burden has been alleviated by using more suitable hardware. Training time has been cut by better optimisation techniques (Kingma and Ba, 2014), and so on. The upshot is that training deep neural networks is easier than ever, and applications of neural networks have proliferated as a result.

Despite strong interest in deep learning across many subfields of AI, there has been little investigation into how neural networks could be used to improve planning. In particular, there is not yet a consensus on which neural network architectures and training techniques are appropriate for planning, or even a consensus on what planning-related tasks a neural network should be expected to perform. With that in mind, the object of this thesis is to answer the following question:

How can we use deep learning to accelerate probabilistic planning?

Our focus will be on developing a neural network architecture which is able to learn generalised policies. A generalised policy is a mapping of states to actions which can be applied to any problem belonging to a given planning domain. For example, a generalised policy for a truck routing domain could be directly applied to truck routing tasks with any number of trucks, destinations, etc. It is sometimes possible to solve very large planning problems by learning a generalised policy using only small, tractable problems from the same domain, then applying the learnt policy to a very large problem-of-interest. While generalised policies are only one possible approach to accelerating planning, the techniques we present here could be generalised to other tasks, and we suggest some options in Section 6.2. At a high level, this thesis is intended to serve as a step towards bridging the worlds of planning and neural networks by showing how deep learning techniques can fruitfully be applied to planning problems.

1.3 Contributions and structure

In the chapters which follow, we will give a detailed account of the following contributions:

1. First, we propose a novel family of neural network architectures which we call Action Schema Networks (ASNets). Each ASNet is automatically tailored to the structure of a specific probabilistic planning problem. In particular, an ASNet is organised into alternating layers of *action modules* and *proposition modules*. The action modules correspond to the actions which an agent can take, while the proposition modules correspond to the collection of binary variables (propositions) which define each state. An action module in one layer is only connected to directly related proposition modules in the previous and next layers. However, over the course of many layers, it is possible for an ASNet to use those local connections to build a rich internal representation of a state for a planning problem. We will argue that

this is a close analogue to the way that convolutional neural networks—which are popular in the computer vision community—process images.

2. Having introduced ASNets, we present a weight-sharing scheme which enables ASNet-based policies (mappings from states to actions) to generalise to probabilistic planning problems of different sizes, so long as those problems are drawn from the same domain. Our weight-sharing scheme allows proposition modules which belong to the same family of propositions to use the same parameters (i.e. weights), and likewise for action modules. This means that the number and type of parameters which need to be learnt is the same for all problems in a domain. Hence, we are able to train an ASNet on problems of one size and transfer the learnt knowledge directly to problems of any other size.

3. Finally, we introduce a scheme for efficiently training ASNets to serve as generalised policies. Our method uses a mixture of supervised learning and random exploration on small problems to teach an ASNet how to choose appropriate actions. We show that weights learnt in this way can be applied directly to larger problems, as one would expect given our use of weight-sharing. Generalisation of this kind can be viewed as a mechanism for amortising the cost of training. While it would be faster to solve small problems without an ASNet, the knowledge which an ASNet learns on small problems can be used to solve large problems which non-learning planners cannot approach.

The remainder of this thesis proceeds as follows: Chapter 2 provides the necessary planning and machine learning background to understand the chapters which follow. In particular, we define what we mean by "probabilistic planning problem", "planning domain", and so on; these definitions are essential to understanding the generalisation capabilities of ASNets. Chapter 2 also surveys related work in order to place our own work in context. Chapter 3 builds on this background to introduce ASNets, while Chapter 4 explains how to use ASNets to learn and exploit generalised policies. Chapter 5 evaluates ASNets and our proposed training algorithm on a range of challenging probabilistic planning benchmarks. Finally, Chapter 6 summarises our contributions and closes by suggesting promising avenues of future research.

Chapter 2

Background and Prior Work

The objectives of this chapter are twofold: first, we aim to give the reader a basic grounding in automated planning and deep learning, as we will draw heavily on both fields in later chapters. Second, we wish to survey relevant existing work in machine learning, automated planning, and the intersection of the two fields, thereby allowing us to put our original contributions in context later in the thesis.

2.1 Probabilistic planning

We will begin the chapter with a brief (and necessarily limited) overview of probabilistic planning. For the interested reader, Mausam and Kolobov (2012) provide an in-depth overview of probabilistic planning problem formalisms and common solution methods.

2.1.1 Formalism

Formally, we will view probabilistic planning as the task of solving a *Stochastic Shortest Path* problem (SSP), expressed compactly as a factored SSP. We will see that the notions of an SSP and factored SSP naturally lend themselves to the sorts of problems considered in the field of probabilistic planning.

SSPs

An SSP (Bertsekas and Tsitsiklis, 1996) is a tuple

$$(\mathcal{S}, \mathcal{A}, \mathcal{T}, \mathcal{C}, \mathcal{G}, s_0) \,, \tag{2.1}$$

where $\mathcal{S}$ is a finite set of states, $\mathcal{A}$ is a finite set of actions, s_0 is an initial state, and $\mathcal{G} \subseteq \mathcal{S}$ is a set of goal states. $\mathcal{T} : \mathcal{S} \times \mathcal{S} \times \mathcal{A} \to [0, 1]$ is a probability distribution, where $\mathcal{T}(s' \mid s, a)$ gives the chance of transitioning from state s to state s' after taking action a. The cost function $\mathcal{C} : \mathcal{S} \times \mathcal{A} \to (0, \infty)$ gives the cost of applying a particular action in a given state. An agent's objective in an SSP is to select a sequence of actions getting it from s_0 to some state in $\mathcal{G}$, typically with the lowest expected cost over the trajectory of visited states.

Formally, a solution to an SSP is a *policy* $\pi : \mathcal{A} \times \mathcal{S} \to [0, 1]$, where $\pi(a \mid s)$ is the probability of an agent applying action a in state s. Each policy π has a cost-to-go function $V^\pi : \mathcal{S} \to [0, \infty)$, defined as

$$V^\pi(s) = \begin{cases} \sum_{a \in \mathcal{A}} \pi(a \mid s) \left[\mathcal{C}(s, a) + \sum_{s' \in \mathcal{S}} \mathcal{T}(s, a, s') \cdot V^\pi(s') \right] & \text{if } s \notin \mathcal{G} \\ 0 & \text{if } s \in \mathcal{G} \end{cases} . \tag{2.2}$$

We can also define a Q-value $Q^\pi : \mathcal{S} \times \mathcal{A} \to [0, \infty)$, where

$$Q^\pi(s, a) = \mathcal{C}(s, a) + \sum_{s'} \mathcal{T}(s' \mid s, a) \cdot V^\pi(s') \,. \tag{2.3}$$

A policy π^* which minimises the expected cost-to-go $V^\pi(s)$ among all policies π is said to be *optimal*. We refer to this minimal cost-to-go using the shorthand $V^*(s)$. By treating $V^*(s)$ as an $|S|$-dimensional vector of costs-to-go, we can equivalently define it as the solution a *Bellman equation*,

$$V^*(s) = \begin{cases} \min_{a \in \mathcal{A}} \left[\mathcal{C}(a) + \sum_{s' \in \mathcal{S}} \mathcal{T}(s, a, s') \cdot V^*(s') \right] & \text{if } s \notin \mathcal{G} \\ 0 & \text{if } s \in \mathcal{G} \end{cases} \quad \text{for all } s \in \mathcal{S} \,. \tag{2.4}$$

It's worth pointing out that this minimal cost-to-go $V^*(s)$ can always be achieved by a deterministic policy. That is, there is an optimal policy where $\pi(a \mid s) \in \{0, 1\}$ for every action a and state s reachable under π. In later chapters, however, we will find it convenient to instead talk about stochastic policies, particularly in the context of obtaining a neural network representation of a policy.

To ensure that $V^*(s)$ is well-defined and finite for every state reachable from s_0, Bertsekas and Tsitsiklis (1996) assume SSPs are structured in such a way that a *proper* policy is guaranteed to exist. Specifically, a proper policy is one in which every state $s \in \mathcal{S}$ is connected to some state $s_g \in \mathcal{G}$ by a sequence of state transitions which each have nonzero probability. Under a proper policy, there are no *dead ends* from which a goal state can never be reached. In real problems, however, is common to encounter problems where dead ends *do* exist. Rather than assuming the existence of a proper policy, we will thus augment the definition of an SSP to include a dead-end penalty $D \in (0, \infty)$. D serves as a limit on the expected cost-to-go of a state, so that

$$V^\pi(s) = \min \left\{ D, \sum_{a \in \mathcal{A}} \pi(a \mid s) \cdot Q^\pi(s, a) \right\} \,. \tag{2.5}$$

This limit allows an agent to give up on reaching the goal from a dead end, rather than incurring a potentially infinite sequence of costs. Formally, this model is referred to a finite penalty Stochastic Shortest Path Problem with Unavoidable Dead Ends (fSSPUDE) (Kolobov et al., 2012b).

Readers familiar with Markov Decision Processes (MDPs) may notice that the definition of an SSP is similar to the definition of an MDP. The main difference the notion of a "goal" state, which an MDP does not have. Another difference is the use of $V(s)$ to refer to expected cost-to-go to reach the goal (which one attempts to minimise), rather than expected reward (which one attempts to minimise). Despite these differences, both formalisms are commonly used for modelling decision-making under uncertainty. However, SSPs subsume both finite-horizon MDPs and infinite-horizon MDPs (Mausam and Kolobov, 2012), and are arguably a more natural way of expressing the sorts of goal-driven problems studied by the planning community. It is for this reason that we will use SSP-style notation and terminology throughout the remainder of this thesis.

It's also worth noting the connections between solving an SSP and doing *reinforcement learning* (Sutton and Barto, 1998). Reinforcement Learning (RL) is typically cast as the task of solving an MDP where the states and actions are known, but the transition dynamics are not. Most RL algorithms work by executing a learnt policy many times

and improving the policy in response to the observations made and rewards received. In particular, after recording a state trajectory with relatively high reward, the policy can be adjusted to make the actions which produced that trajectory more likely in the future. Conversely, after recording a trajectory with low reward, the policy can be adjusted to make the trajectory less likely. Effective reinforcement learning usually requires some form of planning; some RL algorithms thus attempt to learn the transition dynamics of an environment as an MDP or SSP, then solve that model using a traditional probabilistic planner. We will not investigate RL deeply in this thesis, but will attempt to contrast our work with relevant literature on reinforcement learning.

Factored SSPs

For large probabilistic planning problems, manually specifying a state space $\mathcal{S}$ and goal set $\mathcal{G}$ can be tedious. For instance, consider the problem of rotating a combination lock with n wheels to its correct setting. If each wheel has 10 possible positions, then there are 10^n possible codes for the lock, and thus 10^n possible states in the problem. Specifying each state by hand would be infeasible even for relatively small n. Likewise, if our goal was to move only a fixed subset of k wheels into set positions, then $\mathcal{G}$ would have to include 10^{n-k} states: one for each setting for the remaining (unspecified) $n-k$ wheels.

In practice, it is much easier to specify problems of this nature as *factored SSPs*. Formally, a factored SSP is a tuple

$$(\mathcal{X}, \mathcal{A}, \mathcal{T}, \mathcal{C}, \mathcal{G}, s_0)\ , \tag{2.6}$$

where $\mathcal{X}$ is a set of n propositions (binary variables), $\mathcal{A}$ is a set of actions, $\mathcal{T}$ is a transition function, $\mathcal{C}$ is a cost function, $\mathcal{G}$ is a set of goal states, and s_0 is an initial state. In a factored SSP, the state space $\mathcal{S} = \{\top, \bot\}^n$ is simply the set of assignments of truth values to the variables in $\mathcal{X}$. Furthermore, $\mathcal{A}$, $\mathcal{T}$, $\mathcal{C}$, and $\mathcal{G}$ can be specified compactly using expressions over the variables in $\mathcal{X}$, instead of having to be stored in tabular form. We will see how this can be achieved by examining $\mathcal{G}$, $\mathcal{A}$, $\mathcal{C}$ and $\mathcal{T}$ in turn.

First, consider the goal $\mathcal{G}$. Rather than being explicitly specified, the set of goal states $\mathcal{G}$ is implicitly defined as $\mathcal{G} = \{s \in \mathcal{S} \mid s \models c_G\}$. In this case, c_G is a logical formula over the variables $\mathcal{X}$, and $s \models c_G$ iff the all variables satisfy that formula in the state s. The formula c_G is constructed using ordinary propositional logic. The simplest formulae are positive atoms: the formula x is true iff $x \in \mathcal{X}$ is true in the current state. More complex formulae can be defined recursively through the negation ($\neg c$), conjunction ($c_1 \wedge \cdots \wedge c_C$), and disjunction ($c_1 \vee \cdots \vee c_C$) of conditions.

Each action $a \in \mathcal{A}$ is composed of a precondition $\mathrm{pre}(a)$, an effect $\mathrm{eff}(a)$, and a cost $\mathcal{C}(a) \in \mathbb{R}$. The precondition $\mathrm{pre}(a)$ is a logical formula over $\mathcal{X}$—just like the goal condition c_G—which determines whether a can be applied. If $s \not\models \mathrm{pre}(a)$, then we prevent the agent from executing the action a in state s. The effect $\mathrm{eff}(a)$ specifies how a manipulates values of the state variables in $\mathcal{X}$ when it is applied. The most basic effects are those which make a variable x_i true or false; these effects are denoted x_i and $\neg x_i$, respectively. More complex effects can be defined recursively. Specifically, if $e_1, \ldots, e_E$ is a list of effects, then

- A conjunctive effect $e_1 \wedge \ldots \wedge e_E$ applies all effects $e_1, \ldots, e_E$. We assume that these $e_1, \ldots, e_N$ are *consistent*: in particular, it should never be the case that two effects each assign different values to a variable. This assumption allows effects in a conjunction to be safely applied in any order.

- A probabilistic effect $p_1 e_1 | \cdots | p_n e_n$ chooses and applies a single effect e_i using the probability distribution defined by $p_1, \ldots, p_n \in [0, 1]$.
- A conditional effect $c \rhd e_i$ applies the effect e_i when the condition c is true in the current state. Otherwise, it does nothing.

Applying an effect $\mathrm{eff}(a)$ in a state s could lead the agent to transition into one of several possible successor states s', depending on the outcome of each probabilistic subeffect in $\mathrm{eff}(a)$.

Armed with a definition of the action set $\mathcal{A}$ and the state space $\mathcal{S}$, the transition function $\mathcal{T} : \mathcal{S} \times \mathcal{S} \times \mathcal{A} \to [0, 1]$ is straightforward to define. When $\mathrm{pre}(a)$ holds in s, we define $\mathcal{T}(s' \mid s, a)$ to be the probability that applying $\mathrm{eff}(a)$ in s will yield s'. This probability can be calculated by performing inference on a specially constructed dynamic Bayesian network for the action (Younes and Littman, 2004). When $\mathrm{pre}(a)$ does not hold in s, $\mathcal{T}(s' \mid s, a)$ is undefined. Note that the SSP framework put forward above does not have a notion of inapplicable actions or undefined transition probabilities. In principle, we could extend the SSP framework by making each inapplicable action transition the agent into an artificial dead-end state. In practice, the presence of inapplicable actions does not pose a significant practical or theoretical barrier (Mausam and Kolobov, 2012), so we will not mention this discrepancy for the remainder of the thesis except where relevant.

Lifted SSPs

Factored SSPs often fall into families with similar actions and variables. For instance, two navigation tasks might feature a different selection of locations which an agent can navigate between, and a different goal to be obtained. However, at an abstract level, both tasks will have actions to move from one location to another, variables to indicate whether the agent is at a given location, and so on. We can capture this common structure with a *Lifted SSP*. Concretely, a lifted SSP is a tuple

$$(\mathbb{P}, \mathbb{A}, \mathcal{C}) \,, \tag{2.7}$$

where $\mathbb{P}$ is a finite set of *predicates*, $\mathbb{A}$ is a set of *action schemas*, and $\mathcal{C}$ is a cost function. A predicate can be thought of as a function mapping objects to propositions, and an action schema can considered as a mapping from objects to actions. These basic ingredients can be combined with *objects*—symbols representing entities in an environment—to produce a set of actions and propositions. Those objects and propositions can then be combined with a goal $\mathcal{G}$ and start state s_0 to produce a factored SSP.

To make the relation between lifted and factored SSPs concrete, consider the objects shakey, hall, $\mathit{kitchen}$. If we have an action schema $\mathrm{drive}(?\mathit{robot}, ?\mathit{from}, ?\mathit{to})$, then we could apply it to our three objects in order to obtain an action $\mathrm{drive}(\mathit{shakey}, \mathit{hall}, \mathit{kitchen})$. Similarly, if we wanted a proposition to indicate that shakey is in the hall, then we could apply a predicate $\text{robot-at}(?\mathit{robot}, ?\mathit{location})$ to the objects to obtain a proposition $\text{robot-at}(\mathit{shakey}, \mathit{hall})$.

Grounding is the process by which a lifted SSP is turned into a factored SSP through application of action schemas and predicates to objects. The inputs to the grounding process are a lifted SSP and a set of objects. From these objects, a set of propositions for the factored SSP can be systematically enumerated by applying each predicate to each applicable collection of objects. As there are $n!/(n-k)!$ ways to apply a predicate (or

action schema) with arity k to n objects, naive grounding can produce a very large set of actions and propositions in some instances. We will see in Section 2.1.2 that it is often possible to be more selective when deciding which propositions and actions to generate, thus leading to more compact factored SSPs.

2.1.2 PPDDL

In practice, it is common to specify factored SSPs using the Probabilistic Planning Domain Definition Language (PPDDL) (Younes and Littman, 2004). PPDDL is a probabilistic extension of the Planning Domain Definition Language (PDDL) (McDermott et al., 1998), which is itself popular in the deterministic planning community. PPDDL splits factored SSP definitions into a *domain* and a *problem*. A PPDDL domain specifies a lifted SSP $(\mathbb{P}, \mathbb{A}, \mathcal{C})$, while a problem specifies a set of objects $\mathcal{O}$, an initial state, and a goal condition. Each problem also includes a reference to a domain, which can be used to generate the actions and propositions of a factored SSP.

In addition to the lifted SSP components defined in Section 2.1.1, PPDDL domains typically specify a hierarchy of *types* for objects. Each user-specified type may inherit from a single other type, and those types which do not inherit from any user-specified type inherit from a root type called `object`. The parameters of action schemas and predicates in a PPDDL domain can be annotated with these types. Each object in a corresponding PPDDL problem can also be given a type. During grounding, these types can be used to determine which predicates and action schemas can be applied to which objects. Returning to our previous example, the predicate $\text{robot-at}(?r, ?p)$ might stipulate that $?r$ should be of type `robot`, and $?p$ of type `place`. In a problem definition, we could then mark objects representing locations as having type `place`, preventing nonsensical propositions like $\text{robot-at}(\mathit{kitchen}, \mathit{office})$. A similar effect can be achieved with unary predicates, but types make modelling more straightforward, and also free the planner from having to deal with additional propositions after grounding.

As well as using type annotations, planners which perform PPDDL grounding typically minimise the number of actions and propositions which they generate by exploiting domain structure and a *closed-world assumption*. In general, the closed-world assumption stipulates that propositions which are not specified must be false. PPDDL makes this assumption, so the initial state s_0 is normally only specified as a list of propositions which are true initially, with the remainder of propositions assumed to be false. During grounding, planners begin by generating only those propositions which appear in the initial state or the goal condition c_G. They then use various approximations to build a set of actions which it could be possible to apply at some point, as well as a set of propositions which could potentially be made true. While different planners use different mechanisms for doing this approximation, the common end-result is that grounded PPDDL problems are often have far fewer ground actions and propositions than a naive analysis of objects, predicates and action schemas might imply. As an example, consider a predicate $\text{path}(?\mathit{from}, ?\mathit{to})$ which is used to model problems in which an agent can navigate throughout a static network of locations. Assume that there is a corresponding $\text{drive}(?\mathit{from}, ?\mathit{to})$ action which can move the agent between those connected locations. Say that proposition $\text{path}(\mathit{office}, \mathit{kitchen})$ is not true initially, and there is no action which can add that proposition. This means that $\text{path}(\mathit{office}, \mathit{kitchen})$ can be assumed to be false, and never has to be generated. Further, it will never be possible to apply the action $\text{drive}(\mathit{office}, \mathit{kitchen})$, so it does not have to be generated either. This strategy can mas-

```
(define (domain unreliable-robot-domain)
  (:requirements :typing :probabilistic-effects)
  ;; 'robot' and 'place' are subtypes of 'object'
  (:types robot place - object)
  (:predicates
    ;; ?r should be an object of type 'robot'
    ;; ?l, ?from, and ?to should be of type 'place'
    (robot-at ?r - robot ?l - place) (path ?from ?to - place))
  (:action drive
    :parameters (?r - robot ?from ?to - place)
    :precondition (and (robot-at ?r ?from) (path ?from ?to))
    :effect (probabilistic
      ;; moves from ?from to ?to 90% of the time;
      ;; 10% of the time, does nothing
      9/10 (and (robot-at ?r ?to)
                (not (robot-at ?r ?from))))))
```

Figure 2.1: A trivial domain to illustrate the capabilities of PPDDL

sively decrease the number of actions and propositions which need to be generated for a given problem.

To make the ideas behind PPDDL concrete, Figure 2.1 shows a PPDDL domain for a simple robotic navigation task, and Figure 2.2 shows a problem corresponding to that domain. The domain models a robot which is able to move between locations using a drive action schema, albeit one which fails to move the robot 10% of the time. The key thing to note in Figure 2.1 is the correspondence between the PPDDL snippet and the lifted SSP definition from Section 2.1.1. In this case, the action schema set $\mathbb{A}$—defined by `:action` directives—consists only $\{\text{drive}(?r, ?from, ?to)\}$, while the predicate set is $\mathbb{P} = \{\text{path}(?from, ?to), \text{robot-at}(?r, ?l)\}$. `:action` directives implicitly define a cost function $\mathcal{C}(a) = 1$ for all actions a; non-unit action costs could be achieved with numeric effects that manipulate a special reward variable, but we do not consider such costs here. In the problem definition, the `:object` directive is used to declare an object set $\mathcal{O}$, and the `:goal` directive to declare a goal condition c_G. Importantly, the initial state declaration (`:init`) includes only those propositions which are true in s_0—all others are assumed to be false, per the closed world assumption. This domain is not yet very interesting from a planning perspective, it will serve as a sound foundation for more complex illustrations in later chapters.

It should be noted that PPDDL has a range of functionality not covered here. For instance, in addition to predicates, PPDDL can model *functions* which map from combinations of objects to numbers. The grounding process can use functions to produce numeric state variables; those state variables can then be manipulated in effects and referenced in conditions. Further, conditions in PPDDL are able to make use of universal and existential quantification over sets of objects, and perform equality checks between objects (e.g. is the object *sydney* the same as the object *melbourne*?). While we do not consider support for these more advanced features in this thesis, we believe that extension of the methods presented here to support more advanced PPDDL features should be straightforward.

```
(define (problem unreliable-robot-problem)
  (:requirements :typing :probabilistic-effects)
  ;; use lifted SSP declared in 'unreliable-robot-domain'
  (:domain unreliable-robot-domain)
  ;; kitchen, hall, and the offices are of type 'place'
  ;; shakey is of type 'robot'
  (:objects kitchen hall office-1 office-2 - place
            shakey - robot)
  ;; initial state
  (:init (and
    ;; shakey starts in kitchen
    (robot-at shakey kitchen)
    ;; can travel from hall to anywhere, from kitchen to
    ;; hall, and from office to office
    (path kitchen hall) (path hall kitchen)
    (path office-1 hall) (path hall office-1)
    (path office-2 hall) (path hall office-2)
    (path office-1 office-2) (path office-2 office-1))
  ;; objective is to go to second office
  (:goal (robot-at shakey office-2))))
```

Figure 2.2: A demonstration problem to complement the domain in Figure 2.1

2.1.3 Heuristic search planners

For fully-observable deterministic and probabilistic planning, most state-of-the-art planners use *heuristic search*. Indeed, the winners of the probabilistic and deterministic tracks in the two most recent International Planning Competitions (2011 and 2014) have been based on heuristic search (Coles et al., 2012; Vallati et al., 2015). Broadly, planners from this family work by gradually constructing a graph of state transitions, starting from the initial state s_0 and working outwards. The algorithm chooses a direction in which to expand the graph using a *heuristic* $h : \mathcal{S} \to \mathbb{R}$, which approximates the expected cost to achieve the goal from a state. This allows the planner to expend more effort expanding portions of the state space which look most promising. In this thesis, we will use heuristic search planners both as baselines in experiments in Chapter 5, and to train the learning planner which we present in Chapter 4. We will begin our discussion of heuristic search by examining *Policy Iteration* (PI) and *Value Iteration* (VI). PI and VI are not heuristic search methods themselves, but will allow us to introduce LRTDP and LAO*, which are heuristic search methods that build on PI and VI.

VI maintains a table containing a value $V^\pi(s)$ for each state $s \in \mathcal{S}$. On each iteration, each entry $V^\pi(s)$ is updated by applying a *Bellman backup*,

$$V^\pi_{t+1}(s) \leftarrow \min\left\{D, \min_a \left[C(s,a) + \sum_{s' \in \mathcal{S}} \mathcal{T}(s' \mid s, a) \cdot V^\pi_t(s')\right]\right\}, \tag{2.8}$$

to each $s \in \mathcal{S}$. Here, D is the dead-end penalty for the fSSPUDE, and the subscript t is used to make it clear that values for one "sweep" of the state space are computed based only on values from the previous sweep. Once a fixed point is reached, VI's policy is to always choose the action a which maximises $Q^\pi(s,a)$, breaking ties arbitrarily. Importantly,

VI's eventual convergence to an optimal value function (and thus an optimal policy) for an fSSPUDE is guaranteed regardless of the initial value $V_0^\pi(s)$ of each state (Kolobov et al., 2012b).

In contrast to VI's strategy of maintaining and periodically updating a table of value functions, PI instead maintains and periodically updates a policy π. At each iteration, it first computes the value $V^\pi(s)$ of the policy in each state, and then updates the policy with

$$\pi(s) \leftarrow \max_a Q^\pi(s, a) \,, \tag{2.9}$$

where $\pi(s) = a$ is a slight abuse of notation to indicate that action a is chosen in state s with probability 1. PI converges to an optimal policy in a finite number of iterations, provided that it is initialised with a proper policy (Mausam and Kolobov, 2012).

Real Time Dynamic Programming (RTDP) is a heuristic search planner which, like VI, plans by gradually improving an internal table of state values. RTDP updates its state value table by performing "trials": on each trial, RTDP starts at $s \leftarrow s_0$, then chooses an action which maximises $Q^\pi(s, a)$ according to some internal table of state values. It then performs a Bellman backup on s, samples a successor state $s' \sim T(s' \mid s, a)$, and updates $s \leftarrow s'$. A trial terminates once RTDP reaches a goal state or, in the fSSPUDE case, a dead end. If a state s has not previously been visited, its value is initialised to $V(s) \leftarrow h(s)$. RTDP only performs backups on states visited through trials, and uses a heuristic $h(s)$ to initialise values for states when they are first visited. This allows RTDP to avoid enumerating every state of the problem before beginning to plan—it can simply add entries for newly visited states to its internal value table on the fly. RTDP's trial mechanism often manages to find a reasonable approximation of $V^*(s)$—and thus a reasonable policy—even when time constraints force it to be terminated before convergence (Barto et al., 1995). In practice, it is common to pair RTDP with a mechanism to detect when states' values have converged, yielding Labelled RTDP (LRTDP). This allows the algorithm to terminate early in some cases, and to avoid backing up states whose values have already converged (Bonet and Geffner, 2003).

LAO* is also a heuristic search algorithm, but operates in a very different manner to (L)RTDP. Rather than building up a table of state values, LAO* builds up an *explicit graph* G of states rooted at s_0. LAO* also maintains subgraph of G known as a *best partial solution graph* G_{s_0}. G_{s_0} contains the states which can be visited by an optimal policy for the explicit graph, under the simplifying assumption that leaf state in the graph has an expected cost-to-goal of precisely $h(s)$. At each iteration, LAO* chooses a single non-goal state s from the leaves of the best partial solution graph G_{s_0}, and adds its children to the explicit graph G. It then performs value iteration or policy iteration on the newly expanded best partial solution graph to obtain an updated optimal policy for the explicit graph, and thus an updated best partial solution graph. LAO* terminates once the leaves of G_{s_0} are all goal states. At this point, the best partial solution graph encodes a *closed* policy π for the original problem, in the sense that $\pi(a \mid s)$ is defined for all s reachable from s_0 by following π. In the original paper proposing LAO*, Hansen and Zilberstein (2001) suggest a faster variant of LAO* which adds more states to G_{s_0} at each iteration and does not always optimise the policy for G_{s_0} to convergence. We use this improved variant—sometimes known as *Improved LAO** (ILAO*)—in experiments.

While LRTDP and LAO* are popular heuristic search approaches to probabilistic planning, there are a range of other heuristic search approaches, too. We will close this section by pointing to one particularly relevant class of probabilistic planners: those

which perform *fixed-depth lookahead* (also known as *finite-horizon search*) in state space. Broadly, these planners solve SSPs by relaxing them so that only a fixed number of actions can be taken starting at a given state. The resulting search tree (or graph, in the case of planners which can accommodate loops in state space) may be much shallower than that of the original SSP, thus allowing a fixed-horizon planner to choose an action quickly. Notable fixed-depth lookahead planners include SSiPP (Trevizan and Veloso, 2012), which treats a single large SSP as a series of smaller *short sighted SSPs*, and PROST (Keller and Eyerich, 2012), which uses a form of fixed-depth Monte Carlo tree search to choose actions. In choosing a depth for lookahead, these planners must make a tradeoff: too small a depth could hide traps which lie beyond a depth-limited agents planning horizon.[1] Conversely, the cost of producing a policy for a depth-limited SSP grows rapidly as the depth increases, so too great a depth could prevent the planner from converging to a good action within a reasonable time frame (Kolobov et al., 2012a). The neural network architecture which we propose in Chapter 3 chooses actions in a manner which is reminiscent of, but distinct from, fixed-depth lookahead in state space. We will further touch on the differences between our work and existing fixed-depth lookahead planners in Section 3.3, and as part of the empirical evaluation in Chapter 5.

2.1.4 Heuristics

The strength of a heuristic state space planner rests on the quality of its heuristic. Heuristics for deterministic planning are a well-studied area: the typical approach is to weaken the semantics of the original planning problem to make it easier to solve. This *relaxed* problem is then solved from the current state s, and the length (or approximate length) of the solution is used as the heuristic value $h(s)$. Heuristics derived from relaxations have a convenient property: if the relaxation is solved optimally, then the cost of the solution must satisfy $h(s) \leq V^*(s)$. In other words, the heuristic is *admissible*. Many heuristic search algorithms are guaranteed to converge to an optimal policy when equipped with an admissible heuristic, but not necessarily guaranteed to converge to an optimal policy when the heuristic is inadmissible. Algorithms with this property include LRTDP and LAO* in probabilistic planning, and A* (Russell and Norvig, 1995) in deterministic planning. The downside of using admissible heuristics is that they are often less informative than inadmissible ones. Hence, they may take longer to find a solution, even though the resulting solution may be less costly than a solution uncovered by an inadmissible heuristic. In this section, we will cover several relevant admissible and inadmissible heuristics for deterministic problems, then explain how these heuristics can be applied to probabilistic problems using *determinisation*.

In a deterministic setting, it is common to derive heuristics using a *delete relaxation*.[2] Rather than restricting propositions to be true or false, a delete-relaxed problem allows propositions to be both true and false at the same time. Once a proposition is made false,

[1] We should note that, for some fixed-depth lookahead approaches, it is still possible to avoid traps beyond the lookahead depth. For instance, SSiPP is able to obtain an optimal policy for a complete SSP by repeatedly solving many short-sighted SSPs rooted at different states in the original SSP (Trevizan and Veloso, 2012). However, converging to a policy for the original (non-depth-limited) SSP can still be expensive in many domains.

[2] Strictly speaking, the relaxation which we are describing is a *monotonic relaxation*, or an instance of *value accumulation semantics*. "Delete relaxation" technically describes a relaxation of STRIPS-style planning problems—as described later in the section—rather than general PDDL-style problems. However, we use the term "delete relaxation" over "monotonic relaxation" because the former appears to be more common in the planning literature.

it can be treated as if it false in all subsequent states, even if an action is later executed which would make the proposition true under non-relaxed semantics. Likewise, once a proposition is made true, it can be treated as if it's true in all later states, even if an action is executed which would make it false under non-relaxed semantics. A condition in a delete-relaxed problem may reference variables which can be either true or false. In such a situation, the condition is assumed to hold if there is some assignment of truth values to the ambiguous variables which makes the original condition hold under non-relaxed semantics. These relaxed semantics apply to action preconditions, conditions in conditional effects, and the goal condition. Hence, a delete relaxation typically under-approximates the distance-to-the-goal and over-approximates the set of actions which can be applied at a given point.

The most obvious approach to employing a delete relaxation is to solve a delete-relaxed problem optimally, then use the length of the resulting plan as a heuristic value. The resulting heuristic is sometimes referred to as h^+. Unfortunately, h^+ is NP-hard to compute in general (Bylander, 1994), so we must resort to approximations of h^+ to derive a practical heuristic.

$h^{\max}$ is one simple admissible approximation of h^+ (Haslum and Geffner, 2000). $h^{\max}$ is easiest to define for STRIPS problems (Fikes and Nilsson, 1971): in this setting, each state s is interpreted as a set of propositions which is true, and each action is interpreted as an operation on this set of propositions. Effects are restricted to conjunctions of literals (i.e. propositions and negations of propositions). We split an effect $\mathrm{eff}(a)$ into a set of propositions $\mathrm{eff}^+(a)$ to be "added", which appear un-negated in $\mathrm{eff}(a)$, and a set of propositions $\mathrm{eff}^-(a)$ to be "deleted", which instead appear negated. Conditions—including each precondition $\mathrm{pre}(a)$, and the goal condition c_G—are restricted to conjunctions of propositions, with no use of negation, disjunction, etc. Because STRIPS treats states and conditions as sets of propositions, we can abuse notation slightly and use $c \subseteq s$ to indicate that condition c holds in state s, $p \in c$ to denote that proposition p appears in condition c, $|c|$ to indicate the number of propositions in a conjunctive condition, etc.

Using the above set notation, $h^{\max}(s, c)$ for some conjunction of propositions c can be defined as

$$h^{\max}(s,c) = \begin{cases} 0 & \text{if } c \subseteq s\ , \\ \max_{p \in c} h^{\max}(s, \{p\}) & \text{if } |c| > 1\ , \\ \min_{a \in \mathcal{A}, p \in \mathrm{eff}^+(a)} [h^{\max}(s, \mathrm{pre}(a)) + \mathcal{C}(s,a)] & \text{if } c = \{p\}\ , \end{cases} \tag{2.10}$$

where the $\min$ in the final branch is ∞ if there is no action which adds p. The resulting heuristic is simply $h^{\max}(s) = h^{\max}(s, c_G)$ for the goal condition c_G. Replacing the $\max$ in the above definition with a summation over the costs of each precondition yields a heuristic known as h^{add}. h^{add} is inadmissible, but often yields better guidance than $h^{\max}$; we will consider both heuristics in Chapter 5.

The landmark cut (LM-cut) heuristic of Helmert and Domshlak (2009) is a more sophisticated approximation to the optimal delete-relaxed heuristic h^+. We will omit a full explanation of how LM-cut is computed. From the perspective of heuristic search planning, the key point is that LM-cut is a much closer admissible approximation of h^+ than $h^{\max}$ is, and thus provides better guidance in practice. In later chapters, we will be more interested in *how* LM-cut is computed: the "landmark" in "landmark cut" refers to the fact that LM-cut attempts to find a series of *disjunctive action landmarks* $L_1, \ldots, L_K$ for the delete-relaxed planning problem. In general, a set of actions $L_i \subseteq \mathcal{A}$ forms a disjunctive

action landmark for a planning problem if at least one $a \in L_i$ needs to be applied in any solution to the problem. If each action has constant action-specific (but not state-specific) cost $\mathcal{C}(a)$, then the landmarks $L_1, \ldots, L_K$ can be used to derive a heuristic

$$h(s) = \min_{A' \subseteq A \wedge \forall i: L_i \cap A' \neq \varnothing} \sum_{a \in A'} \mathcal{C}(a) \,. \tag{2.11}$$

We note that Equation (2.11) is only one possible way of deriving a heuristic from landmarks, and that LM-cut heuristic uses a slightly different strategy to compute heuristic values. Not only will we use LM-cut landmarks to compute heuristic values for baseline planners, but we will also use the landmarks as input features to a neural network, as described in Section 3.3.

The above heuristics are designed for deterministic problems, and cannot be applied directly to probabilistic problems. Instead, it is common to relax probabilistic problems through all-outcomes determinisation. Consider a stochastic action $a \in \mathcal{A}$ which could apply one of K distinct, deterministic effects $e_1, \ldots, e_K$, depending on the outcomes of each probabilistic choice made in $\text{eff}(a)$. In the determinised version of the problem, we replace a with a set of deterministic actions $a_1, \ldots, a_K$, where each a_i applies only the deterministic effect e_i. This can be interpreted as allowing the planner to ignore the stochasticity of actions and instead pick the outcome which is most convenient. Applying a deterministic heuristic to the determinised problem often yields a reasonable approximation of the cost-to-go in the original probabilistic problem. However, there are still many probabilistic problems in which determinisation leads to dramatic underestimation of the true cost-to-go (Little and Thiébaux, 2007). It should also be noted that determinisation is no longer the only competitive strategy for devising admissible heuristics for probabilistic planning problems. In recent work, Trevizan et al. (2017) propose heuristics based on relaxations of linear program formulations of probabilistic planning, and show that these heuristics often yield better planning performance than determinising heuristics. The methods presented in this thesis could easily be extended to make use of these sorts of new probabilistic heuristics, although for ease-of-implementation we will only make use of $h^{\max}$, h^{add}, and LM-cut.

2.2 Machine learning for automated planning

While the planners and heuristics described in Section 2.1 are often quite effective, they are not able to improve with experience. We will now consider existing work on applying machine learning techniques to the task of *speed-up learning*. Speed-up learning allows planners to use past experience to avoid having to rediscover tricks or avoid traps which are specific to a domain. In this section, we will begin by describing the broad approaches to learning-for-planning which dominate the literature. We will then investigate of the representations used for learnt knowledge, the ways in which knowledge can be acquired, and finally the ways in which learnt knowledge can be exploited in practice.

2.2.1 Approaches

We will first consider the most common ways that learning can be used to accelerate planning. Specifically, we will classify different approaches to speed-up learning using an extension of the scheme of Jiménez et al. (2012):

Macro actions A *macro action* is a sequence of actions from a planning problem which have been joined together to form one combined action. A well-chosen set of macro actions can increase planning performance by allowing a planner to jump ahead several steps at a time in state space. On the other hand, a poorly chosen set of macro actions can slow down planning by needlessly increasing the branching factor of heuristic search. It is possible to learn macro actions from small training problems by applying statistical analysis on plans produced by a non-learning planner (Muise et al., 2009). It is also possible to arrive at an effective set of macro actions using genetic programming (Newton et al., 2007), or a mixture of static analysis of the domain and statistical techniques (Botea et al., 2005).

Decompositions A *decomposition* breaks a planning problem down into a hierarchy of simpler subproblems. The objective is to ensure that the cumulative difficulty of solving each subproblem is lower than the cost of solving the original planning problem. While macro actions can be treated as a rudimentary kind of decomposition, it is more common to formulate decompositions as *Hierarhical Task Networks* (HTNs) or *options*. Unlike macro actions, HTNs and options both allow for specification of subtasks in terms of subgoals to be achieved, rather than merely sequences of actions to be executed. There is some existing work on learning HTNs from specifications and plans for deterministic problems (Georgievski and Aiello, 2015), and on learning options from execution traces in an MDP (Stolle and Precup, 2002).

Unsolvability In practice, many planning problems are either unsolvable, or contain unsolvable subproblems (e.g. dead ends) which must be avoided (Krajňanskỳ et al., 2014). Recent work has looked at augmenting heuristic search planners with the capability to learn logical formulae describing unsolvable states. These formulae can be used to avoid repeatedly entering, exploring, and backtracking out of regions of state space from which the goal is unreachable (Krajňanskỳ et al., 2014; Steinmetz and Hoffmann, 2017; Steinmetz et al., 2017).

Autoselection and autoconfiguration There are many planners which are well-suited to a particular kind of task for which other planners perform poorly, but which are not uniformly better than other planners on all common problems. Hence, in planning competitions, it is common to employ a portfolio of different planners, or a portfolio of similar planners with different configurations (Coles et al., 2012; Vallati et al., 2015). By selecting only the planner which is best-suited for a given task, portfolio planners can ensure that performance remains high across a broad range of test problems. There exist several systems which can learn to select appropriate planners by using features of a planning problem under consideration (Lindauer et al., 2015; Seipp et al., 2014; Virseda et al., 2014).

Generalised heuristics A generalised heuristic is able to estimate the cost-to-go for any problem in a given domain—that is, for any factored SSP instantiated from a specific lifted SSP. Typically, generalised heuristics are learnt with small problems from a given domain; the learnt knowledge can then be transferred to larger, more difficult problems. Previous approaches to this problem have included learning a linear function to correct estimates from a delete-relaxed heuristic (Yoon et al., 2006b), as well as an extension of the same approach to learn corrections which are well-suited to beam search (Xu et al., 2007).

Generalised policies As the name suggests, a generalised policy is one which can be applied to any planning problem from a given domain. Techniques from this family include ROLLER (de la Rosa et al., 2008, 2011), which learns policies for deterministic problems from (state, action) pairs, and the work of Yoon et al. (2002) on generalised policies in MDPs.

In this thesis, we are interested in methods which allow knowledge learnt on one planning problem to be applied to any problem from the same domain, and in particular in generalised policies. Hence, in subsequent subsections, we will explore aspects of the above learning strategies which are relevant to learning generalised policies.

2.2.2 Knowledge representations

We will first examine the ways in which existing learning-based planners represent observations and learnt knowledge. In particular, we are interested in the way in which states are encoded for the learning system—usually the state representation includes the values of some or all propositions in a problem, but some learning systems also make use of heuristic information to improve their representational power. We are also interested in the models used to transform encoded states into a predicted action or heuristic value.

Khardon (1999) presented one of the first approaches to learning generalised policies, and their choice of input representation and model has been influential in later work. Khardon represents learnt knowledge with a collection of action selection rules, each taking the form

$$\text{pred-1}(?o_{1,1}, ?o_{1,2}, \ldots) \wedge \cdots \wedge \text{pred-n}(?o_{n,1}, ?o_{n,2}, \ldots) \rightarrow \text{schema}(?o_{s,1}, ?o_{s,2}, \ldots)\,. \quad (2.12)$$

If the parameters $?o_{i,j}$ (for all i, j) can be replaced with objects in such a way that the conjunction on the left holds, then those same objects are used to instantiate the action schema on the right, and the corresponding ground action is subsequently applied. A collection of these rules forms a decision list; if more than one rule holds in a given state, then Khardon chooses the rule which was correct more often during training.

If a decision list is restricted to use only predicates of the original domain, then it would be unable to represent some concepts which are essential to solving common planning problems. For instance, the venerable blocks world domain (Slaney and Thiébaux, 2001), in which an agent must stack blocks on top of one another in a particular order, includes a predicate $\text{on}(?a, ?b)$ indicating that block $?a$ sits on top of block $?b$. To determine whether the block at the top of a tower of blocks needs to be moved, the agent must check whether it is *in-position*. That is, whether the top block sits atop the correct block, whether the block below it is also atop the correct block, and so on, down to the bottom of the tower. If any block in the tower is in the wrong place, then the tower must be unstacked to remove the offending block; if all blocks are in the correct position, then dismantling the tower would be a waste of time. Hence, an agent may have to reason over an arbitrarily long chain of propositions of the form $\text{on}(b_1, b_2), \text{on}(b_2, b_3), \ldots, \text{on}(b_{n\text{-}1}, b_n)$. However, decision lists can only accommodate conjunctions of fixed length. In each domain, Khardon (1999), thus enriched the input representation with hand-coded *support predicates* to capture key knowledge which fixed-length conjunctions of propositions could not express. The resulting system was able to learn reasonable—but not entirely reliable—generalised policies for the blocks world domain.

Later work on generalised heuristics and policies explored models which increased

expressiveness, removing the need for support predicates. One common thread in this later work was the use of more expressive logical syntax to express rules in decision lists. Techniques belonging to this category include *concept language* (Martin and Geffner, 2000), *taxonomic syntax* (Yoon et al., 2002), and first-order (logical) regression of optimal policies and value functions (Gretton and Thiébaux, 2004). In addition to modelling of unary and binary predicates, both types of syntax can model the kinds of recursive relationships which were beyond the capacity of basic decision lists, including following arbitrarily long chains of $\mathrm{on}(?a, ?b)$ relations in blocks world. Generalised policies can be represented using a decision list in which each the condition for each rule is a concept language or taxonomic syntax expression, instead of a conjunction of predicates.

Decision trees have also been explored as a representation for generalised policies. In particular, the ROLLER planner (de la Rosa et al., 2008, 2011) uses one decision tree to select an action schema in a given state, then another decision tree to select which objects to use to instantiate the given action schema. These decision trees can be interpreted as hierarchical decision lists; however, they are in fact equivalent to decision lists in representational power, and can therefore be thought of as decision lists for the purposes of this discussion (Blockeel and de Raedt, 1998). ROLLER's critical advantage over the decision lists of Khardon is its choice of representation for observations. Specifically, ROLLER's decision trees are supplied with three kinds of information: predicates which appear in the goal, static predicates (that is, those which no action can affect), and *helpful action* flags. The helpful action flags are produced by the Fast-Forward (FF) planner (Hoffmann, 2001). As part of its heuristic computation process for each state, FF computes a subset of helpful actions which could be useful for reaching the goal in a delete relaxation of the problem. ROLLER passes this information to its decision trees with pseudo-predicates. For instance, in blocks world, $\text{helpful-put-on-block}(?b_1, ?b_2)$ can be used to indicate that $\text{put-on-block}(?b_1, ?b_2)$ is a helpful action. In practice, this additional information compensates for the inability of relational decision trees to directly model recursive relationships of arbitrary depth. In Section 3.3, we show how a similar strategy can be used with neural networks to produce a policy representation which exhibits strong performance and generalisation.

Several existing approaches to generalised policy learning use a hierarchical approach in which the predictions of several similar models are combined through weighting or averaging. For instance, Gretton (2007) proposes a mechanism to probabilistically choose between several control rules; the weights for this selection process are obtained through reinforcement learning on a single problem. *Ensembles* are another possible mechanism for combining several imperfect models policies into a single strong model (Dietterich, 2000). Both Yoon et al. (2002) and de la Rosa et al. (2011) note that their respective policy representations often include imperfections which can be highly damaging in some domains. However, if several models are trained on several different subsets of the training set, then it is likely that their imperfections will not overlap. Hence, training several models in this way and then averaging their predictions (e.g. through a voting mechanism) can substantially improve performance. This technique is model-agnostic, and could in principle be used in conjunction with the work presented later in this thesis.

Concurrent with the preparation of this thesis, Groshev et al. (2017) have proposed another method for learning generalised policies for planning problems. Similar to us, Groshev et al. consider the problem of learning weights for a neural network which encode a generalised policy for a deterministic planning problem (e.g. one expressed as PDDL). Unlike us, Groshev et al. encode this policy as a 2D convolutional neural

network, which is trained on image representations of each state. Convolutional neural networks—which are covered in Section 2.3.2—can generalise to images of different sizes, so this method is in principle capable of generalising to problems of different sizes. As a result of their use of convnets, the method of Groshev et al. requires the user to explicitly define a mapping from propositional states to 2D images, whereas our proposed method can work directly with propositions.

Finally, we note that the Factored Policy Gradient (FPG) planner (Buffet and Aberdeen, 2009) uses a multi-layer perceptron (Section 2.3.1) to map a vector of propositions' truth values to a probability distribution over actions. While FPG's use of a neural network is reminiscent of the approach which we present in Chapter 3, it cannot be applied to problems of different sizes, and is thus inadequate for learning generalised policies or heuristics.

2.2.3 Knowledge acquisition

Having considered representations which are commonly used to encode states and to store learnt knowledge, we now turn our attention to the methods have previously been employed to acquire learnt knowledge. The broad topic of knowledge acquisition can be broken down into two components: first, we will consider the training algorithms used to turn experience into learnt knowledge. Second, we will address the issue of acquiring experience, and in particular how appropriate pairs of states and actions can be derived from one or more user-supplied training problems.

The choice of training algorithm for a given machine learning system depends largely on the choice of model. The decision lists of Khardon (1999) can be trained using Rivest's algorithm (Rivest, 1987). In a planning context, Rivest's algorithm begins by enumerating all rules which could predict the correct action for at least one state in the training set. It then selects a set of the most effective rules to use in a decision list. For Yoon et al. (2002) learning is more complex, as each rule in a learnt decision list could depend on a complex taxonomic syntax expression, rather than just a single conjunction of predicates. Yoon et al. thus use a form of heuristic search through the space of taxonomic expressions. This search produces a series of taxonomic syntax expressions which predict the correct action on a large number of observed states, but do not predict an incorrect action for any observed states. Rather than resorting to another ad-hoc form of heuristic search, ROLLER (de la Rosa et al., 2011) can instead leverage existing work on learning decision trees (Blockeel and de Raedt, 1998; Quinlan, 1986) to acquire a policy. Roughly speaking, these tree learning algorithms greedily build a model which classifies the training data, then prune the model back to make overfitting less likely.

It's worth pointing out that all three of the algorithms described above must acquire a policy by performing some sort search through a discrete space of hypotheses. This search can be computationally taxing due to the high branching factor, thus limiting what can be learnt in practice. For instance, Yoon et al. find it is necessary to limit themselves to taxonomic syntax expressions with a fixed bound on depth. In contrast, the model which we present in Chapter 3 has real-valued parameters which are differentiable with respect to our chosen loss—although any loss that is a differentiable function of the model's output could be used. Not only does this give us a great deal more flexibility in our choice of loss, but it also makes it possible to use of state-of-the-art first-order optimisers (Kingma and Ba, 2014) to train the model. This approach presents different tradeoffs to existing approaches, and in Chapter 5, we show that it can work well on a range of domains in

practice.

Having considered the mechanisms by which observations—e.g. pairs of states and optimal actions–can be distilled into policies, we can now consider methods by which those observations can be acquired. To bootstrap the learning process, most existing approaches require a series of planning problems which each have a different initial state or goal, but all belong to the same domain. From there, it is possible to explore the supplied problems to generate a large set of observations for training. For instance, Yoon et al. begin with a set of randomly generated problems for a given domain, then run a probabilistic planner on each training problem to obtain a set of policies. Those policies are then executed repeatedly to obtain a new set of observations which are known to lie on goal trajectories. In a deterministic setting, the ROLLER planner instead attempts to find state and action pairs along *all* optimal (or near-optimal) trajectories for each training problem, then uses those pairs to train its operator and binding classifiers (de la Rosa et al., 2011). Specifically, ROLLER first finds a single reasonable-quality goal trajectory of cost c to upper-bound the cost of solutions it will consider. It then uses a branch-and-bound search to enumerate all paths to the goal with a cost no greater than c. This ensures that ROLLER's learnt knowledge is not influenced by the (arbitrary) initial goal trajectory which the underlying deterministic planner returns.

While both Yoon et al. and de la Rosa et al. attempt to acquire a full set of training observations before learning begins, there also exist approaches which intersperse acquisition of new observations with policy improvement. For example, Fern et al. (2004a) present an algorithm based on *Approximate Policy Iteration* (API). Their API variant alternates between optimising a policy to ensure that it performs well on states in observed traces, and executing the policy to obtain more traces. In general, approaches like this have the advantage of being able to adapt to the weaknesses of a partially-trained policy. For instance, if a learning planner continually gets stuck in some subset of states on the supplied problems, then an algorithm with alternating training and execution will be able to correct that deficiency by adding those states to the training set. In Chapter 4, we present another strategy for performing this kind of alternation between planning and learning.

Although most learning planners require a user-supplied set of training problems for each domain, there are a handful of existing approaches which can instead be trained using only a single problem. For instance, the LRW-LEARN algorithm (Fern et al., 2004b) is able to automatically synthesise training problems using random walks. Specifically, LRW-LEARN is supplied with a single planning problem with start state s_0, then takes a series of long random walks from s_0. At the end of each long random walk, LRW-LEARN records the final state s in which it arrives, and produces a new training problem in which the start state is s_0 and the goal is to reach state s. Fern et al. show that LRW-LEARN works well when the distribution of random walk tasks is representative of the tasks used for testing. However, this is seldom the case in problems which require improbable sequences of actions. For instance, LRW-LEARN is unlikely to discover that it must fetch a key from one corner of a map to unlock a door on the other.

The FPG planner (Buffet and Aberdeen, 2009) takes a slightly different approach to learning from a single problem. Initially, FPG takes random walks through state space, hitting the goal only a small fraction of times. After each walk, FPG updates its parameters using reinforcement learning. FPG is rewarded for reaching the goal quickly, so each parameter update (hopefully) increases the proportion of times FPG reaches the goal and decreases the expected cost of reaching the goal. The catch is that FPG may take a long

time to improve if few of its execution trajectories reach a goal state. This problem can be somewhat ameliorated by using importance sampling with a low-cost "teacher" policy that can lead FPG away from less-promising states (Buffet and Aberdeen, 2007). This can improve performance, but requires making a tradeoff between the quality of the teacher policy and the cost of computing it. Performance can also be improved by giving FPG small "shaping" rewards when it makes useful progress towards the goal—for instance, when it achieves one proposition in a conjunctive goal. However, it can be hard to devise good incremental reward schemes to use for shaping Buffet and Hoffmann (2010). These mixed results, along with those for LRW-LEARN, suggest that learning policies without a set of small training problems is quite challenging, and we do not consider the problem further.

2.2.4 Knowledge exploitation

When a generalised policy optimally solves a family of problems, it is trivial to use it at test time: one can merely keep choosing actions recommended by the policy until a goal state is reached, or an unavoidable dead end encountered. In practice, it is quite difficult (and sometimes impossible) to learn optimal generalised policies, and so learnt policies are often combined with an exploitation mechanism which is able to recover from poor action recommendations using search. These exploitation mechanisms are generally agnostic to the representation chosen for learnt knowledge, and to the way in which learnt knowledge is acquired. Hence, the strategies considered in this section could be combined with the policy representation which we consider in Chapter 3, and the training mechanism which we consider in Chapter 4.

de la Rosa et al. (2011) suggest two ways of exploiting the generalised policies which their ROLLER system can learn for deterministic problems. Both exploitation mechanisms use a sorting and filtering mechanism for actions which accounts for the confidence which a policy has in an action, and for whether or not the action is considered helpful by the FF heuristic. The first method which uses these filtering and sorting mechanisms, *depth-first H-context policy*, performs a kind of Depth-First Search (DFS). Specifically, it produces a search tree by, at each iteration, taking an action from the most recently visited state which is most highly recommended by the sorting and filtering policy. If all actions have been filtered out, or a repeated state is encountered, the algorithm backtracks up the search tree to instead take an action for an ancestor state which has not been filtered out. If no such action is available, the algorithm begins to consider filtered states as well. The second method, *H-context policy lookahead BFS*, instead extends Breadth-First Search (BFS). Like all breadth-first algorithms, it grows a search tree outward from the initial state by taking one action at a time, preferring to take actions in states which appear at a lower depth in the tree. However, at each visited state, the algorithm also expands all states reachable using actions recommended by the action filtering procedure, up to some fixed lookahead depth. The states added through this lookahead mechanism can be explored before states earlier in the search tree, thus allowing the planner to jump ahead in state space when the policy recommends good actions.

de la Rosa et al. note that their BFS-based algorithm is not as sensitive to flawed policies their DFS-based procedure. This makes intuitive sense if one considers how each algorithm uses action recommendations. Where DFS follows recommended actions all the way to a dead end, goal, or repeated state, BFS instead follows recommended actions only for a fixed number of steps. BFS thus has less backtracking to do if it encounters a

series of flawed action recommendations. While the above algorithms cannot be applied directly to the probabilistic planning problems which we consider here, it is likely that the same insights about flawed policies still apply in a probabilistic setting.

Limited Discrepancy Search (LDS) has also been explored as a possible strategy for exploiting learnt knowledge in a deterministic setting. LDS tries to find a trajectory of goal states by greedily choosing actions using a supplied policy. However, LDS is also able to ignore those recommendations at a limited number of states along each trajectory, and instead explore other, non-recommended successors. Allowing only k discrepancies along a trajectory greatly reduces the size of the search tree for a problem, while still allowing LDS to discover plans where the recommended action is incorrect in up to k states. This strategy has been successfully employed in deterministic planning (Yoon et al., 2006a). However, it's not clear how to extend the same approach to probabilistic planning, where actions can have multiple outcomes and a single goal trajectory is thus insufficient to solve a problem.

Another option for exploiting learnt knowledge is beam search, which takes a similar approach to LDS. Like greedy search, beam search tries to follow the guidance of a heuristic until it reaches a goal state. Unlike greedy exploitation, which only considers a single state at a time, beam search is able to maintain a fixed-size *beam* of b states. At each iteration, the successors of those b states are enumerated, and the states in the beam are replaced by the b "best" successors. Beam search requires some way of ranking the desirability of states; while a policy could be used to rank the desirability of the successors of a *single* state. Hence, beam search is better suited to exploitation of learnt heuristics (Xu et al., 2007) or learnt state-ranking functions (Xu et al., 2009).

There are also several sampling-based strategies for exploiting learnt policies. The simplest of these is policy rollout: to obtain an action for a state s, policy rollout first runs a series of trials to compute an approximation $\hat{Q}(s, a)$ of the Q-value $Q(s, a)$ for each action $a \in \mathcal{A}$. $\hat{Q}(s, a)$ can be obtained by averaging the costs observed over all trials in which action a was the first action chosen. The planner can then choose the action which maximises $\hat{Q}(s, a)$—an approach which has proven helpful for exploiting learnt policies in probabilistic planning (Fern et al., 2004a). Yoon et al. (2007) presented a variation on this strategy in which periodic rollouts with a learnt policy π are used to expand the search tree for an ordinary heuristic search strategy. This is much like ROLLER's H-context policy lookahead BFS strategy, but for probabilistic problems rather than deterministic ones. More options can be found among the family of Monte-Carlo Tree Search (MCTS) algorithms, which use repeated policy rollouts to gradually expand a search tree. Of particular note is the UCT variant of MCTS, which uses the UCB1 formula to trade off exploration of novel parts of the game tree against exploitation of the most promising parts of the game tree (Kocsis and Szepesvári, 2006). Combined with a non-learning heuristic, UCT has yielded state-of-the-art performance in probabilistic planning (Keller and Eyerich, 2012). UCT has also been combined with learnt policies for the game of Go, yielding a Go agent with superhuman performance (Silver et al., 2016). While there is debate over whether UCT is the most appropriate MCTS variant for planning (Domshlak and Feldman, 2013), its excellent performance in other applications nevertheless makes UCT an attractive option for exploiting learnt policies.

2.3 Structured deep learning

The main contribution of this thesis is a sneural network architecture which is structured for learning on probabilistic planning problems. In this section, we place that contribution in context by examining how a structured approach differs from an unstructured one, and introducing a number of existing strategies for structured deep learning in other domains.

2.3.1 Unstructured neural networks

The simplest kind of neural network is known as a *Multi-Layer Perceptron* (MLP), or *fully-connected network*. An MLP is structured into L layers. The first of these layers receives an input vector which subsequent (intermediate) layers then transform in some way, finally leading to an output vector which approximates some quantity of interest. Each intermediate (hidden) layer takes a vector-valued hidden representation $h^{(l-1)}$ from the previous layer. It applies an affine transform to obtain $z^{(l)} = W^{(l)}h^{(l-1)} + b^{(l)}$, then passes each element of $z^{(l)}$ through an elementwise transform f to obtain a hidden representation

$$h^{(l)} = f\left(W^{(l)}h^{(l-1)} + b^{(l)}\right) . \tag{2.13}$$

f's purpose is to stretch or squash the output of each layer so that the network is capable of learning nonlinear functions. If f were the identity function, then a K-layer network would be equivalent to a single affine transformation; hence, f is always a nonlinear function like $f(x) = \max(0, x)$ (ReLU) or $f(x) = \tanh(x)$. With these nonlinearities, the *units* or *neurons* which comprise each hidden representation are able to capture successively more complex properties of the input. In fact, even a two-layer MLP is a "universal" function approximator, in the sense that any continuous function on a compact domain can be approximated to arbitrary accuracy given enough hidden units (Bishop, 2006).

MLPs are frequently trained in a supervised setting: given a dataset $\mathcal{D} = \{(x_1, y_1), \ldots, (x_N, y_N)\}$ composed of inputs $x_1, \ldots, x_N$ and labels $y_1, \ldots, y_N$, the weights $W^{(1)}, b^{(1)}, \ldots, W^{(L)}, b^{(L)}$ will be chosen to minimise a loss

$$\mathcal{L}(\mathcal{D}) = \sum_{i=1}^{N} \ell(\hat{y}_i, y_i) . \tag{2.14}$$

$\ell(\hat{y}_i, y_i)$ measures the mismatch between the true label y_i for input x_i, and the MLP's output $\hat{y}_i$ when given input x_i. When ℓ is a differentiable function of the MLP's output (e.g. $\ell(\hat{y}, y) = \|\hat{y} - y\|^2$), this loss can be minimised by gradient descent. By starting from the output layer and working backwards, *backpropagation* (i.e. the chain rule) can used to calculate the gradient of the loss with respect to each weight. Ultimately, this produces a gradient $\nabla_\theta \mathcal{L}(\mathcal{D})$ of the loss with respect to a concatenated vector θ of all model weights.[3]

Given a gradient, an optimiser can perform a single parameter update of the form

$$\theta \leftarrow \alpha \nabla_\theta \mathcal{L}(\mathcal{D}) , \tag{2.15}$$

which will obviously decrease $\mathcal{L}(\mathcal{D})$ given a sufficiently small step size $\alpha \in \mathbb{R}$, except when $\mathcal{L}(\mathcal{D})$ is at a stationary point. In practice, faster convergence to a local optimum

[3] It is common in machine learning—not just in the context of neural networks—to use θ to denote a collection of all parameters of a model. We will use this notation throughout the thesis.

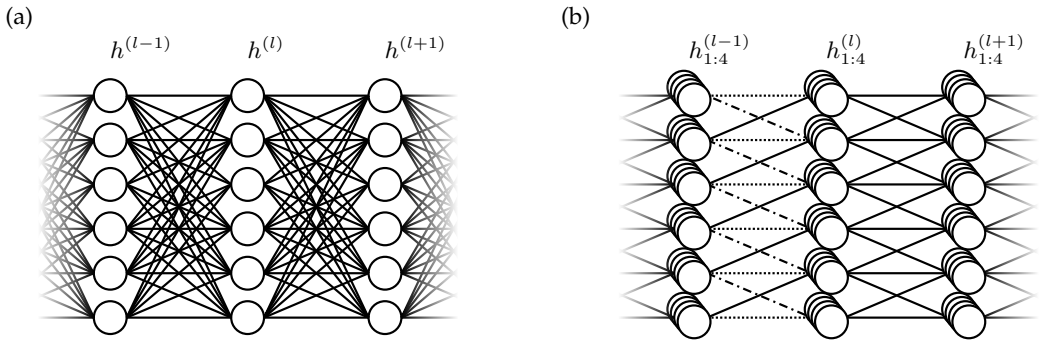

Figure 2.3: (a) Connections (lines) between neurons (circles) in successive layers of a fully-connected MLP, compared to (b) connectivity in a 1D convnet with a kernel of width three, and four feature maps at each layer (stacked circles). Lines between the first and second depicted layers of the convnet have been given dashes to illustrate that the same operation is being applied to each patch of input. Edges with the same dash style correspond to the same learnt weight.

can be attained with *stochastic gradient descent* (SGD), which replaces the whole-dataset gradient $\nabla_\theta \mathcal{L}(\mathcal{D})$ with a gradient $\nabla_\theta \mathcal{L}(\mathcal{B})$ computed on a randomly chosen *minibatch* $\mathcal{B} \subseteq \mathcal{M}$. It's worth pointing out that neither gradient descent nor SGD are guaranteed to converge to a globally optimal parameter vector θ^* in general.

An MLP's lack of structure poses two problems: first, the meaning and size of an MLP's inputs and outputs are fixed, so you are often unable to transfer learnt weights between similar learning tasks. For instance, if an MLP is trained to interpret the values of a set of propositions, then scrambling the order of those propositions will cause the MLP to output nonsense, and adding more propositions will preclude the MLP from processing their values at all. Second, MLPs are not able to make use of sensible priors about which units in one layer should be connected to which units in the next. In principle, they can connect any unit to any unit, and must learn a weight for every of those connections independently, from scratch. In contrast to MLPs, structured neural networks can make learning easier by imposing sensible priors on neuron connectivity. In some cases, structured networks can also generalise to problems of different sizes by reusing weights for connections which perform similar roles.

2.3.2 Convolutional neural networks

The first class of structured deep learning approaches which we will discuss are *Convolutional Neural Networks* (CNNs). Although CNNs are most commonly applied to 2D image inputs, the underlying concepts can be generalised to inputs with any number of dimensions, so we will start with 1D convnets. Instead of producing a single hidden representation $h^{(l)} \in \mathbb{R}^D$ at each intermediate layer, a convnet produces a series of feature maps $h^{(l)}_1, \ldots, h^{(l)}_M \in \mathbb{R}^D$. Each feature map $h^{(l)}_m$ is itself produced by convolving each map in the previous layer with a corresponding filter from a filter bank $\mathcal{F}^{(l)}_m = \{\mathcal{F}^{(l)}_{m,1}, \ldots, \mathcal{F}^{(l)}_{m,M}\}$. Each filter is a vector $\mathcal{F}^{(l)}_{m,n} \in \mathbb{R}^w$, where w is the filter width. Hence, the m-th feature map at layer l is

$$h^{(l)}_m = f\left(\sum_{n=1}^{M} h^{(l-1)}_n * \mathcal{F}^{(l)}_{m,n} + b^{(l)}_m\right) , \tag{2.16}$$

where $u * v$ denotes 1D convolution, f is a nonlinearity, and $b_m^{(l)} \in \mathbb{R}$ is a learnt bias applied to every element of the feature map. The summation indexes over feature maps in the previous layer. Figure 2.3 highlights the intuitive the difference between a 1D convnet and an MLP: each element of $h_m^{(l)}$ is produced by applying a linear transformation to nearby elements of each feature map in the previous layer. Because the same linear transformation—with the same weight matrix—is applied at each position in the feature map, this scheme constitutes a kind of *weight-tying* or *weight-sharing* (LeCun et al., 1995).

The kind of local connectivity and weight sharing scheme employed by CNNs has several convenient properties. First, applying only local transformations to obtain each hidden representation output can substantially decrease the size of weight vectors which need to be learnt, as there are fewer connections between units in successive layers. Second, the use of weight-sharing decreases the number of weight vectors to be learnt, since the same learnt transformation can be applied at each position in the input vector. Third, weight-sharing and local operations allow the network to be applied to inputs of different size. A convolution operation merely "slides" a filter across a feature map, and sliding a filter across a longer feature map will just produce a proportionally longer output. Finally, weight-sharing introduces a form of translational invariance to the network: if the input is shifted by k positions, then the output will be shifted by that many positions as well.

Convolutional neural networks have met with astounding success in computer vision. In a vision setting, the 1D feature maps described above are replaced with 2D feature maps, where positions within the map correspond roughly to pixels in an image. Likewise, the 1D filters are replaced with 2D filters of some fixed spatial extent (e.g. 3x3 pixels). Even though the windows considered by these filters are small, the overlap between windows in successive layers means the *effective receptive field* of a CNN—that is, the region of input pixels which are able to contribute to the output of a given filter in the final layer of the network—grows rapidly with the number of layers (LeCun et al., 2015). CNNs are thus able to learn to detect increasingly complex image features at each layer. Figure 2.4 shows some image patches which lead to high activations of filters in different layers of a neural network, along with a visualisation of the outputs of those filters (Zeiler and Fergus, 2014). Filters from early layers mainly capture local geometric patterns like lines and corners, while filters from later layers can capture patterns which closely resemble larger objects like faces, wheels, etc.

2.3.3 Graph convolutions

The local connectivity pattern of a CNN can be characterised by a highly regular graph of connections among hidden units in successive layers. In the case of a 1D convnet, this graph connects units in each width-w spatial extent in the lth layer with units at a single location in the $l+1$th layer. In the case of a 2D convnet, this graph connects units in a fixed-size patch at a certain position in one layer with units in a single-element patch in the same position at the next layer. From this perspective, it is natural to ask whether a similar trick can be extended to irregular graphs to yield a general *graph convolution*. In Chapter 3, we present a convolution-like method which operates on a graph of actions and propositions for a factored planning problem. In this subsection, our purpose is to relate that work to the extant literature by briefly surveying previous approaches to CNN-like architectures for irregular graphs.

Much of the recent work on graph convolutions has focused on *spectral methods* for

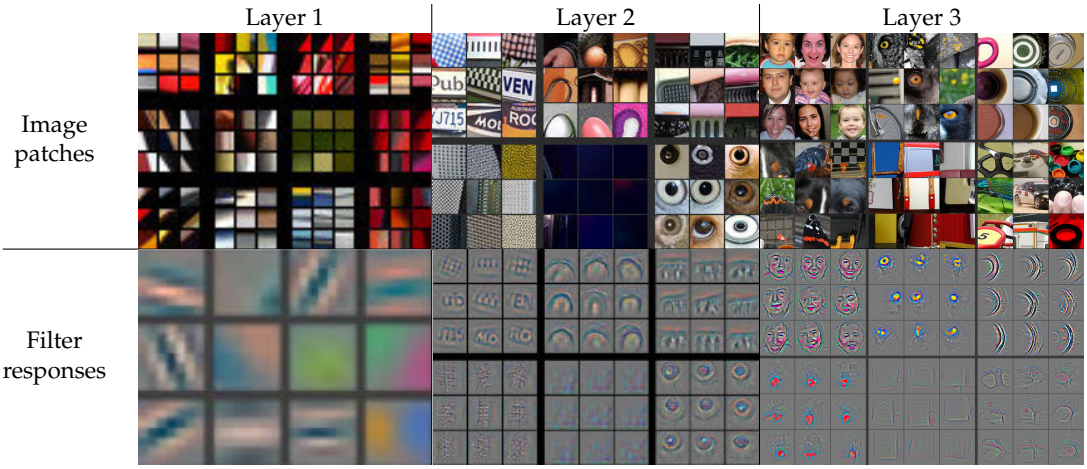

Figure 2.4: Visualisation of concepts learnt by the first three layers of a CNN. The top row shows images which activated a particular filter in a given layer, while the bottom row visualises the information contained in the corresponding filter activations. Fine filter visualisations for later layers are best viewed digitally. Adapted from Zeiler and Fergus (2014).

learning on undirected graphs (Bruna et al., 2013; Defferrard et al., 2016; Henaff et al., 2015; Kipf and Welling, 2016). Spectral methods leverage a graph equivalent of the *convolution theorem* from signal processing. Where convolutions in the spatial domain of a signal (e.g. the pixels of an image) are achieved by sliding a filter across the signal, convolutions in the frequency domain are instead achieved through an elementwise product between the sampled signal and the filter. For graphs, the equivalent of 'frequency space' is defined in terms of the eigenvalues (spectrum) of the graph's Laplacian matrix; the nascent field of *graph signal processing* is dedicated to the study of this representation (Shuman et al., 2013). Unfortunately, graphs with different connectivity, different vertex count, etc. do not necessarily have directly comparable eigenvalues and eigenvectors, and neural network parameters learnt with current spectral methods are thus difficult to transfer between graphs (Bronstein et al., 2017). In generalised planning, we have to accommodate problems of different sizes, with different numbers of objects, propositions, and actions, and different patterns of connections between them. This makes spectral methods unsuitable for our purposes.

Graph convolutions in the spatial domain are much easier to generalise to different graphs than those in the frequency domain. In general, spatial domain approaches choose a neighbourhood around each vertex, then apply a learnt transformation to the data associated with edges and vertices in each of those neighbourhoods. This process can be repeated several times to produce a multi-layer network.

The general strategy underlying spatial graph convolutions is perhaps best illustrated by the Neural Graph Fingerprints (NGFs) of Duvenaud et al. (2015), as depicted in Figure 2.5. NGFs take as input a set of features for each atom in a molecule, and produce as output a fixed-size vector which captures chemically-relevant information about the molecule. Each intermediate layer in the NGF network performs a kind of convolution across each of the atoms. Specifically, for an atom a in layer $l+1$, the NGF gathers the atom's hidden representation $r_a^{(l)} \in \mathbb{R}^d$ at the lth layer, along with the hidden representations of the neighbouring atoms to which it is connected, and produces a new hidden

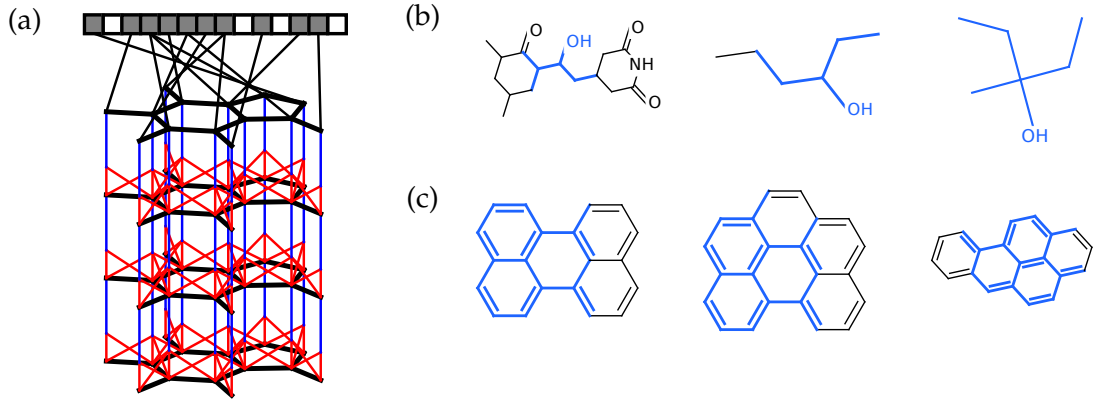

Figure 2.5: (a) A neural network for producing neural graph fingerprints from molecule descriptions. Information flows from bottom to top through depicted edges. (b) Fragments of molecules (highlighted in blue) which obtained the greatest response from a hidden feature correlated with solubility. (c) Fragments which obtained the greatest response from a hidden feature correlated with insolubility. Illustrations from Duvenaud et al. (2015).

representation $r_a^{(l+1)} \in \mathbb{R}^d$ for atom a:

$$v_a^{(l)} = r_a^{(l)} + \sum_{i \in \text{neighbours}(a)} r_i^{(l)} \tag{2.17}$$

$$r_a^{(l+1)} = f\left(H^{(l)} v\right) , \tag{2.18}$$

where f is a nonlinearity, and $H \in \mathbb{R}^d$ is a learnt weight matrix. The operations presented here have been slightly simplified relative to the original paper, but still capture the critical features of their convolution-like operation. In particular, each layer maintains a hidden representation for each vertex in the original graph. The hidden representation for an atom is produced by combining the previous layer's representations for it and its neighbours, then applying some learnt transformation to it. By "pooling" the representations for neighbouring nodes into a single fixed-size vector using summation, and by applying the same transformation to the hidden representations for each atom in a layer, Duvenaud et al. (2015) ensure that the same set of learnt weights can be applied to a molecule of any size.

Other approaches to graph convolutions in the spatial domain are broadly similar, albeit with domain-specific complications in some instances. Kearnes et al. (2016) present another method for molecular fingerprinting, albeit one which includes modules for each *pair* of connected atoms, and modules for each individual atom. The Structural Recurrent Neural Network (SRNN) of Jain et al. (2016) uses a similar architecture, but it is instead applied to the task of modelling interactions between people and objects in a visual scene. The graph considered by the SRNN includes vertices for both humans and objects, and the vertices for humans apply a learnt transformation which is different to that applied by vertices for objects. In Chapter 3, we use a similar strategy to Kearnes et al. and Jain et al. to accommodate modules for each action and each proposition in a planning problem. Finally, Niepert et al. (2016) present an alternative method which takes a graph as input, then performs a convolution-like operation on only a fixed-size subset of nodes. This ensures that the network's output is always of the same size, regardless of the size of the input graph. The network presented in Chapter 3 produces a single output for each

action, so strategy of Niepert et al. could be useful when only a single output for each state is desired (e.g. when learning a heuristic).

2.3.4 Alternative approaches

Although we have focused on convolution-like architectures, there are a number of other neural network architectures which could be well-suited to the sorts of structured problems which we consider in this thesis. For instance, Milan et al. (2017) suggest using a *Recurrent Neural Network* (RNN) with *Long Short-Term Memory* (LSTM) units (Hochreiter and Schmidhuber, 1997) to solve the Travelling Salesman Problem (TSP), bipartite matching, and other intractable problems which require a sequence of items to rearranged in some order according to difficult-to-satisfy constraints. For instance, in TSP, a list of cities (without duplicates) must be rearranged into an order which minimises the sum of distances between cities. LSTM RNNs are sequence processing models which can take a sequence of vectors as input, and produce a corresponding sequence of vectors as output, where the vector produced at time t potentially depends on all input vectors observed up to time t. Milan et al. thus suggest solving the above problems by training an RNN to take a sequence of item descriptions as input, and produce a sequence of destination positions—one for each input item—as output. The approach of Milan et al. is limited to fixed-size sequences, although it is possible to use the *attention-based* mechanism present in *pointer networks* (Vinyals et al., 2015) to overcome this limitation.

It is conceivable that RNN-based architectures could also be used to solve problems which arise in planning. For instance, in a delete-relaxed deterministic problem, it is only ever necessary to apply an action once, as the effects of actions cannot be "undone". Hence, the NP-hard task of computing the optimal delete-relaxed heuristic h^+ could be rephrased as a task of finding an optimal ordering of delete-relaxed actions, and could therefore be approximated using a pointer network or RNN. Unfortunately, these sequence-to-sequence approaches cannot exploit the relational structure of planning problems, and also lack the *order-invariance* of graph convolutional networks: presenting the actions and propositions of a problem to an RNN in a different order could yield a different output! Hence, we defer exploration of these alternatives to future work.

Recently, we have also seen a number of neural network architectures which mimic the mechanisms underlying general-purpose computers, and are consequently able to learn to solve planning problems directly. For instance the Neural Turing Machine (NTM) (Graves et al., 2014) and its successor, the Differentiable Neural Computer (DNC) (Graves et al., 2016), use a recurrent neural network "controller" and a differentiable form of random access memory to learn how to mimic computer programs from input–output samples. In fact, the DNC is even able to solve small instances of the classic blocks world planning problem. Unfortunately, NTMs and DNCs are extremely difficult to train (Zaremba and Sutskever, 2015), and tend to exhibit poor generalisation ability (Reed and de Freitas, 2015), which makes them inappropriate for reliably scaling up traditional planning approaches.

The Neural Programmer Interpreter (NPI) (Cai et al., 2017; Reed and de Freitas, 2015) is a similar model which obtains better generalisation ability by breaking a high-level computing task down into subtask and environment interactions. Each subtask is equivalent to a function or procedure in an imperative programming language, while interactions are similar to actions in planning. This decomposition simplifies learning by subtasks to be trained separately. However, the decomposition must be provided by the

user. Further, the NPI must be trained on execution traces which show the precise order in which subtasks are invoked by a reference program. While it may be possible to create manual decompositions for specific planning domains, as Reed and de Freitas (2015) do for some algorithmic tasks, having such a decomposition for each domain could make the actual planning task trivial, and merely shift work from the planner to a human. This suggests that NPIs—like NTMs and DNCs—are not directly applicable to speedup learning in planning problems.

2.4 Related work in deep reinforcement learning

"Deep RL" is a label commonly applied to the growing body of work on the use of neural networks for reinforcement learning. In many environments, an ability to plan is essential to effective reinforcement learning, so there is a large overlap between the fields of probabilistic planning and deep RL. Further, some RL methods can also be helpful for planning, as exemplified by the FPG planner's use of reinforcement learning for probabilistic planning (Buffet and Aberdeen, 2009). In this section, we will briefly cover related work at the intersection of deep reinforcement learning and planning, illustrating how the work presented here differs from that in deep RL.

The recently proposed Value Iteration Networks (Tamar et al., 2016) (VINs) are one possible model for integrating planning with deep reinforcement learning. VINs learn to formulate a reward function and state transition distribution for an MDP with a fixed (user-specified) number of states. The VIN then solves the formulated problem by repeatedly applying (differentiable) Bellman backups—that is, by doing value iteration. Finally, the agent uses the solution to the learnt MDP to choose an appropriate action. The original VIN used a 2D convolutional neural network to do Bellman backups, and was thus limited state spaces arranged in a 2D grid of transitions. Later work has yielded a Generalised VIN (GVIN) which lifts that restriction by using graph convolutions to solve the VIN's internal MDP (Niu et al., 2017). The architectures of VINs and GVINs force them to learn to model their environments, thus making it more likely that learnt knowledge will generalise to environment configurations.

The primary difference between our work and the work on VINs is the setting. Where VINs aim to give neural networks the ability to formulate and solve planning problems, we are instead interested in using neural networks to learn generalised solutions to planning problems with known dynamics. Further, the network which we propose in Chapter 3 reasons about a factored representation of a known planning problem. In contrast, (G)VINs must reason about a problem representation in which each possible state is considered separately. As noted previously, this representation can be exponentially larger than a factored one.

Similar comments apply to the Strategic Attentive Writer (STRAW) of Vezhnevets et al. (2016). Like the VIN, STRAW improves generalisation of policies acquired through deep RL by structuring the policy network in such a way that it is forced to plan. Specifically, STRAW maintains an *action plan* indicating which actions it intends to take up to a fixed time horizon, and a *commitment plan* indicating the time steps at which it intends to revise the action plan. At each iteration, it either executes the appropriate action from the action plan, or chooses to replan, depending on the corresponding entry in the commitment plan. STRAW can thus be interpreted as a method for learning to produce macro actions on-the-fly. Unlike the VIN, STRAW's architecture allows it to consider only ac-

tions in its internal plan, rather than having to maintain value estimates for each state in an unfactored MDP. Of course, STRAW is still aimed at improving the capabilities of deep RL methods in environments requiring planning, rather than at learning policies for known planning problems. Further, unlike solutions to the generalised policy learning problem considered in this thesis, STRAW is not able to generalise to problems with different numbers of actions.

There are also similarities between the action schema networks presented in Chapter 3 and the schema networks of Kansky et al. (2017).[4] Schema networks are a kind of neural network architecture composed of several layers of *entity* modules. Each entity module corresponds to a particular object in a scene, as identified by a vision system. Kansky et al. test schema networks on the game of Breakout, where appropriate objects might include the paddle, the ball, and the bricks which the agent must hit. Each layer of a schema network corresponds to a future time step. Layers are connected in such a way that the network is able to predict states and rewards at each future time step using the outputs of the entity modules in the corresponding layer. The weights for each module and the relations between them are learnt automatically. Importantly, Kansky et al. only use schema networks to learn a model for an environment. They can then obtain a policy for the modelled environment by casting planning as a MAP inference problem, which is solved using a separate inference procedure. In contrast, this thesis assumes a model of the environment is given as PPDDL, and is instead concerned with directly learning a generalised policy.

[4] It should be emphasised that "schema networks" and "action schema networks" were devised concurrently and independently, despite the colliding names and similarity in structure.

Chapter 3

Action Schema Networks

In Section 2.3, we argued that much of the success of deep learning in natural language processing, vision, and elsewhere has been due to the existence of network architectures and training strategies which are well-suited to those domains. Despite the established utility of machine learning in automated planning, there are, to the best of our knowledge, no such architectures for PPDDL-style planning problems. In this chapter, we close this gap by introducing a new family of neural networks which we call Action Schema Networks (ASNets). Further, we will show how weights can be shared between components of an ASNet in such a way that the total number and size of weights is the same for any problem in a given PPDDL domain. This scheme is essential to an ASNet's ability to encode a generalised policy: an ASNet can take propositions and heuristic values describing a state of *any* problem from a given domain, and produce an appropriate action as output.

3.1 Network structure

At a high level, an ASNet is composed of alternating action layers and proposition layers, as depicted in Figure 3.1. Each action layer includes a single *action module* for each ground action, and each proposition layer likewise includes a single *proposition module* for each ground proposition. The first layer of an ASNet is always an action layer, where each action module takes as input a vector describing features of the current state which are directly relevant to that action. Proposition modules in each subsequent proposition layer of an ASNet connect only to *related* action modules in the previous action layer. Likewise, action modules in each subsequent action layer connect only to related propo-

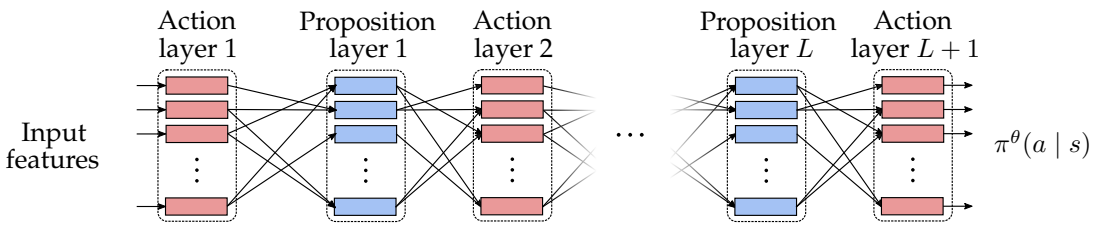

Figure 3.1: High-level illustration of an ASNet with L proposition layers and $L+1$ action layers; we refer to such a network as an "L-layer ASNet". Action modules are shown in red and grouped into action layers, while proposition modules are shown and blue and grouped into proposition layers.

```
(define (domain unreliable-robot-domain)
  ;; ...
  (:action drive
    :parameters (?r - robot ?from ?to - place)
    :precondition (and (robot-at ?r ?from) (path ?from ?to))
    :effect (probabilistic
      9/10 (and (robot-at ?r ?to)
                (not (robot-at ?r ?from))))))
)

(define (problem unreliable-robot-problem)
  ;; ...
  (:domain unreliable-robot-domain)
  ;; ...
  (:objects kitchen hall office-1 office-2 - place
            shakey - robot)
  ;; ...
)
```

Figure 3.2: Relevant portions of the Unreliable Robot domain and problem definitions. The full PPDDL domain for Unreliable Robot is shown in Figure 2.1, while the full problem is shown in Figure 2.2.

sition modules in the previous proposition layer. We will formalise our notion of relatedness in Section 3.1.1; for now, the important point is that this connectivity scheme allows local propagation of information from one layer to the next. Over the course of many layers, the action and proposition modules of an ASNet can build up a rich vector-space representation of an input state. In the last layer of an ASNet—which is always an action layer—this representation can be used to obtain a probability of selecting each applicable action. Hence, an ASNet can be used to define a policy $\pi^\theta(a \mid s)$ parameterised by the weights θ of the network.

3.1.1 Relatedness

The connections between action modules and proposition modules reflect whether the corresponding actions and propositions are *related*. Formally, we say that a proposition $p \in \mathcal{P}$ is related to an action $a \in \mathcal{A}$, denoted $R(a, p)$, if p appears in $\mathrm{pre}(a)$ or $\mathrm{eff}(a)$ This relationship is symmetric, so if p is related to a, then we can also say that a is related to p. For brevity, we will say that $R(a, p)$ is true when a and p are related, and false otherwise.

To illustrate this relation, we will extract a few related pairs of actions and propositions from the Unreliable Robot problem which we encountered earlier. We have reproduced relevant sections of the domain and problem in Figure 3.2. The action $\mathrm{drive}(\mathit{shakey}, \mathit{kitchen}, \mathit{hall})$ includes $\text{robot-at}(\mathit{shakey}, \mathit{kitchen})$ and $\mathrm{path}(\mathit{kitchen}, \mathit{hall})$ as preconditions. Hence, both of those propositions are related to $\mathrm{drive}(\mathit{shakey}, \mathit{kitchen}, \mathit{hall})$, and $\mathrm{drive}(\mathit{shakey}, \mathit{kitchen}, \mathit{hall})$ is related to both of those propositions. Likewise, $\text{robot-at}(\mathit{shakey}, \mathit{kitchen})$ and $\text{robot-at}(\mathit{shakey}, \mathit{hall})$ appear in the effect of the action, and are thus related to it (and vice versa). This is true even though $\text{robot-at}(\mathit{shakey}, \mathit{kitchen})$ only appears in a probabilistic sub-effect.

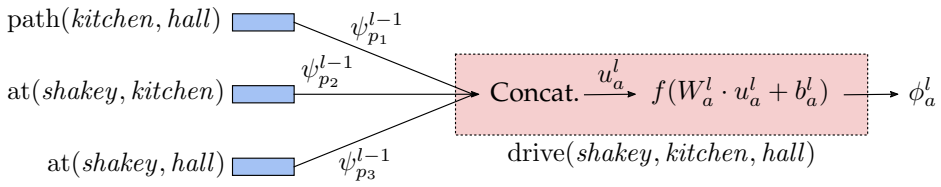

Figure 3.3: Illustration of an action module for the Unreliable Robot problem from Figure 2.2. In this case, the action module is for the $\mathrm{drive}(\mathit{shakey}, \mathit{kitchen}, \mathit{hall})$ action, which moves the robot *shakey* from the *kitchen* to the *hall*. The action module is given hidden inputs $\psi^{l-1}_{p_1}, \psi^{l-2}_{p_2}, \psi^{l-3}_{p_3}$ from related proposition modules in the previous layer, and produces a hidden output ϕ^l_a by applying a linear transform to the concatenated inputs.

To extract a set of related propositions for an action programmatically—as we must do to construct an action module in Section 3.1.2—we first inspect the action schemas in the domain for the problem. For each action schema, we obtain a list of predicates and their arguments from the schema's precondition and effect. To obtain a list of propositions relevant to each ground action, we look up the list of predicates for the corresponding schema, then instantiate each predicate into a proposition, employing the same parameters which were used to generate the ground action. Obtaining lists of relevant propositions in this way allows us to avoid duplicate propositions. For instance, in the action schema for drive (Figure 2.1), the predicate $\mathrm{robot\text{-}at}(?r, ?\mathit{from})$ appears twice—once in the precondition, and once in the effect. However, since both occurrences of the predicate supply the predicate with the same arguments, we only count them as a single occurrence. Enumerating related propositions in this way also makes it straightforward to keep a consistent order between corresponding propositions for two ground actions instantiated from the same schema. As we will later note, keeping a consistent order for related propositions is important for weight-sharing among action modules. Similar comments apply to proposition modules, where we must instead ensure that we have consistent orderings for the actions which are relevant to each proposition.

Having obtained a list of propositions related to each action, and a list of actions related to each proposition, wiring up an ASNet is straightforward. We simply connect action modules in one layer to related proposition modules in the next, and proposition modules in one layer to related action modules in the next. As we noted in Section 2.3.3, this strategy can be interpreted as a kind of graph convolution, in analogue with the 2D convolutional neural networks used to process images. We have merely replaced the spatial neighbourhoods used by convnets with neighbourhoods defined by the relatedness predicate $R(\cdot, \cdot)$. There are also connections between ASNets and past work in planning: on an abstract level, our choice to connect alternating action and proposition layers was inspired by the Graphplan planner (Blum and Furst, 1997), which uses a similar graph of alternating action and proposition layers. However, the links in Graphplan's graph are used to reason about and propagate constraints regarding which propositions can hold and which actions will be applicable at certain times. In contrast, our relations are merely used as a convenient means for locally propagating information.

3.1.2 Action modules

Each action layer in an ASNet is composed of action modules, with exactly one module for each ground action in the corresponding planning problem. The first and last layers of an ASNet are always action layers. However, inputs and outputs to modules in those layers are slightly different to inputs and outputs in intermediate action layers, so we will defer discussion of the first and last layers until later in this section.

An action module in the lth intermediate layer for an action $a \in \mathcal{A}$ takes an input feature vector $u_a^l \in \mathbb{R}^{d_a^l}$ which is composed of hidden representations from relevant proposition modules in the $l-1$th proposition layer. It then applies a learnt transformation on that input to produce a new hidden representation $\phi_a^l \in \mathbb{R}^{d_h}$, which can then be used in the next proposition layer. Formally, this transformation is

$$\phi_a^l = f(W_a^l \cdot u_a^l + b_a^l)\,, \tag{3.1}$$

where $W_a^l \in \mathbb{R}^{d_h \times d_a^l}$ is a weight matrix for the module and $b_a^l \in \mathbb{R}^{d_h}$ is a bias vector, both of which can be learnt through stochastic gradient descent. d_h is a fixed intermediate representation size for the output, and d_a^l is the size of the inputs to the action module. $f(\cdot)$ is a nonlinearity; we use the Exponential Linear Unit (ELU) proposed by Clevert et al. (2016), but any other common activation could also work.

The feature vector u_a^l, which serves as input to a module for action a in the lth layer, is constructed by enumerating the propositions $p_1, p_2, \ldots, p_M$ which are related to the action a, then concatenating their hidden representations. Concatenation of representations for the related propositions produces a vector

$$u_a^l = \begin{bmatrix} \psi_1^{l-1} \\ \vdots \\ \psi_M^{l-1} \end{bmatrix}\,, \tag{3.2}$$

where ψ_j^{l-1} is the hidden representation produced by the proposition module for proposition $p_j \in \mathcal{P}$ in the preceding proposition layer. Each of these constituent hidden representations has dimension d_h, so the input vector u_a^l has dimension $d_a^l = d_h \cdot M$. A complete action module, along with the inputs feeding into it, is depicted in Figure 3.3.

As alluded to in Section 3.1.1, our notion of propositional relatedness ensures that the input vectors for ground actions instantiated from the same schema have comparable structure. This property is essential to the weight-sharing scheme which we propose in Section 3.2. As an example, consider propositions $p_1, \ldots, p_M$ which are related to $a_1 \in \mathcal{A}$, and propositions $q_1, \ldots, q_N$ which are related to $a_2 \in \mathcal{A}$, where a_1 and a_2 are instantiated from the same action schema. Because a_1 and a_2 share an action schema, the lists $p_1, \ldots, p_M$ and $q_1, \ldots, q_N$ must be of the same length (i.e. $N = M$), since the the propositions in each list are instantiated from the same list of predicates. Further, two propositions p_i and q_i in the same position in the two lists must be related to the corresponding action in the same way; that is, they appear in the same locations in their corresponding action definitions. Hence, the input vectors $u_{a_1}^l$ and $u_{a_2}^l$ have the same size and similar semantic meanings. In fact, these observations hold even if a_1 and a_2 are actions from different *problems*—so long as they are instantiated from the same action schema in the same domain, they will have similar input structures.

Although we did not consider it in Chapter 2, PPDDL supports quantification in action schemas. It's worth noting that quantifiers would violate our assumption that ac-

tions derived from the same schema in different problems have the same number of related propositions. To see why this is the case, we will return to our running example of the Unreliable Robot domain. If we had an action schema which took as arguments a robot $?r$ at location $?l$, then we might want to ensure that the action could only be applied if there was no other robot $?r_o$ at a neighbouring location $?l_n$. We can express this constraint using an existential quantification over the objects $\mathcal{O}$ of the problem:

$$\nexists ?l_n \in \mathcal{O}. \exists ?r_o \in \mathcal{O}. \left[\text{path}(?l, ?l_n) \wedge \text{robot-at}(?r_o, ?l_n)\right] . \tag{3.3}$$

This expression will be false if there exists a robot at some location which is connected to $?l$ by a single path, and true otherwise. The result of such an expression could be influenced by a different number of ground propositions depending on how many locations and robots are declared in $\mathcal{O}$. Simply concatenating the hidden representations for those propositions may thus produce a vector with a different size in different problems, which would break the weight sharing scheme in Section 3.2. To avoid this problem, we would have to use a *pooling* strategy like the one which we will introduce for proposition modules in Section 3.1.3.

Input and output layers

There are two subtleties which must be considered for action modules in the first and last layers of a ASNet. In the first layer, there are no preceding proposition layers, so it's necessary to use a different input representation. Likewise, we are interested in using ASNets to learn a policy, so in the last action layer we would like to output a single probability $\pi(a \mid s)$ for each action $a \in \mathcal{A}$, rather than producing another d_h-dimensional hidden representation.

We will begin by considering the changes which need to be made to the outputs of modules in the final layer. We will assume that this is the $L+1$th action layer—in other words, there are L pairs of action and proposition layers beforehand. First, we need to ensure that each module outputs a scalar, rather than a d_h-dimensional hidden representation. We can do this by using a weight vector $W_a^{L+1} \in \mathbb{R}^{d_a^{L+1}}$ in place of our previous weight matrix, and a scalar bias $b_a^{L+1} \in \mathbb{R}$ in place of our previous vector-valued bias. Second, we need to ensure both that the network cannot select an action a for which $\text{pre}(a)$ does not hold in the current state s, and that the probabilities of actions which can be selected add up to 1. We can achieve both of these tasks by concatenating the outputs of our action modules and passing them through a *masked softmax* function. To determine which of the actions $a_1, \ldots, a_N \in \mathcal{A}$ are enabled or disabled, the network takes as input a binary mask vector $m = \begin{bmatrix} m_1 & \ldots & m_N \end{bmatrix}^T \in \{0,1\}^N$, where $m_i = 1$ iff $\text{pre}(a_i)$ holds in the current state. Given the scalar outputs $\phi_{a_1}^{L+1}, \ldots, \phi_{a_N}^{L+1} \in \mathbb{R}$ of the action modules in the $L+1$th layer, an ASNet can produce a series of action selection probabilities $\pi_1, \ldots, \pi_N$, where

$$\pi_i = \frac{m_i \exp\left(\phi_{a_i}^{L+1}\right)}{\sum_{j=1}^{N} m_j \exp\left(\phi_{a_j}^{L+1}\right)} . \tag{3.4}$$

This ensures that $\pi_i = 0$ for each a_i where $\text{pre}(a_i)$ does not hold. Further, it guarantees that the probability distribution over enabled actions is appropriately normalised, assuming that at least one action is enabled.

Now we can consider the changes which need to be made to the input spaces of action modules in the first layer of the network. Rather than receiving hidden represen-

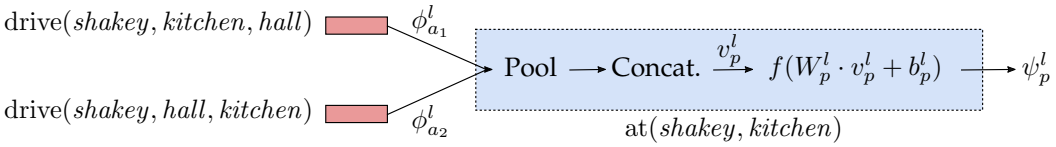

Figure 3.4: Illustration of a proposition module for the $\mathrm{at}(shakey, kitchen)$ proposition in the Unreliable Robot problem (Figure 2.2). As input, the module receives hidden representations $\phi^l_{a_1}$ and $\phi^l_{a_2}$ for the two actions which are related to it. It then pools those vectors together, since they are associated with actions of the same schema, and uses the pooled vector to obtain a new hidden representation ψ^l_p. Since all related actions are of the same schema, there is only one pooling operation, and the subsequent concatenation operation is trivial—it just lets that single pooled vector pass through unchanged. In general, if there are K schemas, then there will be K separate pooling operations, and the results of those pooling operations will be joined by the concatenation operation.

tations from proposition modules, an action module in the first layer receives features derived from the state. This includes truth values for related propositions, an indication of whether each related proposition appears in the goal, and an indication of whether the action is enabled. To be precise, a first-layer module for an action a_i is given a feature vector

$$u^1_{a_i} = \begin{bmatrix} v^T & g^T & m_i \end{bmatrix}^T . \tag{3.5}$$

Assume that the related propositions for a_i are $p_1, \ldots, p_M$. In this case, $v \in \{0,1\}^M$ is a vector of truth values for the propositions $p_1, \ldots, p_M$. We have $v_i = 1$ if proposition p_i is true in the current state, and $v_i = 0$ otherwise. Similarly, $g \in \{0,1\}^M$ indicates whether each proposition p_i appears in the goal for the problem ($g_i = 1$) or not ($g_i = 0$). Finally, as noted above the mask value $m_i \in \{0,1\}$ indicates whether a_i is enabled or disabled in the current state.

In our definition of the goal information vector g, we have implicitly assumed that the goal is always a conjunction of unnegated propositions. Supporting more complex goals—for example, arbitrarily nested conjunctions and disjunctions—would require changes to our input scheme. However, we found that most PPDDL benchmark problems expressed their goals as conjunctions of unnegated propositions, or could be trivially modified so that their goals would take this form. Hence, we do not consider support for more complex goals in this thesis.

3.1.3 Proposition modules

Between each pair of action layers is a proposition layer, composed of a single proposition module for each proposition in the planning problem. Proposition modules are structured in a similar way to action modules. Specifically, a proposition module for proposition $p \in \mathcal{P}$ in the lth proposition layer of the network will compute a hidden representation

$$\psi^l_p = f(W^l_p \cdot v^l_p + b^l_p) . \tag{3.6}$$

f is the same nonlinearity used before, while $W^l_p \in \mathbb{R}^{d_h \times d^l_p}$ and $b^l_p \in \mathbb{R}^{d_h}$ are learnt weights and biases for the module, respectively. v^l_p is a feature vector which serves as input to

the proposition module; we discuss how to construct this below. Because proposition modules cannot appear in the first or last layer of a network, we can use this same pattern to construct every proposition module in the network. We do not need special input spaces for modules in the first layer or special output spaces for modules in the last layer.

Construction of input vectors for proposition modules is more involved than construction of input vectors for action modules. The number of propositions which are relevant to an action—and thus the number of hidden representations which must be fed into an action modules—is the same for all action modules which correspond to the same action schema. This is true even if those actions are taken from different problems with different numbers of objects. In contrast, two propositions instantiated from the same predicate may have a different number of related actions depending on the structure of the problem. Hence, simply concatenating the hidden representations for these related actions would result in inputs of different sizes for some proposition modules instantiated from the same action schema, which would break the weight sharing scheme which we propose in Section 3.2. As an example, we can return to our Unreliable Robot domain, where the location of a robot is tracked with robot-at propositions that can be manipulated by actions instantiated from a drive action schema. A location l_1 with one incoming road and no outgoing roads will have only one related drive action, which moves from the other location to l_1. In contrast, a location l_2 with two incoming roads and no outgoing roads will have two related move actions—one action to move to l_2 along each road.

As noted in Section 2.3.3, existing graph convolutional architectures also have to accommodate nodes with varying indegree (Duvenaud et al., 2015; Jain et al., 2016), and we will solve the corresponding issue for our architecture in much the same way. Specifically, to construct the input v_p^l, we first find the predicate $\text{pred}(p) \in \mathcal{F}$ for proposition $p \in \mathcal{P}$. We then enumerate all action schemas $A_1, \ldots, A_S \in \mathbb{A}$ which reference $\text{pred}(p)$ in a precondition or effect. This allows us to define a feature vector

$$v_p^l = \begin{bmatrix} \text{pool}(\{\phi_a^l \mid \text{op}(a) = A_1 \wedge R(a,p)\}) \\ \vdots \\ \text{pool}(\{\phi_a^l \mid \text{op}(a) = A_S \wedge R(a,p)\}) \end{bmatrix}, \tag{3.7}$$

where $\text{op}(a) \in \mathbb{A}$ denotes the action schema for ground action a, and pool is a pooling function that combines several d_h-dimensional feature vectors into a single d_h-dimensional one. In this thesis, we assume that pool performs mean pooling, so that

$$\text{pool}(\{v_1, \ldots, v_K\}) = \frac{1}{K} \sum_{i=1}^{K} v_i \,. \tag{3.8}$$

It would be equally possible to use other pooling operations. For instance, in convolutional neural networks, it's common to pool over inputs by taking only maximum input (Krizhevsky et al., 2012), or by choosing an input at random (Zeiler and Fergus, 2013). However, we leave exploration of these alternatives for future work. When all pooled vectors are concatenated, the dimensionality d_p^l of v_p^l becomes $d_h \cdot S$. The number of schemas S for potentially-related actions is the same for all propositions instantiated from the same predicate, so this allows us to use the weight-sharing mechanism proposed in Section 3.2. A complete proposition module is shown in Figure 3.4.

3.2 Weight sharing

In order to use an ASNet as a *generalised* policy, we should be able to apply the same set of learnt weights to any problem from a given domain. We achieve this with a weight-sharing scheme: at each action layer l, and for each pair of ground actions c and d instantiated from the same action schema, we stipulate that $W_c^l = W_d^l$ and $b_c^l = b_d^l$. In other words, modules for actions which appear in the same layer and correspond to the same action schema will share weights, but modules which appear in different layers or which correspond to different schemas will learn different weights Likewise, at proposition layer l, and for propositions q and r with $\mathrm{pred}(q) = \mathrm{pred}(r)$, we tie the corresponding weights $W_q^l = W_r^l$ and $b_q^l = b_r^l$ for both training and evaluation. Together with the weight sharing scheme for action modules, this enables us to learn a single set of weights

$$\begin{aligned}\theta =&\{W_a^l, b_a^l \mid 1 \leq l \leq L+1, a \in \mathbb{A}\} \\ \cup&\{W_p^l, b_p^l \mid 1 \leq l \leq L, p \in \mathcal{F}\}\end{aligned} \tag{3.9}$$

for an L-layer model. Because the number and size of weights depends only on the predicate set $\mathcal{F}$ and action schema set $\mathbb{A}$—both of which are defined in the domain—the weights can be applied to any problem from a domain.

From an implementation perspective, enforcing that two weight matrices W_1 and W_2 satisfy $W_1 = W_2$ is trivial. We simply treat W_1 and W_2 as the same matrix W, and use the multivariate chain rule to derive a gradient with respect to W during optimisation, as usual. This is standard practice, and is well-supported by deep learning frameworks, including the Tensorflow framework which we used to implement ASNets (Abadi et al., 2016).

In Section 2.3, we noted that similar weight-sharing schemes have been used by other structured deep learning methods. Indeed, convolutional neural networks (LeCun et al., 1995) and several of the spatial graph convolution architectures which we considered (Duvenaud et al., 2015; Jain et al., 2016; Kearnes et al., 2016) use forms of weight-sharing which make the number and shape of weights to be learnt independent of problem size. LeCun et al. (1989) point out that weight sharing can have other benefits, too. Because architectures which share weights typically have few parameters, it is often possible to learn a good representation for the task at hand with very little data. In contrast, architectures without weight sharing may have redundant weights which need to behave in the same way, and will require more diverse training data to ensure that those weights take similar values. For instance, if we applied an ASNet without weight sharing to a fixed-size navigation problem in our Unreliable Robot domain, we would have to ensure that the training set included states with the robot in every possible position. Otherwise, some of the modules for robot-at propositions may not be able to learn appropriate weights, and the resultant network would be unprepared to choose a good action at the corresponding locations. In contrast, weight-sharing allows all modules for robot-at propositions to use the same weights, so we can get away with a training set in which the robot appears at only a few locations.

3.3 Heuristic inputs

One limitation of an ASNet is its effective receptive field. Recall from Section 2.3.2 that the effective receptive field of a single unit in a 2D CNN is the window of pixels in the

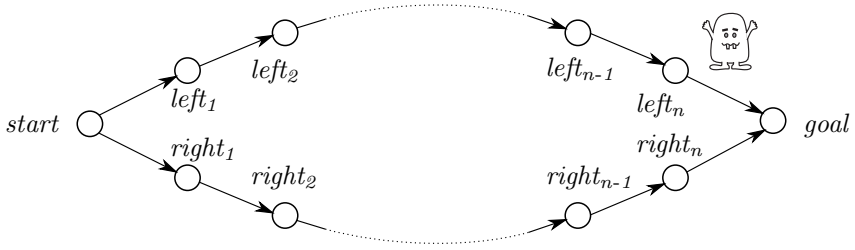

Figure 3.5: Illustration of the form taken by problems from the Monster domain. While the wumpus is shown on the top path in this diagram, real problems from the wumpus domain randomly choose which path to place the wumpus on using a pseudo-action which must be executed by the agent. Wumpus reproduced from Russell and Norvig (1995).

input image which can influence that unit's output. The size of this patch is larger for units later in the network, since each layer is able to propagate information about the input image slightly further. In an ASNet, the receptive field of a unit in an action or proposition module is the set of input propositions which can influence its output. As in CNNs, the effective receptive fields of units in action and proposition modules grow as more layers are added. The difference is that the effective receptive fields expand according to the relatedness $R(\cdot, \cdot)$ of actions and propositions, rather than the spatial distance between pixels. Because an ASNet will have a fixed depth, its effective receptive field limits the length of chains of related actions and propositions which it can reason about.

To illustrate the practical limitations of a fixed receptive field, we will consider a new domain which we call "Monster". Figure 3.5 shows the structure of Monster problems: the agent starts at an initial location where it can choose between two outgoing paths. There are n locations along each path, and the agent must apply $n + 1$ `drive` actions to move from the start to the goal through one chain of those locations. At the final location along of one of the paths sits a *wumpus*, which is a creature that loves the taste of agents and will try to eat any which it encounters (Russell and Norvig, 1995). In the initial state, the agent must take a pseudo-action whose only effect is to randomly place the wumpus on one of the two paths. The agent is then able to choose between two actions: one action which moves it onto one path to the goal, and one action which moves it onto the other. The agent cannot turn back once it has started down a path, so its objective is to choose the path that is wumpus-free.

Because the problem is fully-observable, a hand-coded policy would have no trouble detecting where the wumpus is placed and choosing the opposite path. However, this task is more difficult for ASNets: a specific ASNet may be able to solve Monster problems of bounded size, it is not possible for an ASNet to obtain a generalised policy for Monster problems. The domain definition in Appendix A.5 shows that $\mathrm{drive}(?from, ?to)$ actions in the Monster domain include a $\mathrm{has\text{-}monster}(?from)$ check which controls whether or not the agent should be killed. Hence, the action $\mathrm{drive}(start, right_1)$ is only related to

$\text{monster-at}(\mathit{right}_n)$ by a chain of relations of the following form:

$$
\begin{array}{c}
R(\text{drive}(\mathit{start}, \mathit{right}_1), \text{robot-at}(\mathit{right}_1)) \\
\downarrow \\
R(\text{robot-at}(\mathit{right}_1), \text{drive}(\mathit{right}_1, \mathit{right}_2)) \\
\downarrow \\
R(\text{drive}(\mathit{right}_1, \mathit{right}_2), \text{robot-at}(\mathit{right}_2)) \\
\downarrow \\
\vdots \\
\downarrow \\
R(\text{drive}(\mathit{right}_n, \mathit{finish}), \text{has-monster}(\mathit{right}_n))
\end{array}
$$

Each action layer propagates information from proposition modules to directly related action modules, and each proposition layer propagates information from action modules to directly related proposition modules. Further, the truth value for a proposition is only supplied to first-layer action modules which are related to that proposition. Hence, propagating information about the value of the $\text{has-monster}(\mathit{right}_n)$ proposition to the $\text{drive}(\mathit{start}, \mathit{right}_1)$ module requires an ASNet of $n+1$ action layers and n proposition layers. Analogous comments hold for comments along the left path. The upshot is that a network with fewer than n proposition layers will be unable to discriminate which of the first two drive actions to take, and will thus be unable to learn an optimal policy.

We can partly compensate for the receptive field limitations of an ASNet by supplying the network with features derived from domain-independent planning heuristics. In this thesis, we will derive such features from disjunctive action landmarks produced by LM-cut. Recall from Section 2.1.4 that a disjunctive action landmark for a deterministic planning problem (or a determinised probabilistic planning problem) is a set of actions for which at least one action must be taken on any path to the goal. If an action is the *only* action in a disjunctive action landmark, then it is always the case that it should be invoked before reaching the goal. Similarly, if an action is one action in a disjunctive landmark of several actions, then it's more likely to be useful than if it is not in a landmark at all.

The domain given in Appendix A.5 is defined so that there is always a 1% chance that the monster will *not* kill the agent. Combined with the determinisation process—which allows the agent to assume that the best-case outcome happens all the time, instead of 1% of the time—this ensures that disjunctive action landmarks are still not informative for this problem. However, if we modified the domain so that the monster *always* kills the agent, then disjunctive action landmark sets could be refined to include only drive actions leading along the correct path, and an ASNet-based agent would consequently be able to learn a good generalised policy. In practice—and particularly in Chapter 5—we observe landmarks from LM-cut are often sufficiently to allow ASNet to overcome its depth limitations.

To make use of computed disjunctive action landmarks for a state, we need to augment the input spaces of action modules in the first layer of the network. Specifically, a module for action a_i in the first network layer is now given a feature vector

$$u^l_{a_i} = \begin{bmatrix} v^T & g^T & m_i & c_t \end{bmatrix}^T, \tag{3.10}$$

where g, m_i and v are defined the same way they were in Section 3.1.2. The new input $c \in \{0, 1\}^3$ indicates whether a_i is the sole action in at least one landmark ($c_1 = 1$), an action in a landmark of two or more actions ($c_2 = 1$), or does not appear in a landmark ($c_3 = 1$). No further changes need to be made in later layers.

On the whole, this strategy is similar to the way in which ROLLER (de la Rosa et al., 2011), a tree-based generalised policy learner, makes use of heuristic information. Rather than using LM-cut landmarks, ROLLER employs helpful actions extracted by the Fast Forward (FF) planner (Hoffmann, 2001). de la Rosa et al. note that such binary inputs have shortcomings in some domains, where numeric inputs like heuristic values can be more helpful. Hence, in future work, it would be interesting to investigate the use of numeric heuristic inputs for ASNets, rather than merely binary ones. For instance, we could use the operator counts produced by the planner of Trevizan et al. (2017), which give stochastic-action-aware approximation of the expected number of times each action will have to be executed under an optimal policy. We defer investigation of alternative features of this sort to future work.

Finally, it's worth comparing the receptive field limitations described above with the limitations of finite-horizon heuristic search planners. We explained the basic approach underlying such planners in Section 2.1.3. Roughly, finite-horizon planners relax an SSP so that only a fixed number of actions can be applied. The finite-horizon SSP produced by this relaxation can be solved through heuristic search. Solving this relaxed SSP is usually much faster than solving the original SSP, since the depth of the corresponding search tree is limited. Nevertheless, the actions chosen for the relaxed SSP can often work well in the original SSP, too. The disadvantage of this approach is that certain kinds of traps or opportunities which lie far in the future can be hidden from the planner. This is somewhat similar to the way that an ASNet may be unable to avoid traps which require reasoning about long chains of related actions and propositions. However, this apparent similarity belies the substantial differences between these two approaches. In Chapter 5, we will make the distinction concrete by showing that an ASNet can learn to avoid some kinds of long-term traps which finite-horizon SSP solver would fall prey to.

Chapter 4

Training and Exploiting Action Schema Networks

ASNets, which we described in the previous chapter, are only a representation for learnt knowledge. For a representation to be useful, there must also be some way of efficiently acquiring knowledge using that representation, and a way of exploiting it for planning. In this chapter, we propose an algorithm for efficiently learning weights for an ASNet, then briefly discuss the options which are available for using an ASNet-based policy to actually solve large probabilistic planning problems.

4.1 Training

To learn weights for an ASNet, we execute it on small problems from a domain, then train it to pick good actions in each state which it encounters. The objective of this process is to acquire a set of weights which will still form an effective policy on problems much larger than those in the training set. We found that supervised learning was the most effective form of training: in this setting, the agent attempts to mimic the actions of a *teacher policy* obtained through heuristic search. However, past work has also considered the use of reinforcement learning, where the agent receives a scalar reward indicating how well it is performing, but does not receive direct advice on which actions would work better. We consider the challenges to training an ASNet in this setting at the end of the section.

4.1.1 Supervised training algorithm

Our proposed algorithm for supervised training is depicted in Algorithm 1. At a high level, the algorithm executes over a number of *epochs*, each of which consists of an exploration phase and a learning phase. In the exploration phase, the algorithm repeatedly chooses a problem ζ from the set of training problems P_{train}. For each chosen problem the algorithm then samples several state trajectories to add to a state memory $\mathcal{M}$. In the learning phase, the network weights θ are optimised to increase the likelihood that the ASNet chooses the right actions for states in $\mathcal{M}$. We will now consider the components of the training algorithm in greater detail.

Initialisaton The training process starts by sampling a set of initial weights for the ASNet. In Algorithm 1, the weights are obtained using RANDOM-INITIAL-WEIGHTS, which samples each parameter in the weight set θ from a zero-centred Gaussian distribution.

Algorithm 1 Learning ASNet weights using a training problem set P_{train}

procedure ASNET-TRAIN
  $\mathcal{M} \leftarrow \varnothing$ ▷ State memory
  $\theta \leftarrow$ RANDOM-INITIAL-WEIGHTS() ▷ Weights
  $n \leftarrow 0$ ▷ Epoch counter
  repeat
    TRAIN-EPOCH($\theta, \mathcal{M}$)
    $n \leftarrow n + 1$
  until $n > T_{\text{max-epochs}}$ or early stopping condition satisfied
procedure TRAIN-EPOCH(θ, $\mathcal{M}$) ▷ Explores and learns for one epoch
  for $i = 1, \ldots, T_{\text{explore}}$ **do** ▷ Exploration (expand $\mathcal{M}$)
    for all $\zeta \in P_{\text{train}}$ **do**
      $s_0, \ldots, s_N \leftarrow$ RUN-POLICY($s_0(\zeta), \pi^\theta$)
      $\mathcal{M} \leftarrow \mathcal{M} \cup \{s_0, \ldots, s_N\}$
      for $j = 0, \ldots, N$ **do**
        $s^*_j, \ldots, s^*_M \leftarrow$ POLICY-ENVELOPE(s_j, π^*) ▷ States reachable under π^*
        $\mathcal{M} \leftarrow \mathcal{M} \cup \{s^*_j, \ldots, s^*_M\}$
  for $i = 1, \ldots, T_{\text{train}}$ **do** ▷ Learning (optimise weights)
    $\mathcal{B} \leftarrow$ SAMPLE-MINIBATCH($\mathcal{M}$)
    Update θ using $\frac{d\mathcal{L}_\theta(\mathcal{B})}{d\theta}$ (Equation 4.1)
function RUN-POLICY(s, π) ▷ Samples a trajectory
  $t \leftarrow 0$ ▷ Number of states observed
  $s_t \leftarrow s$ ▷ Current state
  while $s \notin \mathcal{G} \wedge t < T_{\text{trajectory-limit}} \wedge \neg$DEAD-END-DETECTED($s_t$) **do**
    $a \sim \pi(a \mid s_t)$
    $t \leftarrow t + 1$
    $s_t \sim \mathcal{T}(s_t \mid s_{t-1}, a)$
  return $s_0, \ldots, s_t$ ▷ Return all observed states

The standard deviations for those Gaussians are chosen to keep the variance of activations and gradients roughly the same across every layer of the network. The exact technique which we have employed for this purpose is known as *Glorot initialisation*, or *Xavier initialisation* (Glorot and Bengio, 2010).

Exploration After initialising the weights, ASNET-TRAIN calls TRAIN-EPOCH to begin the first epoch of training, which in turn initiates the first exploration phase. The exploration phase repeatedly executes the ASNet policy π^θ from the initial state of each training problem $\zeta \in P_{\text{train}}$. Each execution collects $N + 1$ states $s_0, \ldots, s_N$ visited under the policy. The final timestamp N for a given trajectory will be no greater than the limit $T_{\text{trajectory-limit}}$ on trajectory length, but could also be shorter if a goal or dead end is encountered. In addition, for each visited state s_j, TRAIN-EPOCH computes an optimal policy π^* so that it can extract the states $s^*_j, \ldots, s^*_M$ which form that optimal policy's *policy envelope*. In other words, it extracts the set of all states reachable with nonzero probability under the policy π^*. All states along the trajectories computed for the ASNet policy π^θ are added to $\mathcal{M}$, as are the states extracted from the policy envelope for π^*. Saving the states of π^*'s policy envelope ensures that $\mathcal{M}$ always contains states along "good" trajectories. On the other hand, saving trajectories from the exploration policy ensures that

ASNet will be able to improve on the states which it visits most often, even if they are not on an optimal goal trajectory.

To minimise time spent on exploration in each epoch, the algorithm makes use of early dead end detection. For the first few epochs, an ASNet will typically take more-or-less random actions during exploration. In problems with many dead ends, this random behaviour frequently leads the agent into traps where it is able to keep applying some actions, but can never reach the goal. This kind of behaviour can continue until the agent executes enough actions to reach the trajectory length limit ($T_{\text{trajectory-limit}}$), which is a highly inefficient use of exploration time. Hence, it is advantageous to detect such dead ends and terminate an agent's exploration early if a dead end is detected. During training, Algorithm 1 attempts to obtain and execute an optimal policy at each visited state, so dead-end states will have to be detected later anyway as part of the planning process. In practice, it's also possible to *immediately* detect some dead ends by evaluating a determinised, delete-relaxed heuristic in each visited state. If the heuristic is infinite, then a dead end has been reached. Because the heuristic only solves a relaxation of the original planning problem, it is not guaranteed to catch all dead ends. However, the dead ends which it does catch for the relaxed problem are necessarily also dead ends in the original problem. If RUN-POLICY detects a dead end in this way during training, it immediately cuts training short. This same technique can be used at test time, where there is no teaching policy available, thus making evaluation runs of the network slightly faster. Further, this technique does not incur any added computational overhead, as our heuristic input scheme already requires us to compute an LM-cut value $h^{\text{LM-cut}}(s)$ for each visited state s.

Learning After each exploration phase, Algorithm 1 moves into a learning phase. The objective of the learning phase is to ensure that the ASNet can select a good action for each state in $\mathcal{M}$. Specifically, we attempt to minimise the following loss:

$$\mathcal{L}_\theta(\mathcal{M}) = \frac{1}{|\mathcal{M}|} \sum_{s \in \mathcal{M}} \sum_{a \in A} \pi^\theta(a \mid s) \cdot Q^*(s, a) \; . \tag{4.1}$$

$Q^*(s, a)$ is the expected cost of reaching the goal from state s by taking action a and following policy π^* thereafter. These Q-values are obtained together with π^* computed by any optimal planning algorithm. At each optimisation step, we use an approximation of the gradient $\frac{d\mathcal{L}_\theta(\mathcal{M})}{d\theta}$ to update the weights θ in a direction which *minimises* $\mathcal{L}_\theta(\mathcal{M})$ using Adam (Kingma and Ba, 2014). Note, however, that we do not compute the loss L_θ with respect to the entire dataset $\mathcal{M}$; rather, we compute L_θ and $\frac{d\mathcal{L}_\theta}{d\theta}$ on to a randomly selected *minibatch* $\mathcal{B} \subseteq \mathcal{M}$ of fixed size. This strategy is called minibatch Stochastic Gradient Descent (SGD). Minibatch SGD is sometimes preferred to whole-dataset gradient descent due to the expense of computing gradients on a very large dataset. Even when the dataset is small, though, minibatch SGD can still yield faster convergence than full-dataset gradient descent (Li et al., 2014).

Termination The process of exploration and learning terminates when the maximum number of epochs $T_{\text{max-epochs}}$ is exceeded, or when an early stopping condition is met. Two conditions are required to hold for TRAIN-ASNET to terminate after the nth epoch. First, in the nth epoch, at least 99.9% of trajectories returned by RUN-POLICY for π^θ must reach a goal state. In other words, the success rate of the policy must be at least 99.9%

for the most recent epoch. Second, at least five epochs must have elapsed since the success rate of π^θ last increased by more than 0.01% over the previous best success rate. The objective of this early termination scheme is to allow the network to spend less time training on problems where it is possible to learn a generalised policy relatively quickly. For instance, in Chapter 5, we note that the Triangle Tire World domain has an optimal generalised policy which can easily be learnt in under a minute of training, while other domains may take much longer. We note that this early stopping policy is probably overly restrictive; for instance, in problems where it is not possible to obtain a near-perfect policy, the first necessary condition for early stopping will never hold. For such domains, it may be better to adjust the thresholds used for early stopping according to the success rate of the optimal teacher policy, π^*.

Relation to existing work

At this point, it is worth noting the similarities and differences between our proposed training algorithm and the existing training methods covered in Section 2.2. First we will consider the way in which we obtain states for our state memory $\mathcal{M}$.

Some states in $\mathcal{M}$ are obtained by extracting policy envelopes for an optimal policy from each visited state. The approach of pulling out the whole envelope of a single policy falls somewhere between the techniques of de la Rosa et al. (2011) and Yoon et al. (2002). For de la Rosa et al., the training set includes of all states reachable under *any* optimal policy for the problem. For Yoon et al., the training set includes only those states which were reached over a series of policy rollouts with a single near-optimal policy. Extracting an entire policy envelope for an optimal policy will yield at least as much training data as the latter, sample-based approach, but at no greater cost, since optimal probabilistic planners typically compute a policy envelope anyway. However, since we only consider a *single* optimal policy, our approach may leave out some states which are reachable under a different optimal policy, unlike the approach of de la Rosa et al.. On the other hand, de la Rosa et al. note that computation of all optimal policies can be extremely expensive, while our approach is much more efficient.

In addition to states included in optimal policy envelopes, $\mathcal{M}$ also includes states visited while executing the learnt policy π^θ during training. This can be interpreted as a form of hard negative mining, where we use the results of policy execution to find states where the policy does not yield good actions (Felzenszwalb et al., 2008). This technique was not used in the generalised policy learning work of de la Rosa et al. or Yoon et al., but a similar mechanism has previously been employed by Xu et al. (2010) to learn generalised heuristics.

The loss which we attempt to minimise is also substantially different to that used by previous approaches to generalised policy learning. Almost all the approaches in Section 2.2 attempt to minimise some form of classification error, where models are penalised for picking *any* action which is not optimal. In contrast, minimising $\mathcal{L}_\theta(\mathcal{M})$ is similar to minimising the expected cost to reach the goal from a state sampled from $\mathcal{M}$, under the assumption that an optimal policy is followed after the first sampled action. This loss also penalises the agent for not picking suboptimal actions, but the penalty is only proportional to the degree to which the action is suboptimal, so good-but-imperfect actions are still permissible. In general, the fact that our policy has a continuous parametrisation allows us a substantially more flexibility in choosing a suitable loss, as we noted in Section 2.2.3. In Section 4.1.2, we will see that this flexibility even allows us to train ASNets

using policy gradient reinforcement learning—something which would not be possible using a discrete model like a decision tree.

4.1.2 Training with reinforcement learning

Minimisation of the supervised objective in Equation (4.1) can lead to good policies in practice, but the overall approach is theoretically inelegant. On a given problem, what we really wish to minimise is $V^{\pi}(s_0)$—the expected cost of reaching the goal from the initial state under π^{θ}. With Policy Gradient Reinforcement Learning (PG RL), it is possible to optimise this quantity directly with an approximate form of gradient descent. Specifically, PG RL methods obtain an approximation $\widehat{\nabla_{\theta} V^{\pi}(s_0)}$ of the (intractable) gradient $\nabla_{\theta} V^{\pi}(s_0)$, then apply parameter updates of the form

$$\theta \leftarrow \theta - \lambda \widehat{\nabla_{\theta} V^{\pi}(s_0)} \tag{4.2}$$

for some small learning rate λ. The approximation $\widehat{\nabla_{\theta} V^{\pi}(s)}$ is obtained from sampled trajectories. Given a single trajectory $s_0, a_0, s_1, a_1, \ldots, s_{T-1}, a_{T-1}, s_T$ sampled under π^{θ}, and a cost C for that trajectory, one simple PG approximation is

$$\widehat{\nabla_{\theta} V^{\pi}(s)} = C \sum_{t=0}^{T} \nabla_{\theta} \log \pi^{\theta}(a_t \mid s_t) \,. \tag{4.3}$$

In practice, this approximation can have unacceptably high variance, and it is necessary to use variance reduction techniques like those presented by Baxter and Bartlett (2001) to obtain a good policy in a reasonable amount of time.

The approach described above is almost exactly the one taken by the FPG planner (Buffet and Aberdeen, 2009), which we covered in Section 2.2.3. FPG uses a PG RL variant (with several variance-reduction techniques) to train a neural-network-based policy $\pi^{\theta}(a \mid s)$ for a *single* probabilistic planning problem. Initially, we attempted to train ASNets in the same way. Unfortunately, we found that it could take tens of minutes or even hours to learn policies even the simplest benchmark problems. This was true for both ASNets and for the traditional (non-generalising) MLPs which FPG used, despite our use of modern Graphics Processing Units (GPUs) to execute the networks. Hence, we do not present results for RL-based training in our experiments. However, some sort of RL-augmented training could still prove useful, especially in conjunction with the supervised training mechanism presented earlier. For instance, an ASNet could be trained on small problems in a supervised manner, then fine-tuned on larger ones using only reinforcement learning. That would allow an ASNet with flawed knowledge to strengthen its performance on large problems, without the expense of computing a teacher policy for those problems. We leave this research direction for future work.

4.2 Exploitation

As noted in Section 2.2.4, existing work on generalised policy learning has generally assumed that policies are exploited within some sort of search framework. This allows the planner to verify that actions recommended by the generalised policy are beneficial, and to choose better actions if the recommended ones are flawed. In this thesis, we are primarily concerned with the ability of an ASNet to learn a robust generalised policy on its

own. Hence, for the experiments in Chapter 5 we use the ASNet's recommended actions directly, with no supporting search framework. Specifically, in a state s, we choose the action $a = \operatorname{argmax}_a \pi^\theta(a \mid s)$—that is, the action which maximises the action-selection probability $\pi^\theta(a \mid s)$. This allows us to avoid *ever* executing the (typically poor) actions which the ASNet recommends with low probability. Still, we note that the search-based exploitation mechanisms outlined in Section 2.2.4 could easily be used in conjunction with ASNets, particularly the rollout-based mechanisms of Fern et al. (2004a) and Yoon et al. (2007). Exploitation of learnt policies using Monte Carlo tree search—as AlphaGo does with neural-network-based policies for the game of Go (Silver et al., 2016)—could also be effective. As future work, Section 6.2.5 suggests unifying these exploitation mechanisms with the planning mechanism used to compute the teacher policy. A unified system could improve the effectiveness of planning on test problems *and* reduce the cost of obtaining teacher policies during training.

Chapter 5

Empirical Evaluation

Having introduced ASNets in Chapter 3, and proposed a mechanism to train them in Chapter 4, we will now consider how ASNets stack up against state-of-the-art probabilistic planners. Specifically, our evaluation will focus on the following questions:

When is it worth training an ASNet? ASNets aim to scale to problems beyond the reach of traditional heuristic search planners by using small training problems to learn a generalised policy for an entire domain. However, if only a handful of small problems need to be solved, then training an ASNet may be more expensive than using heuristic search to solve those problems directly. In part, our experiments are intended to characterise how expensive it is to train an ASNet, and when that training pays off.

Are ASNets doing fixed-depth lookahead in state space? In Section 3.3, we noted that the reasoning limitations imposed by the fixed receptive field of an ASNet appear to be similar to the limitations of planners which do fixed-depth lookahead. Our experiments will highlight the differences between the classes of problems which these two approaches are capable of solving.

When are heuristic input features useful? To overcome limitations imposed by the fixed receptive field of an ASNet, Section 3.3 proposed supplying ASNets with information about disjunctive action landmarks uncovered by LM-cut. In experiments, we are interested both in determining when these heuristic input features are actually necessary, and in characterising the sorts of problems for which our chosen heuristic input features are insufficient.

Can ASNets be trained by a suboptimal teacher? In Section 4.1, we suggested a supervised loss which required a teacher policy to provide an optimal Q-value $Q^*(s, a)$ for each visited state s and each applicable action a. Computing an optimal teacher policy can be much more expensive than computing a suboptimal one, so we will also perform a series of experiments to ascertain whether a suboptimal teacher suffices to train an ASNet.

5.1 Experimental setup

Our experiments follow a common pattern: for each domain, we produce a set of small training problems, and a set of larger evaluation problems. We first train a several ASNet configurations using the domain's training problems, then execute each of the resultant policies for 30 trials on each evaluation problem. Each evaluation problem is also solved

30 times by each baseline planner. In each of those 30 runs, the baseline is allowed a fixed amount of time for planning, then performs a single trial in a simulator. For each baseline or ASNet configuration, and each evaluation problem, we report the *coverage* of the planner—that is, what proportion of the 30 trials reached a goal state—and the mean length of successful trials. Further, we plot a comparison between the time taken to train and execute the ASNet in each evaluation problem, and the time which each baseline takes to solve each evaluation problem. We will now consider the configurations of our ASNets, the parameters of our baselines, and the characteristics of our domains.

5.1.1 ASNet configuration

Except for our experiments on the Monster domain (described below), all experiments used a common ASNet configuration. Our ASNets use three action layers and two proposition layers, with a hidden representation size of $d_h = 16$. In each epoch, we sample $T_{\text{explore}} = 100/|P_{\text{train}}|$ trajectories for each training problem in P_{train}, then perform $T_{\text{train}} = 300$ minibatches of optimisation during the learning phase. For learning, we configure the Adam optimiser with a batch size of 128 and a learning rate of 0.0005, but otherwise leave its default parameters (Kingma and Ba, 2014) unchanged. Trajectories sampled during exploration and testing are limited to $T_{\text{trajectory-limit}} = 300$ actions, after which the network is assumed to be stuck in a dead end. We train the ASNets with one LRTDP-based teacher that uses the $h^{\text{LM-cut}}$ heuristic—which will lead to an optimal policy—and one LRTDP-based teacher that uses the h^{add} heuristic—which may not lead to an optimal policy. We report results with both teacher configurations.

Each ASNet experiment was performed on an x86-64 server with 64GiB of RAM, using a single core clocked at 2.3GHz. We imposed a maximum time limit of two hours on training, but still allowed training to stop beforehand if the early termination condition described in Section 4.1 was satisfied. Each evaluation run of the learnt generalised policy was limited to 2.5 hours for parity with our baselines, although in our experiments this time limit was never reached.

5.1.2 Baseline probabilistic planners

We compare against three state-of-the-art planners: LRTDP, ILAO*, and SSiPP. All three were briefly described in Section 2.1.3, but we will again summarise the most important features of each planner here. LRTDP learns a (partial) optimal value function for a problem by performing a series of trials. In each trial, LRTDP chooses the action which results in the greatest payoff, according to its partial optimal value function; it updates its partial value function by performing Bellman backups on visited states. In contrast, ILAO* maintains an optimal policy on a subset of states known as a best partial solution graph; this graph is gradually expanded in the direction of the goal. ILAO* terminates once it can verify that the optimal policy for the states in the best partial solution graph is also an optimal policy for the original SSP. Since LRTDP and ILAO* employ very different mechanisms for planning, obtaining good results relative to both of them would be strong indication of the merits of our approach. SSiPP is a complementary approach which solves a series of *short sighted SSPs*—that is, SSPs which have been relaxed to limit the number of actions which can be taken. By solving a series of relaxed SSPs, SSiPP can converge to an optimal policy for the original SSP, even if the original SSP does not have a depth limit. In Section 3.3, we noted that finite-horizon solvers such as SSiPP have

seemingly similar limitations to ASNets. We have included SSiPP in our set of baselines so that we can empirically show how the two approaches differ.

All baselines were executed on a single core of an x86-64 CPU clocked at 2.6GHz; each run was limited to 10GiB of memory (which was never exhausted) and 2.5 hours of time. For logistical reasons, the baseline planners were executed on dedicated cores of a cluster, rather than cores of the single server used to evaluate the ASNets. However, it's worth noting that the difference in CPU clock is slightly in favour of the baselines, and we do not expect the difference in hardware to substantially affect our conclusions. We report results for each baseline planner using both the admissible LM-cut heuristic and the inadmissible h^{add} heuristic. The latter heuristic is more informative, but may cause the planner to converge to a suboptimal solution. The planner-specific settings for the baselines were as follows:

1. LRTDP and ILAO* were executed until their value estimates converged to within a tolerance of $\epsilon = 10^{-4}$. After this, the corresponding policy was evaluated for a single trial. If the planner did not converge within the allotted time, then no evaluation trial was executed, and the run was recorded as a failure.

2. SSiPP was configured to solve short-sighted SSPs of depth 3 using LRTDP. We used an early termination condition which would allow SSiPP to stop after 100 consecutive "practice" trials reached a goal state. If the early-termination condition was not triggered, then SSiPP's planning period would be terminated and its policy trialled as soon as there were 60s remaining in the planning period. In either case, upon stopping, SSiPP would execute a single evaluation trial. In contrast to LRTDP and ILAO*, this final trial occurred regardless of whether SSiPP's value estimates had converged.

5.1.3 Deterministic baseline planners

In addition to our experiments on probabilistic domains, we also evaluate one deterministic domain. State-of-the-art probabilistic planners are typically not state-of-the-art on deterministic problems, so we used a different, more competitive set of baselines for the deterministic domain. Our chosen deterministic baselines are all heuristic search planners, and were each implemented using the Fast Downward framework (Helmert, 2006). Briefly, each planner works as follows:

Greedy best-first (with $h^{\text{LM-cut}}$, h^{add}) Greedy best-first maintains a list of unexplored states, each with a matching heuristic value. At each iteration, it removes the state with the lowest heuristic value from the unexplored state list, adds the unexplored children of that state to the unexplored state list, and continues. Greedy best-first search can be extremely fast when equipped with an informative heuristic, but provides no guarantee of optimality.

A* (with $h^{\text{LM-cut}}$, h^{add}) A* works similarly to greedy best-first search, but ranks states in its *open list* using the sum of each state's heuristic value and the cost of the shortest known plan to reach the state. With an admissible heuristic, A* is guaranteed to eventually find an optimal solution, if one exists. Even without an admissible heuristic, A* can often find shorter paths than greedy best-first search, and is less vulnerable to uninformative heuristics (e.g. a heuristic which always reports a cost-to-go of zero). However, it is generally much more expensive than greedy best-first search.

LAMA-2011 and LAMA-first LAMA-2011, a variant of the LAMA planner (Richter et al., 2011), was one of the winners of the 2011 International Planning Competition (Coles et al., 2012). LAMA uses a combination of heuristics, including a landmark-based heuristic (Richter and Westphal, 2010) and the FF heuristic (Hoffmann, 2001). Both heuristics are able to provide an approximate cost-to-go for each state, and guidance about which actions are likely to be most helpful. LAMA's search strategy operates in two phases: the first, greedy phase tries to find *a* plan quickly, and subsequent A*-like phases try to find successively shorter plans until the available time expires. LAMA-first is a variation which uses only the first (greedy) phase of search performed by LAMA-2011.

The above baselines were executed on the same hardware as the ASNets, under the same time and memory limitations. Deterministic problems do not have any uncertainty in the outcomes of actions, so it suffices for a planner to find a single sequence of actions which can be executed in-order to reach a goal state. For this reason, we only executed a single trial of each baseline on each evaluation problem, and we are thus able to directly report the cost of the plan found (if any) instead of an expected cost over several trials.

5.1.4 Domains and problems

We first perform a full evaluation, as described at the beginning of this section, on three probabilistic domains: CosaNostra Pizza, Probabilistic Blocks World, and Triangle Tire World. To show that ASNets are able to solve deterministic problems, we also evaluate on a deterministic domain, Gripper, using the deterministic baseline planners described above. Further, to illustrate the limitations of ASNet's fixed receptive field, and the limitations of its input features, we perform a separate evaluation on the Monster domain, as described below. The PPDDL definitions for each domain are included in Appendix A. Note that we have assumed a dead-end penalty of 500 for all problems in all domains.

CosaNostra Pizza

As a Deliverator for CosaNostra Pizza (Stephenson, 1992), your job is to safely transport pizza from a pizza shop to a waiting customer, then return to the pizza shop. There is a series of toll booths between you and the customer: at each booth, you can either spend a time step paying the operator, or save a step by driving through without paying. However, if you do not pay, the (angry) operator will try to drop the tollbooth boom on your car when you pass through their booth on the way back to the shop, crushing your car with 50% probability. On the way from the pizza shop to the customer, you should always pay the operator of each toll booth. Otherwise, there is a high risk of your car being crushed when you pass through the same toll on the way back to the pizza shop. After returning from the customer to the pizza shop, you do not have to leave the pizza shop or pass through the toll booths again. Hence, an optimal policy will not pay any of the toll booth operators on the return trip, as it is no longer of any consequence whether they become angry. The challenge for an ASNet is to learn to pay toll booth operators on the way to the customer, but not pay them on the way back.

In our results, problem sizes for CosaNostra pizza indicate the number of toll booths between the pizza shop and the customer. ASNets are trained on problems of size 1-5, and all planners (including ASNets) are tested on sizes 6 and above.

Probabilistic Blocks World

The objective of Probabilistic Blocks World problems is to arrange several blocks into a series of towers by picking up blocks and depositing them on top of one another. The primary difference between Probabilistic Blocks World and the original deterministic version is that the probabilistic version allows blocks to slip onto the table. Specifically, picking up a block succeeds 75% of the time, but results in the block falling onto the table the remaining 25% of the time. Likewise, placing one block on another block succeeds with 75% probability, but may instead result in the top block falling onto the table with 25% probability. A variant of Probabilistic Blocks World featured in both the probabilistic tracks of the fourth (Younes et al., 2005) and fifth International Planning Competitions. The domain which we test on is based on the one featured in IPC-5; however, in contrast to the original IPC-5 domain, we have removed actions for stacking and unstacking entire towers of blocks. This brings the domain closer to original deterministic domain, where only one block may be moved at a time.

Reported problem sizes in Probabilistic Blocks World correspond to the number of blocks in each problem. For each problem size we consider, we generate random instances using the generator from Slaney and Thiébaux (2001). Each ASNet is trained on five generated problems on each size from 4-5, and two generated problems of each size from 6-8, for 16 training problems in total. We test on a different set of randomly generated problems of size 6 and above, with three generated problems per evaluated size.

For Probabilistic Blocks World, we have also included three new baselines. Each of the three baselines was obtained by treating each Probabilistic Blocks World problem as a deterministic problem, then running a blocks world baseline planner from Slaney and Thiébaux (2001). Each plan could then be turned into an expected cost for a corresponding Probabilistic Blocks World policy by using expected costs for each operation. Specifically, moving a block to the table has an expected cost of 1.75, and moving a block onto another block has an expected cost of 28/9. These costs account for the fact that a block could slip when it is picked up or put down, in which case the grasp or drop would have to be repeated until it succeeded. The three baselines use the following strategies to solve deterministic blocks world problems:

GN1 If GN1 can move a block to its goal position, then it does so. Otherwise, it chooses a block which is not in its goal position, and puts it on the table. This is repeated until the problem is solved.

GN2 A slight improvement on GN1 which is more selective when deciding which misplaced block to put on the table. Rather than choosing a misplaced block arbitrarily, it chooses a misplaced block which is in a *deadlock*. Concretely, a misplaced block b_1 is in a deadlock if there is a cycle of misplaced blocks $b_1, \ldots, b_k, b_{k+1}$ such that $b_1 = b_{k+1}$ and b_i must be moved before b_{i+1} can be put into position, for $1 \leq i \leq k$. This strategy is often able to yield shorter plans than GN1.

OPT Simply find an optimal solution.

Plans can be obtained using GN1 or GN2 in linear time. Computing a plan with OPT, however, is NP-hard. Refer to Slaney and Thiébaux (2001) for further explanation of the algorithms and relevant theoretical results.

Triangle Tire World

Each Triangle Tire World (Little and Thiébaux, 2007) environment consists of a set of locations arranged in a triangle, with connections between adjacent locations. An vehicle starts at one corner of the triangle, and the planner's objective is to move it to another corner by following the connections between locations. A move from location a to adjacent location b has a 50% chance of flattening the vehicle's tire. If there is a spare tire at b, then the vehicle's tire can be replaced; otherwise, the corresponding state is a dead end. Spare tires are placed around the two outside edges of the triangle, which form the longest path from the start state to the goal location. Hence, an ASNet must learn to sense and follow the outside edge of the triangle instead of taking the most direct route to the goal state. Little and Thiébaux (2007) observe that large Triangle Tire World instances can be challenging for planners which determinise probabilistic problems, or which use determinising heuristics. That is because both classes of planner are unable to perceive that the shortest goal trajectories are also the riskiest, and tend to get stuck trying to traverse the interior of the triangle instead of going around the outside.

Per Little and Thiébaux, we say that a problem of size n is one with $2n + 1$ locations along the shortest path from the start location to the goal location (that is, along the inside edge of the triangle). Such a problem thus has $(n + 1)(2n + 1)$ locations in total. We use problems of size 1-3 for training, and test with sizes from 4 onward.

Gripper

So far, we have only considered probabilistic problems. To demonstrate ASNet's ability to learn effective policies for deterministic problems, we evaluate on the deterministic Gripper domain (Long et al., 2000). Each Gripper problem consists of a robot with a pair of grippers, two rooms, and a set of balls in one of the rooms. The objective is to move all balls from the first room to the second. This can be achieved most efficiently by having the robot pick up two balls from the first room (one per gripper), moving to the second room, and depositing them. This process can be repeated until all balls have been moved.

The size of a Gripper problem is equal to the number of balls to be moved. We train ASNet on all problems of sizes 1-6, and test on problems of size 10+.

Monster

We described the Monster domain in Section 3.3, but we will briefly recapitulate the main points here. Each problem in the Monster domain is a basic navigation task with a start location, a goal location, and two one-way paths from start to goal. The first action invoked in each Monster problem will randomly place a hungry wumpus on one of the two paths at random. In a Monster instance of size n, the agent will then have to apply one action to move to the first location along one of the paths, then another n movement actions to reach the end of the path. There is only a 1% chance of the agent surviving if it encounters the wumpus, so it is necessary for the agent to look ahead when choosing a path in order to avoid the wumpus. In Section 3.3 this domain was used to illustrate the limitations which a fixed receptive field imposes on an ASNet. It also exposes a limitation in our choice of heuristic input features. Because there is at least a 1% chance of the agent reaching the goal along either path, our LM-cut landmarks—which are computed from an all-outcome determinisation—are insufficient to show where the wumpus is. Here,

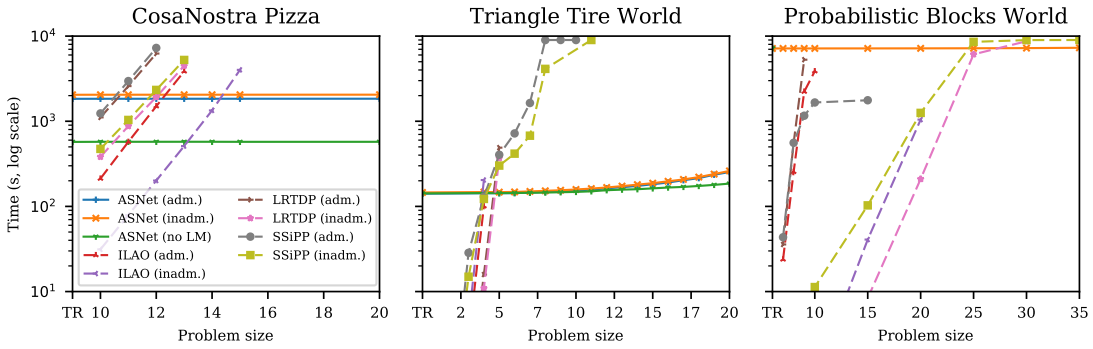

Figure 5.1: Comparison of planner running times on probabilistic evaluation domains. Refer to Section 5.2.1 for guidance on interpreting the plots.

we perform experiments on the domain to back up our theory: an ASNet with L proposition layers can only see along a path of length up to L, so it should be forced to choose at random when faced with paths of greater length.

Since the Monster domain was engineered to illustrate shortcomings of the ASNet approach, we evaluate it differently to the domains above. As observed in Section 3.3, Monster is trivial for heuristic search planners to solve. They need only expand both paths to the goal using time linear in the size of the problem, then check which gives the lowest expected cost. Hence, we do not report any baseline results for Monster. Instead, we report results for a series of ASNets with increasing depth. For each evaluated depth, we first train on *all* Monster problems of size 1-5, then test on the same set of problems. This allows us to be confident that an ASNet's inability to solve a given test problem is due to fundamental representational limitations, rather than simply because of overfitting or improper training. Other than the depth of the network, our ASNets are configured and trained in the way described in Section 5.1.1.

5.2 Results and discussion

In this section, we will intersperse presentation of our results with discussion of their implications.

5.2.1 Probabilistic domains

Figure 5.1 shows the running times for the benchmarked planning approaches across all probabilistic problems. Coverage and expected costs of goal trajectories are given in Table 5.1 (CosaNostra Pizza), Table 5.2 (Triangle Tire World), and Table 5.3 (Probabilistic Blocks World). Each of the graphs in Figure 5.1 shows problem size along the x-axis, and log-scaled running time along the y-axis.

Note that the "TR" point on the x-axis of Figure 5.1 indicates the time taken to train the ASNet—the baselines did not have to be trained, so their curves do not show any time for TR. Note that the reported evaluation times for the ASNet have been adjusted upward to include training time. Specifically, the reported time taken for an ASNet to solve a problem of size s is the time taken to train a generalised policy for the domain (show above as "TR"), *plus* the average time taken to simulate each of the 30 trajectories on the problem. This scheme favours the baselines. In practice, an ASNet would likely

be trained for the purpose of solving many large problems rather than just one, thus amortising the training cost over a larger number of evaluations. However, because we do not have a principled way of deciding how many large problems to solve, we have simply focused on how large a single evaluation problem ought to be before training an ASNet pays off.

To save space in the legend for Figure 5.1, we have used "(adm.)" to denote a baseline trained with the admissible $h^{\text{LM-cut}}$ heuristic, or to denote an ASNet whose teacher policy used $h^{\text{LM-cut}}$. Similarly, "(inadm.)" corresponds to baselines and teachers which used the inadmissible h^{add} heuristic. Further, "(no LM)" indicates that the ASNet was *not* given heuristic features from LM-cut as input, and that it was trained from a teacher policy obtained using h^{add}.

The expected cost tables also require some guidance to interpret. First, the ratio at the top of each cell shows the coverage for the corresponding planner, which is the number of trials (out of 30) that reached a goal state. Second, the parenthesised $(\mu \pm \text{CI})$ figure below each coverage statistic gives the mean cost (μ) of goal trajectories along with a corresponding 95% confidence interval (CI) for mean cost. Note that the latter two statistics are only calculated over trajectories which actually reached a goal state, so they are missing from cells where the planner obtained zero coverage, which have been marked with a "-". The advantage of only calculating cost statistics over goal trajectories is that it allows one to obtain a value $V^\pi(s_0)$ for a policy under any (finite or infinite) dead-end penalty. If the coverage of a problem is c (where $0 \leq c \leq 1$), and the mean cost of goal trajectories is v, then we have $V^\pi(s_0) \approx c \cdot v + (1 - c) \cdot D$ for a chosen dead-end penalty D.

We will begin our interpretation of the experiments with the results for CosaNostra Pizza. As Figure 5.1 demonstrates, ASNet sometimes underperforms on small instances, but massively outperforms all baselines as the problem sizes increase. The baseline without LM-cut flags did particularly well, as it was able to avoid the expense of computing $h^{\text{LM-cut}}$ at each state. It's worth reiterating the y-axes for the plots in Figure 5.1 are on a log scale, so the running times of baselines are increasing exponentially. Table 5.1 shows that the ASNet was in fact able to obtain an optimal policy; the baselines also managed to obtain optimal policies for instances they were able to solve, but the falloff in coverage was sharp as problem size increased. Intuitively, this is not a surprising result: CosaNostra merely requires an agent to learn the "trick" of paying the operator when travelling in one direction, and not paying in the other. Without some ability to learn, a planner will have to rediscover this trick at every toll booth. Further, discovering this trick requires a difficult form of long-term reasoning, since the consequences of not paying the operator of a booth do not become obvious until the agent returns to the same booth on its way back to the shop.

The story for Triangle Tire World—with times in Figure 5.1, and coverage in Table 5.2—is similar. Each ASNet is again able to learn the domain-specific knowledge required to solve large instances, and is able to leapfrog non-learning planners as a result, this time with even less training. Further, avoiding the interior of the triangle only requires a fixed receptive field, so ASNet is able to perform well even without LM-cut flags. SSiPP's behaviour on Triangle Tire World also deserves comment: not only does it manage to solve substantially larger instances than the other baselines, but it also has a shallow falloff in coverage on very large problems. We speculate that the short-sighted SSPs solved by SSiPP allow it to detect that moves away from the outside of the triangle are more likely to lead to dead ends than moves along the edge. However, SSiPP must rediscover this fact at every location along the outside edge of the triangle, so its

performance still falls away on larger problems.

The results for Probabilistic Blocks World—featured in Figure 5.1 and Table 5.3—are interesting for three reasons. First, the baseline results make it clear that the LM-cut heuristic is too expensive or too uninformative to scale in this domain. We speculate that this is the reason for ASNet's inability to learn a generalised policy with LM-cut—the added epochs of exploration allowed by a h^{add}-based teacher may allow the ASNet to learn domain-specific knowledge which is essential to proper generalisation. Second, SSiPP manages to obtain very good coverage on this domain. In fact, its coverage is not much worse than ASNet's, and its running time does not increase as rapidly as the other baselines. However, its huge mean trajectory costs on the larger Probabilistic Blocks World problems—up 3.7x that of the best ASNet—suggest that this good coverage is an artefact of SSiPP's ability to execute a partially-computed policy. Because Probabilistic Blocks World has no dead ends, SSiPP is free to meander around state space until it reaches a goal state, even if its value function estimates for the visited states are nowhere near convergence. In contrast, ASNet manages to learn a more reliable policy—although, unsurprisingly, it requires LM-cut-based heuristic inputs to do so. Third, the results for the deterministic-planner-based baselines (GN1, GN2 and OPT) are all slightly better than the ASNet results for at least a few problems. This suggests that the ASNet has not learnt to emulate GN1 or GN2, which are both simple linear time policies. It also shows that the ASNet has not learnt the optimal policy, although that is not surprising given that it is NP-hard (Slaney and Thiébaux, 2001).

Armed with these results, we can now answer some questions which we posed at the beginning of the chapter:

When is it worth training an ASNet? As expected, ASNets work better on large problems than small ones. Training an ASNet can incur minutes or hours of overhead, and each of the three evaluated domains included some evaluation problems which were too small to justify the added cost. However, Triangle Tire World and CosaNostra Pizza are good examples of the sorts of domain where an ASNet can still do well on small problems, since both domains feature problems with repeated traps that the ASNet can learn to avoid. Probabilistic Blocks World represents a middle ground—an ASNet can learn a reliable generalised policy, but the lack of dead ends means that the problem is not pathologically difficult for heuristic search planners.

Are ASNets doing fixed-depth lookahead in state space? SSiPP's poor performance in CosaNostra supports our earlier argument that ASNets are not merely another way of doing fixed-depth lookahead in state space. However, it's worth noting that SSiPP's ability to avoid short-term traps made it the only competitive baseline in Triangle Tire World and Probabilistic Blocks World. This suggests that state-space lookahead and ASNet-style "relatedness lookahead" may have partially overlapping strengths, even if they are distinct strategies.

When are heuristic input features useful? As we posited earlier, heuristic input features are useful (and in fact necessary) in problems where an agent must reason about recursive relationships between propositions to choose good actions. Probabilistic Blocks World is the most obvious example of such a domain, since the agent must reason about long chains of propositions instantiated from the $\text{on}(?\textit{top-block}, ?\textit{bottom-block})$ predicate to determine whether a block needs to be

moved. However, the results for Triangle Tire World in Figure 5.1 show that the cost of computing these features can also be significant for large problems. In particular, note how the curves for ASNets which used LM-cut-derived features began to angle up sharply beyond size 15. Further, in Section 5.2.3, we will see that heuristic input features are not always sufficient for this kind of reasoning.

Can ASNets be trained by a suboptimal teacher? The answer is an emphatic "yes". In the problems which we have evaluated, the potentially-suboptimal teacher which used h^{add} did not result in a dip in coverage, or a rise in the mean cost of goal trajectories. As noted in the case of Probabilistic Blocks World, the added speed of a h^{add}-based teacher may even be helpful for increasing coverage on some problems.

In Section 6.2, we suggest some ways in which future work could strengthen these results. In particular, Section 6.2.3 suggests some ways in which the expressiveness of the network could be increased while eliminating heuristic features, while Section 6.2.5 suggests a mechanism for reducing training time by using a partially-learnt generalised policy to speed up acquisition of a teacher policy.

Problem	ASNet			ILAO*		LRTDP		SSiPP	
	$h^{\text{LM-cut}}$	h^{add}	h^{add}, no LM	$h^{\text{LM-cut}}$	h^{add}	$h^{\text{LM-cut}}$	h^{add}	$h^{\text{LM-cut}}$	h^{add}
cosanostra-n10	30/30 (34.00 ± 0)	30/30 (34.00 ± 0)	30/30 (34.00 ± 0)	30/30 (34.00 ± 0)	30/30 (34.00 ± 0)	30/30 (34.00 ± 0)	30/30 (34.00 ± 0)	30/30 (34.00 ± 0)	30/30 (34.00 ± 0)
cosanostra-n11	30/30 (37.00 ± 0)	30/30 (37.00 ± 0)	30/30 (37.00 ± 0)	30/30 (37.00 ± 0)	30/30 (37.00 ± 0)	30/30 (37.00 ± 0)	30/30 (37.00 ± 0)	30/30 (37.00 ± 0)	30/30 (37.00 ± 0)
cosanostra-n12	30/30 (40.00 ± 0)	30/30 (40.00 ± 0)	30/30 (40.00 ± 0)	30/30 (40.00 ± 0)	30/30 (40.00 ± 0)	30/30 (40.00 ± 0)	30/30 (40.00 ± 0)	30/30 (40.00 ± 0)	30/30 (40.00 ± 0)
cosanostra-n13	30/30 (43.00 ± 0)	30/30 (43.00 ± 0)	30/30 (43.00 ± 0)	30/30 (43.00 ± 0)	30/30 (43.00 ± 0)	-	30/30 (43.00 ± 0)	-	30/30 (43.00 ± 0)
cosanostra-n14	30/30 (46.00 ± 0)	30/30 (46.00 ± 0)	30/30 (46.00 ± 0)	-	30/30 (46.00 ± 0)	-	-	-	-
cosanostra-n15	30/30 (49.00 ± 0)	30/30 (49.00 ± 0)	30/30 (49.00 ± 0)	-	30/30 (49.00 ± 0)	-	-	-	-
cosanostra-n20	30/30 (64.00 ± 0)	30/30 (64.00 ± 0)	30/30 (64.00 ± 0)	-	-	-	-	-	-

Table 5.1: CosaNostra Pizza coverage for a selection of problems and planners, along with mean cost to reach the goal and 95% CI for cost in brackets. Refer to Section 5.2.1 for guidance on interpreting these figures.

Problem	ASNet			ILAO*		LRTDP		SSiPP	
	$h^{\text{LM-cut}}$	h^{add}	h^{add}, no LM	$h^{\text{LM-cut}}$	h^{add}	$h^{\text{LM-cut}}$	h^{add}	$h^{\text{LM-cut}}$	h^{add}
triangle-tire-4	30/30 (23.37 ± 0.66)	30/30 (23.37 ± 0.66)	30/30 (23.37 ± 0.66)	30/30 (23.17 ± 0.69)	30/30 (23.17 ± 0.69)	30/30 (23.83 ± 0.71)	30/30 (24.10 ± 0.76)	30/30 (23.23 ± 0.76)	30/30 (23.33 ± 0.80)
triangle-tire-5	30/30 (28.87 ± 0.81)	30/30 (28.87 ± 0.81)	30/30 (28.87 ± 0.81)	-	-	30/30 (29.27 ± 0.75)	30/30 (30.23 ± 0.63)	30/30 (29.43 ± 0.85)	30/30 (29.83 ± 0.79)
triangle-tire-6	30/30 (34.87 ± 0.94)	30/30 (34.87 ± 0.94)	30/30 (34.87 ± 0.94)	-	-	-	-	30/30 (35.70 ± 0.96)	30/30 (36.90 ± 0.87)
triangle-tire-7	30/30 (40.77 ± 0.91)	30/30 (40.77 ± 0.91)	30/30 (40.77 ± 0.91)	-	-	-	-	30/30 (41.43 ± 0.86)	30/30 (44.33 ± 1.05)
triangle-tire-8	30/30 (46.83 ± 1.12)	30/30 (46.83 ± 1.12)	30/30 (46.83 ± 1.12)	-	-	-	-	7/30 (48.00 ± 3.66)	26/30 (50.77 ± 1.05)
triangle-tire-9	30/30 (52.93 ± 1.27)	30/30 (52.93 ± 1.27)	30/30 (52.93 ± 1.27)	-	-	-	-	1/30 (54.00)	1/30 (71.00)
triangle-tire-10	30/30 (59.00 ± 1.11)	30/30 (59.00 ± 1.11)	30/30 (59.00 ± 1.11)	-	-	-	-	1/30 (60.00)	-
triangle-tire-11	30/30 (64.77 ± 1.08)	30/30 (64.77 ± 1.08)	30/30 (64.77 ± 1.08)	-	-	-	-	-	-
triangle-tire-12	30/30 (71.07 ± 1.21)	30/30 (71.07 ± 1.21)	30/30 (71.07 ± 1.21)	-	-	-	-	-	-
triangle-tire-13	30/30 (76.90 ± 1.21)	30/30 (76.90 ± 1.21)	30/30 (76.90 ± 1.21)	-	-	-	-	-	-
triangle-tire-14	30/30 (82.80 ± 1.35)	30/30 (82.80 ± 1.35)	30/30 (82.80 ± 1.35)	-	-	-	-	-	-
triangle-tire-15	30/30 (88.67 ± 1.37)	30/30 (88.67 ± 1.37)	30/30 (88.67 ± 1.37)	-	-	-	-	-	-
triangle-tire-16	30/30 (94.83 ± 1.29)	30/30 (94.83 ± 1.29)	30/30 (94.83 ± 1.29)	-	-	-	-	-	-
triangle-tire-17	30/30 (100.80 ± 1.21)	30/30 (100.80 ± 1.21)	30/30 (100.80 ± 1.21)	-	-	-	-	-	-
triangle-tire-18	30/30 (106.50 ± 1.44)	30/30 (106.50 ± 1.44)	30/30 (106.50 ± 1.44)	-	-	-	-	-	-
triangle-tire-19	30/30 (112.50 ± 1.56)	30/30 (112.50 ± 1.56)	30/30 (112.50 ± 1.56)	-	-	-	-	-	-
triangle-tire-20	30/30 (118.43 ± 1.48)	30/30 (118.43 ± 1.48)	30/30 (118.43 ± 1.48)	-	-	-	-	-	-

Table 5.2: Table 5.1 repeated for the Triangle Tire World domain.

Problem	ASNet		ILAO*		LRTDP		SSiPP		Det. ref.		
	$h^{\text{LM-cut}}$	h^{add}	$h^{\text{LM-cut}}$	h^{add}	$h^{\text{LM-cut}}$	h^{add}	$h^{\text{LM-cut}}$	h^{add}	GN1	GN2	OPT
prob-bw-n10-s1	-	30/30 (24.60 ± 1.97)	30/30 (25.50 ± 1.90)	30/30 (26.87 ± 1.71)	-	30/30 (24.03 ± 1.23)	30/30 (78.47 ± 24.76)	30/30 (25.03 ± 1.57)	24.31	24.31	24.31
prob-bw-n10-s2	-	30/30 (34.47 ± 1.80)	-	30/30 (36.37 ± 1.96)	-	30/30 (34.27 ± 1.64)	14/30 (484.50 ± 173.12)	30/30 (35.27 ± 1.79)	34.03	34.03	34.03
prob-bw-n10-s3	-	30/30 (28.73 ± 2.17)	-	30/30 (29.90 ± 2.01)	-	30/30 (28.13 ± 1.77)	30/30 (127.20 ± 33.51)	30/30 (28.60 ± 1.71)	27.42	27.42	27.42
prob-bw-n15-s1	-	30/30 (49.83 ± 2.52)	-	30/30 (48.87 ± 2.83)	-	30/30 (50.10 ± 1.92)	30/30 (94.23 ± 10.12)	30/30 (51.27 ± 1.47)	44.72	42.97	42.97
prob-bw-n15-s2	-	30/30 (55.40 ± 2.18)	-	30/30 (57.67 ± 2.63)	-	30/30 (57.10 ± 2.49)	30/30 (185.00 ± 33.55)	30/30 (58.60 ± 2.00)	53.08	51.33	51.33
prob-bw-n15-s3	-	30/30 (47.13 ± 2.21)	-	30/30 (46.40 ± 2.49)	-	30/30 (45.13 ± 1.83)	30/30 (249.20 ± 50.41)	30/30 (46.00 ± 2.07)	37.72	37.72	37.72
prob-bw-n20-s1	-	30/30 (69.00 ± 2.75)	-	30/30 (69.63 ± 2.54)	-	30/30 (70.70 ± 3.36)	-	30/30 (70.00 ± 2.87)	64.17	60.67	60.67
prob-bw-n20-s2	-	30/30 (73.23 ± 2.71)	-	30/30 (73.87 ± 2.17)	-	30/30 (79.10 ± 2.73)	-	30/30 (83.53 ± 3.16)	69.42	69.42	69.42
prob-bw-n20-s3	-	30/30 (70.57 ± 2.86)	-	30/30 (74.60 ± 2.82)	-	30/30 (76.27 ± 3.44)	-	30/30 (78.20 ± 3.29)	72.53	62.03	62.03
prob-bw-n25-s1	-	30/30 (96.67 ± 2.92)	-	-	-	17/30 (100.94 ± 4.60)	-	28/30 (323.96 ± 91.46)	94.69	91.19	91.19
prob-bw-n25-s2	-	30/30 (92.93 ± 2.44)	-	-	-	27/30 (100.78 ± 3.16)	-	30/30 (145.63 ± 27.99)	87.50	85.75	84.00
prob-bw-n25-s3	-	30/30 (84.10 ± 2.90)	-	-	-	15/30 (95.73 ± 5.99)	-	29/30 (163.41 ± 35.31)	85.36	81.86	80.11
prob-bw-n30-s1	-	30/30 (116.07 ± 2.94)	-	-	-	2/30 (107.50 ± 44.47)	-	27/30 (340.37 ± 63.31)	114.14	105.39	103.64
prob-bw-n30-s2	-	30/30 (112.87 ± 2.93)	-	-	-	-	-	21/30 (418.38 ± 82.95)	108.31	101.31	101.31
prob-bw-n30-s3	-	30/30 (117.67 ± 3.66)	-	-	-	-	-	16/30 (373.31 ± 83.93)	114.53	109.28	109.28
prob-bw-n35-s1	-	30/30 (138.83 ± 3.35)	-	-	-	-	-	1/30 (366.00)	134.94	127.94	124.44
prob-bw-n35-s2	-	30/30 (137.00 ± 3.12)	-	-	-	-	-	3/30 (283.67 ± 199.76)	130.47	130.47	128.72
prob-bw-n35-s3	-	30/30 (139.53 ± 3.96)	-	-	-	-	-	6/30 (287.33 ± 137.80)	134.94	120.94	119.19

Table 5.3: Table 5.2 repeated for the Probabilistic Blocks World domain. "Det. ref." is short for "deterministic reference"—refer to description of the Probabilistic Blocks World domain in Section 5.1.4 for an explanation of the three corresponding baselines (GN1, GN2, OPT). Only expected costs are given for GN1/GN2/OPT; those costs were computed analytically for a policy with a 100% success rate, so no trials were necessary.

5.2.2 Deterministic domain

Our results for the Gripper domain—including both running times and the lengths of discovered plans—are included in Table 5.4. Running times are displayed in the same way that they were for probabilistic problems. However, as noted above, the notions of "cost" and "coverage" are different for a deterministic problem, so our table of costs shows only the length of the plan produced by each planner (if one could be found). Clearly, the results show that ASNets are able to solve deterministic problems: two ASNets were able to learn optimal policies for the problem using either the $h^{\text{LM-cut}}$ or h^{add} heuristic, and those policies generalised to all test problems. Combined with the fact that greedy best-first search could *not* find an optimal solution to any problem, this demonstrates that the ASNet has indeed learnt useful knowledge beyond that supplied by disjunctive action landmarks. Further, the ASNets are much more scalable than the A*-based planners, which time out even on small problems. However, the ASNets are still much slower than the LAMA baselines. LAMA manages to find optimal solutions for this problem extremely quickly, so the high cost of training an ASNet (around 1,200s in this case) simply does not pay off. It is likely that a more sophisticated domain—like the probabilistic domains which we considered previously—would yield an increased gap between the ASNet and LAMA.

One oddity of the results in Table 5.4 is the abysmal performance of the network which was not given LM-cut flags. Analysis of plan traces for the policy revealed that it had learnt to solve the set of training problems slightly faster than equivalent networks without LM-cut flags, and terminated early. However, its confidence in the correct action for each state was only barely sufficient to solve small problems, and decreased on larger problems, thus making the ASNet unable to generalise properly. The fact that a network with two proposition layers was able to solve problems up to size 10 suggests that the underlying issue was not the network's receptive field. Further, an analysis of the structure of the problem suggests that all relevant propositions should be in the receptive fields of key action modules anyway. This result suggests that the training procedure may possibly be too eager to terminate. Had it trained for slightly longer, it may have been able to increase the probability of obtaining the correct action on small problems, and been able to generalise to larger ones as well. We leave validation of this hypothesis to later work.

Problem	A*		ASNet			GBF		LAMA	
	$h^{\text{LM-cut}}$	h^{add}	$h^{\text{LM-cut}}$	h^{add}	h^{add}, no LM	$h^{\text{LM-cut}}$	h^{add}	-2011	-first
gripper-10	29	37	29	29	29	39	37	29	29
gripper-15	45	57	45	45	-	59	57	45	45
gripper-20	-	-	59	59	-	79	77	59	59
gripper-25	-	-	75	75	-	99	97	75	75
gripper-30	-	-	89	89	-	119	117	89	89
gripper-35	-	-	105	105	-	139	137	105	105
gripper-40	-	-	119	119	-	159	157	119	119
gripper-45	-	-	135	135	-	179	177	135	135
gripper-50	-	-	149	149	-	199	197	149	149
gripper-55	-	-	165	165	-	219	217	165	165
gripper-60	-	-	179	179	-	239	237	179	179

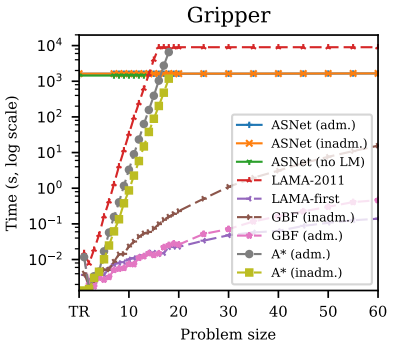

Table 5.4: Plan length and running times for several deterministic planning baselines on Gripper. A "-" indicates the planner was not able to finish within the allowed time. The flat lines for LAMA-2011 indicate that it was killed by a planner timeout on larger instances, before it could finish all scheduled iterations of A* search. The best solution found before timeout is shown in the table at left.

Proposition layers	Path length 1	2	3	4	5
1	30/30	14/30	14/30	14/30	14/30
2	30/30	30/30	14/30	14/30	14/30
3	30/30	30/30	30/30	14/30	14/30
4	30/30	30/30	30/30	30/30	14/30

Table 5.5: Coverage (out of 30) for Monster problem with different ASNet depths.

5.2.3 Monster

The results for our evaluation on the Monster domain are presented in Table 5.5. As expected, the ASNet was only able to solve the problem when it had sufficiently many layers that the $\text{has-monster}(?\mathit{location})$ propositions for the final two locations along each path were in the receptive field for the first two movement actions. Otherwise, the agent chose a path at random, and ended up on the same path as the wumpus around 50% of the time. This underscores two points which we made earlier: first, that the limited receptive field of an ASNet can be a significant handicap in practice. Secondly, that our LM-cut-based heuristic features are not sufficient on their own to correct this deficiency. In Section 6.2.3, we suggest some possible approaches for addressing this shortcoming.

Chapter 6

Conclusion

In Chapter 1, we posed the following research question:

How can we use deep learning to accelerate probabilistic planning?

This chapter will summarise our efforts toward answering this question by recapitulating our main technical contributions. We will also suggest some promising directions for future research in speedup learning, and close by underscoring what this work contributes to the field as a whole.

6.1 Summary

The focus of this thesis has been on obtaining generalised policies for probabilistic planning problems using deep learning. A generalised policy is a mapping from states of a planning problem to actions, where the same mapping can be used for states from any problem in a given planning domain. It is possible to accelerate probabilistic planning by training such a policy using a set of small problems from a domain, then re-using that generalised policy to solve problems too large for a non-learning planner to handle. This thesis makes three novel contributions to the area, which we will now summarise.

Our first contribution was the Action Schema Network (ASNet), a neural network architecture which is specialised to the structure of planning problems. The ASNet is composed of alternating layers of action modules and proposition modules. An action module in one layer takes as input a series of hidden representations for related proposition modules in the previous layer, and produces another hidden representation as output, which can then be fed into proposition modules in the next layer. Proposition modules perform the analogous operation for connecting pairs of action layers together. These modules allow an ASNet to take information about the propositions which are true in a state as input, perform a series of relationally local operations to obtain a rich vector-space representation of the state, then produce an action-selection probability for each action as output. This strategy is comparable to the way in which convnets process images, and is closely related to the other convolution-like architectures covered in Section 2.3. As with traditional convnets, though, an ASNet's expressiveness is limited by its effective receptive field. To address this, we suggested that supplying the network with heuristic input features. Adding flags indicating whether each action belongs to an LM-cut landmark was sufficient to obtain a good policy for several challenging problems.

Our second contribution was a weight-sharing scheme for ASNets. Specifically, we proposed that in each action layer, action modules corresponding to actions instantiated from the same schema could use the same weights. Likewise, in each proposition layer,

proposition modules corresponding to propositions instantiated from the same predicate could also share weights. This scheme makes it possible for the same set of learnt weights to be applied to an ASNet for any problem from a given PPDDL domain, which in turn allows ASNets to encode generalised policies.

Our third contribution was an algorithm for efficiently training ASNet-based generalised policies. The algorithm operates in alternating phases of exploration and learning. During an exploration phase, the algorithm executes the ASNet in each of a series of training problems to build up a memory of visited states. In each visited state, it also augments its state memory with the envelope of a teacher policy obtained by a heuristic search planner; this ensures that the state memory always includes some states on a goal trajectory. During each learning phase, the ASNet's weights are adjusted to minimise a Q-value-based loss defined over the states accumulated in state memory. This loss penalises the ASNet for choosing poor actions. However, in contrast to past work which used classification-based losses, the penalty for choosing a suboptimal action only increases gradually as the Q-value of the action goes up. This means that the network does not have to copy the teacher exactly to decrease the loss. We noted that this supervised training strategy is far more efficient than the reinforcement learning approach used in the FPG planner, which also attempted to apply neural networks to probabilistic planning.

In Chapter 5, we validated our approach with a series of experiments. Our results on a set of challenging probabilistic problems showed that ASNets could substantially outperform baselines for sufficiently large problems, as expected, but that the cost of training ASNets made them less useful on small problems. In general, ASNets appear to be very effective in domains where similar kinds of traps are encountered repeatedly. Such traps occur in the CosaNostra domain, where the agent must learn to pay each toll booth operator when travelling to a customer, and the Triangle Tire World domain, where it is necessary to take the longest path to the goal in order to keep the agent close to spare tires. Separately, our experiments also showed that ASNets are capable of solving deterministic problems, and empirically validated the receptive field limitation which we had hypothesised earlier.

6.2 Future work

There has been little prior work on utilising deep learning methods for the kinds of planning considered in this thesis, so there remains an array of questions to be investigated and problems to be solved. This section serves to highlight some of the related research directions which we consider most promising.

6.2.1 Going beyond SSPs

So far, we have exclusively considered using ASNets for discrete, probabilistic planning. In this setting, each state of a problem is an assignment of truth values to a set of propositions, and all actions' effects can be described as known probability distributions over a series of deterministic outcomes. As alluded to in Section 1.1, this formalism is reasonably powerful, but it is still only one point on a continuum of approaches with different levels of expressiveness. In many cases, it could be possible to apply a slightly modified variant of the ASNet architecture to more complex tasks. For an ASNet-like approach

to work, we must assume three things about the family of planning problems which we wish to solve:

1. Problems in the family should be formulated in terms of a finite number of actions and state variables, although the variables do not necessarily need finite domains. This property is necessary to give us a meaningful notion of "action" and "proposition" (or "variable") when constructing the modules of an ASNet.

2. Actions and state variables should be instantiated from a fixed set of templates, and those templates should be shared across entire the entire family of problems. This requirement allows us to apply weight-sharing to generalise learnt knowledge.

3. There should be some way of determining how instantiated action templates can affect, or depend on, instantiated variable templates. This gives us a way to wire together the modules of an ASNet.

In probabilistic planning, we can use action schemas as templates for actions, and predicates as templates for propositions, then use the PPDDL definitions of action schemas to decide which actions can influence, or depend on, which variables. The same approach would work for any problems declared in a PDDL-like language: for instance, we could support the numeric state variables of PDDL 2.1 (Fox and Long, 2003) or PPDDL (Younes and Littman, 2004) in the same way that we do propositions. The only difference would be that we would have to give the network continuous input values for the new state variables. We could also use an ASNet-like approach to solve problems with more complex transition models. For instance, factored Markov Decision Processes with Imprecise Probabilities (MDPIPs) (White III and Eldeib, 1994)—which are able to express uncertainty in the state transition model itself—might be amenable to this sort of modelling.

All of the above problems are still fully observable: at each time step, an agent is able to observe every aspect of the state, and the only uncertainty which it must deal with is that which is inherent in the outcomes of actions. It could also be possible to extend ASNets to planning problems which are only partially observable, such as Partially Observable Markov Decision Processes (POMDPs). In a POMDP, the observation which an agent receives at each time step may encompass only part of the current state, and may be partly corrupted by measurement noise (Bertsekas, 1995). For some "easy" POMDPs, it is possible to obtain a good policy which depends only on the current observation, in much the same way that it is possible to obtain an optimal policy for SSPs which depends only on the current state. For instance, many Atari games are only partially observable, but reinforcement learning methods for fully observable problems have nevertheless been able to play Atari well (Mnih et al., 2013). This suggests that it may sometimes be possible to obtain a good policy simply by applying an ASNet to the observable variables of a partially observable planning problem. However, a more sophisticated approach would be required for problems which are not merely disguised SSPs.

Bertsekas (1995) note that the optimal action to take in a POMDP at some time step t depends only on the observations made and actions taken up to that time. Hence, to adapt ASNets to POMDPs, it should suffice to replace the single state s_t which the ASNet currently receives as input with a summary of a sequence $o_1, a_1, o_2, a_2, \ldots, a_{t-1}, o_t$ representing past observations and actions. It may be possible to do this with recurrent action and proposition modules. For instance, in addition to taking the outputs of relevant proposition modules in the previous layer as inputs, an action module could also

receive its *own* output from the previous time step as input. In this way, it could build up a rich representation of the entire trajectory of observations which it has made. As in the fully observable case, there is no guarantee that such an ASNet would be able to learn an *optimal* policy, but it could have a good chance of obtaining a reasonable policy for some non-trivial POMDPs. A similar approach is employed by Jain et al. (2016), who use a mixture of graph convolutions and recurrent neural networks to reason about the spatio-temporal behaviour of people and objects in a scene.

6.2.2 Learning other kinds of knowledge

In this thesis, we have focused on using ASNets to learn mappings from states to actions. However, there are many other kinds of mapping which an ASNet could learn. For instance, by outputting both a Q-value $Q^\theta(s, a)$ and an action selection probability $\pi^\theta(a \mid s)$, an ASNet could produce a generalised heuristic

$$h^\theta(s) = \sum_a \pi^\theta(a \mid s) \cdot Q^\theta(s, a) \,. \tag{6.1}$$

Learnt Q-values could also be used for value-function-based reinforcement learning methods, like Q-learning (Mnih et al., 2013) and actor-critic (Konda and Tsitsiklis, 2000). The main challenge to learning *generalised* value functions is the range of possible outputs: the expected cost $V^*(s_0)$ of an optimal policy for a problem can grow rapidly as the problem becomes larger. However, neural networks are best at learning to produce quantities which fall within a fixed range, and tend to generalise poorly to output ranges which were not observed in the training set.

Vector-space embeddings of states are another kind of knowledge which ASNets could be repurposed to learn. To demonstrate what a vector-space embedding is, and why it could be useful, we can turn to natural language processing. Many problems in natural language processing call for a sequence of words—a sentence or a document, for instance—to be transformed into a feature vector, then classified with a learnt classifier. An obvious choices of input feature space for such a classifier might be a count vector indicating how many instances of each known word appear in a document. However, such a representation cannot merge synonymous terms, discount irrelevant terms, and so on, so there is little guarantee that semantically similar documents will have vector space representations which are close to each other. This problem can be tackled by learning an *embedding* which maps words or documents into a vector space where similar texts are nearby, and dissimilar texts are far away; the skip-gram model of Mikolov et al. (2013) is an excellent example of this technique. Statistical machine learning techniques are often much better at classifying learnt embeddings of documents than at classifying simple count-based embeddings. Hinton (1984) refers to such learnt embeddings as *distributed representations*, as a single element of the thing to be represented (a single word in a document, for instance) can affect many elements of the representation.

Planning suffers from a similar problem to natural language processing. While it's trivially easy to encode a state as a binary vector of propositions' truth values, there is no guarantee that nearby vectors will correspond to similar states. Indeed, flipping just a single bit could change a state from a goal to a dead end. Much like word and phrase embeddings, vector space embeddings produced by an ASNet could serve as stable, semantically meaningful inputs to other learnt classifiers, thus enabling applications beyond those which would be possible with an ASNet alone. The most effective method

of training an ASNet to produce such embeddings is an open question.

6.2.3 Removing heuristic inputs

In Section 3.3, we noted that the limited information propagation between layers of an ASNet means that it is not able to reason about long-distance, recursive relationships. We partially addressed this problem by adding heuristic-derived input features to the network, but this solution is inelegant, expensive, and has its own limitations. Future work could consider an architectural solution to this problem. One possible way of lifting this limitation could be to remove the assumption of fixed depth. Instead of learning a series of weights for successive action and proposition layers, we could learn a single set of action layer weights, and a single set of proposition layer weights. The corresponding transformations could be applied as many times as necessary to build a rich representation for a problem, even if the number of applications was different for different problems. This approach would be similar to the way in which recurrent neural networks process time series of varying length. A similar technique has also been explored in the context of computer vision by Belagiannis and Zisserman (2017), who proposed repeatedly applying a small cascade of 2D convolutions to accurately localise human joints within an image. Alternatively, it may be worth going beyond graph convolutions, and instead considering some of the alternative architectures for structured deep learning which we covered Section 2.3.4.

6.2.4 Theoretical limits of reactive neural network policies

While we have talked at length about the practical limitations of ASNets, we have not devoted much attention to the general ability of neural networks to solve, or help to solve, planning problems. This topic is particularly relevant in light of the recent interest in using neural networks for reinforcement learning and other, similar tasks—what can these networks represent, and what can they efficiently learn? These questions are muddied by the lack of a clear definition of what a neural network is. For instance, Siegelmann and Sontag (1991) show that a family of simple recurrent neural network architectures can represent any computable function. However, they prove this by assuming that the internal activations of a neural network can take arbitrary rational values. This allows them to simulate a three-stack pushdown automaton by storing the contents of each stack in a single rational. Even if one could replace the fixed-size floating point activations of contemporary neural networks with arbitrary-precision rationals, the sort of network suggested by Siegelmann and Sontag is unlikely to be uncovered by stochastic gradient descent! A proper characterisation of the limits of neural-network-based reactive policies would require a definition of a "neural network" which is expansive enough to include common architectures such as MLPs and convnets, but still restrictive enough to have meaningful computational limitations.

6.2.5 Fully integrating training into a planner

In this thesis, we have primarily focused on neural network architectures for planning, and not as much on algorithms for training and exploiting neural networks. While the training algorithm in Section 4.1 has been adequate for demonstrating the utility of our proposed architecture, it has a number of shortcomings. One shortcoming is that its "teacher" policy is computed using a traditional non-learning planner, without making

use of an ASNet. Ideally, the training system should be able to make use of a partially-converged ASNet in order to speed up the acquisition of teacher policies. This would in turn allow the training set to be expanded to include larger problems which are more representative of those used for testing, even if those problems could not be quickly solved with a non-learning planner. There are at least four desiderata which we would like such an algorithm to have:

1. **Effective use of flawed policies:** de la Rosa et al. (2011) observe that in practice, slightly inaccurate generalised policies can substantially harm the performance of learning-based planners, and can potentially reduce their performance far below the level of non-learning planners. Hence, an integrated system for planning and learning needs to be robust to such policies. The obvious solution to this problem would be to first try exploiting the learnt policy directly, then fall back on heuristic search if the learnt policy fails to achieve a high success rate. However, this approach would be wasteful in cases where the policy is only flawed in some states. Ideally, a learning-based planner would be able to balance exploitation of learnt policies against use of standard heuristic search.

2. **Effective use of perfect policies:** In Chapter 4, we saw that ASNets can often learn *perfect* policies for a problem. Ideally, we'd like to avoid doing any search at all when we have such a policy. While this quality may appear simple to obtain, we feel it is worth highlighting because it is *not* satisfied by some existing methods for exploiting learnt policies. For instance, even with a perfect policy for a deterministic problem, the H-Context Breadth-First Search of de la Rosa et al. (2011) will still have to explore a number of states which is worst-case exponential in the length of the shortest plan.

3. **Batched policy evaluation:** As we noted in Section 1.2, deep learning owes a significant amount of its recent success to the availability of hardware which is well-suited to neural networks, such as Graphics Processing Units (GPUs). However, the speed advantage of GPUs evaporates in applications where it is necessary to process only a single input at a time, instead of evaluating entire batches of inputs at once (Vanhoucke et al., 2011). This is why we evaluated on a CPU in Chapter 5: a GPU simply didn't provide a significant speedup for our neural network, which is very small by deep learning standards. To make use of deep learning techniques, an ideal learning-based planner would be able to take advantage of the high data parallelism of GPUs by batching up evaluations of the learnt policy.

4. **Asynchronous exploration and learning:** The training algorithm which we proposed in Section 4.1 is highly synchronous. Each exploration phase must perform a fixed number of trials and optimal policy evaluations on each training problem before moving on to the next learning phase. This means that a disproportionate fraction of the wall time used for exploration is spent on larger problems, since it is much more expensive to evaluate an optimal policy for them. The planners used for teaching are all sequential, so in a distributed or multicore environment, this kind of synchronous exploration would waste computational resources that could instead be spent doing more exploration of smaller problems. Further, the fact that optimisation is suspended during exploration also means that the network cannot improve while larger training problems are being explored. In the best case, a learning-based planner would be able to continuously explore all training problems

in parallel, while optimising the policy at the same time. This would be similar to the way that Asynchronous Advantage Actor Critic (A3C) makes use of distributed hardware for reinforcement learning (Mnih et al., 2016).

Some of the above desiderata are not just relevant to speedup learning, but also touch on exciting open problems in other subfields. For instance, with the exception of FPG (Buffet and Aberdeen, 2009), there are virtually no probabilistic planners capable of parallel planning, so a planner which could exploit distributed (or even multi-core) hardware would be a significant achievement on its own.

6.3 Closing remarks

Although planning and learning are both important aspects of artificial intelligence, the corresponding research fields largely operate independently of one another, with limited cross-pollination of ideas. This is especially true of probabilistic planning and deep learning. In this thesis, we presented one possible way of integrating deep learning into probabilistic planning. However, as the suggested research directions in the previous section demonstrate, there is still a plenty of research to be done to better incorporate the ideas of deep learning into planning. There is no doubt a good number of opportunities for transfer of ideas in the opposite direction, too. Our hope is that the work presented in this thesis will help spur further research into the integration of these two fields by highlighting the complementarities which exist between them.

Appendix A

PPDDL Domains for Experiments

A.1 Probabilistic Blocks World

```
(define (domain prob_bw)
  (:requirements :probabilistic-effects :conditional-effects :equality
                 :typing)

  (:types block)

  (:predicates (holding ?b - block) (emptyhand) (on-table ?b - block)
               (on ?b1 ?b2 - block) (clear ?b - block))

  (:action pick-up
    :parameters (?b1 ?b2 - block)
    :precondition (and (emptyhand) (clear ?b1) (on ?b1 ?b2))
    :effect
      (probabilistic
        3/4 (and (holding ?b1) (clear ?b2) (not (emptyhand))
                 (not (on ?b1 ?b2)))
        1/4 (and (clear ?b2) (on-table ?b1) (not (on ?b1 ?b2)))))

  (:action pick-up-from-table
    :parameters (?b - block)
    :precondition (and (emptyhand) (clear ?b) (on-table ?b))
    :effect
      (probabilistic 3/4 (and (holding ?b) (not (emptyhand))
                              (not (on-table ?b)))))

  (:action put-on-block
    :parameters (?b1 ?b2 - block)
    :precondition (and (holding ?b1) (clear ?b1) (clear ?b2) (not (= ?b1 ?b2)))
    :effect (probabilistic 3/4 (and (on ?b1 ?b2) (emptyhand) (clear ?b1)
                                    (not (holding ?b1)) (not (clear ?b2)))
                           1/4 (and (on-table ?b1) (emptyhand) (clear ?b1)
                                    (not (holding ?b1)))))

  (:action put-down
    :parameters (?b - block)
    :precondition (and (holding ?b) (clear ?b))
    :effect (and (on-table ?b) (emptyhand) (clear ?b) (not (holding ?b))))

  ;; Tower actions from IPC version disabled for comparison with deterministic
  ;; blocksworld.
  ;;
  ;; (:action pick-tower
  ;;   :parameters (?b1 ?b2 ?b3 - block)
  ;;   :precondition (and (emptyhand) (clear ?b1) (on ?b1 ?b2) (on ?b2 ?b3))
  ;;   :effect
  ;;     (probabilistic 1/10 (and (holding ?b2) (clear ?b3) (not (emptyhand))
  ;;                              (not (on ?b2 ?b3)))))
  ;;
```

```
  ;; (:action put-tower-on-block
  ;;   :parameters (?b1 ?b2 ?b3 - block)
  ;;   :precondition (and (holding ?b2) (on ?b1 ?b2) (clear ?b3)
  ;;                      (not (= ?b1 ?b3)))
  ;;   :effect (probabilistic 1/10 (and (on ?b2 ?b3) (emptyhand)
  ;;                                    (not (holding ?b2)) (not (clear ?b3)))
  ;;                          9/10 (and (on-table ?b2) (emptyhand)
  ;;                                    (not (holding ?b2)))))
  ;;
  ;; (:action put-tower-down
  ;;   :parameters (?b1 ?b2 - block)
  ;;   :precondition (and (holding ?b2) (on ?b1 ?b2))
  ;;   :effect (and (on-table ?b2) (emptyhand) (not (holding ?b2))))
)
```

A.2 Triangle Tire World

```
(define (domain triangle-tire)
  (:requirements :typing :strips :equality :probabilistic-effects)
  (:types location)
  (:predicates (vehicle-at ?loc - location)
               (spare-in ?loc - location)
               (road ?from - location ?to - location)
               (not-flattire))
  (:action move-car
    :parameters (?from - location ?to - location)
    :precondition (and (vehicle-at ?from) (road ?from ?to) (not-flattire))
    :effect (and (vehicle-at ?to) (not (vehicle-at ?from))
                 (probabilistic 0.5 (not (not-flattire)))))
  (:action changetire
    :parameters (?loc - location)
    :precondition (and (spare-in ?loc) (vehicle-at ?loc))
    :effect (and (not (spare-in ?loc)) (not-flattire))))
```

A.3 CosaNostra Pizza

```
(define (domain cosanostra)
  (:requirements :typing :strips :probabilistic-effects :conditional-effects
                 :negative-preconditions)

  (:types location - object
          toll-booth open-intersection - location)

  (:predicates (have-pizza) (tires-intact)
               (deliverator-at ?l - location) (pizza-at ?l - location)
               (open ?booth - toll-booth) (operator-angry ?booth - toll-booth)
               (road ?from ?to - location))

  (:action load-pizza
           :parameters (?loc - location)
           :precondition (and (deliverator-at ?loc) (pizza-at ?loc))
           :effect (and (not (pizza-at ?loc)) (have-pizza)))

  (:action unload-pizza
           :parameters (?loc - location)
           :precondition (and (deliverator-at ?loc) (have-pizza))
           :effect (and (pizza-at ?loc) (not (have-pizza))))

  (:action pay-operator
           :parameters (?booth - toll-booth)
```

```
            :precondition (and (deliverator-at ?booth))
            :effect (and (open ?booth)))

  (:action leave-toll-booth
            :parameters (?from - toll-booth ?to - location)
            :precondition (and (deliverator-at ?from) (tires-intact)
                               (road ?from ?to))
            :effect (and
                     ;; angry operators might crush your car (even if you pay
                     ;; them---they're really quite spiteful)
                     (when (and (operator-angry ?from))
                           (and (probabilistic
                                  ;; 50% chance they react in time to drop the
                                  ;; boom
                                  0.5 (and (not (tires-intact)))
                                  ;; 50% chance you get through
                                  0.5 (and (not (deliverator-at ?from))
                                           (deliverator-at ?to)))))
                     ;; happy operators don't do that, though
                     (when (and (not (operator-angry ?from)))
                           (and (not (deliverator-at ?from))
                                (deliverator-at ?to)))
                     ;; leaving without paying makes the operator mad at you when
                     ;; you come back
                     (when (and (not (open ?from)))
                           (and (operator-angry ?from)))))

  (:action leave-open-intersection
            ;; no obstacles at these intersections
            :parameters (?from - open-intersection ?to - location)
            :precondition (and (deliverator-at ?from) (tires-intact)
                               (road ?from ?to))
            :effect (and (not (deliverator-at ?from)) (deliverator-at ?to))))
```

A.4 Gripper (deterministic)

```
(define (domain gripper-typed)
 (:types room gripper ball)

 (:predicates
  (at-robby ?r)
  (at ?b ?r)
  (free ?g)
  (carry ?o ?g))

 (:action move
  :parameters  (?from ?to - room)
  :precondition (and  (at-robby ?from))
  :effect (and  (at-robby ?to)
      (not (at-robby ?from))))

 (:action pick
  :parameters (?obj - ball ?room  - room ?gripper - gripper)
  :precondition  (and (at ?obj ?room) (at-robby ?room) (free ?gripper))
  :effect (and (carry ?obj ?gripper)
      (not (at ?obj ?room))
      (not (free ?gripper))))

 (:action drop
  :parameters  (?obj - ball ?room - room ?gripper - gripper)
  :precondition  (and (carry ?obj ?gripper) (at-robby ?room))
  :effect (and (at ?obj ?room)
      (free ?gripper)
```

```
      (not (carry ?obj ?gripper)))))
```

A.5 Monster

```
;; Problems for this domain have start location, a goal location, and two paths
;; of n transitions leading between them (both unidirectional). One path has a
;; monster at the end of it with a 99% chance of killing the agent, while the
;; other does not. You need to be able to see the monster---for this
;; probabilistic version of the domain, LM-cut landmarks produced from a
;; determinisation will not help, since the determinisation assumes the agent
;; can simply choose not to get killed by the monster if the agent runs into it.
(define (domain monster)
  (:requirements :typing :strips :probabilistic-effects :conditional-effects
                 :negative-preconditions)

  (:types location - object)

  (:constants start finish left-end right-end - location)

  (:predicates (robot-at ?l - location) (has-monster ?l - location)
               (conn ?from ?to - location) (initialised))

  ;; This action spawns a monster at a random location. The ASNet is able to
  ;; observe where the monster spawns (recall all states are fully observable),
  ;; but cannot always make use of that information due to its limited receptive
  ;; field.
  (:action init-monster
           :parameters ()
           :precondition (and (not (initialised)))
           :effect (and (initialised)
                        (probabilistic
                         1/2 (and (has-monster left-end))
                         1/2 (and (has-monster right-end)))))

  (:action drive
           :parameters (?from ?to - location)
           :precondition (and (conn ?from ?to) (robot-at ?from) (initialised))
           :effect (and
             (not (robot-at ?from))
             (when (and (has-monster ?from))
                   ;; only a 1% chance of getting to your destination
                   ;; otherwise, reach a dead end
                   (probabilistic 0.01 (robot-at ?to)))
             (when (and (not (has-monster ?from)))
                   ;; get there with certainty
                   (robot-at ?to)))))
```

List of Figures

List of Tables

Bibliography

M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. S. Corrado, A. Davis, J. Dean, M. Devin, et al. Tensorflow: Large-scale machine learning on heterogeneous distributed systems. *arXiv:1603.04467*, 2016.

R. Alford, H. Borck, J. Karneeb, and D. W. Aha. Active behavior recognition in beyond visual range air combat. Technical report, Naval Research Lab Washington DC, 2015.

A. G. Barto, S. J. Bradtke, and S. P. Singh. Learning to act using real-time dynamic programming. *AIJ*, 1995.

J. Baxter and P. L. Bartlett. Infinite-horizon policy-gradient estimation. *JAIR*, 2001.

V. Belagiannis and A. Zisserman. Recurrent human pose estimation. In *FG*, 2017.

D. Bertsekas and J. N. Tsitsiklis. *Neuro-Dynamic Programming*. Athena Scientific, 1996.

D. P. Bertsekas. *Dynamic programming and optimal control*, volume 1. Athena Scientific Belmont, MA, 1995.

C. M. Bishop. *Pattern recognition and machine learning*. Springer, 2006.

H. Blockeel and L. de Raedt. Top-down induction of first-order logical decision trees. *AIJ*, 1998.

A. L. Blum and M. L. Furst. Fast planning through planning graph analysis. *AIJ*, 1997.

B. Bonet and H. Geffner. Labeled RTDP: Improving the convergence of real-time dynamic programming. In *ICAPS*, 2003.

A. Botea, M. Enzenberger, M. Müller, and J. Schaeffer. Macro-FF: Improving AI planning with automatically learned macro-operators. *JAIR*, 2005.

J. Bresina, R. Dearden, N. Meuleau, S. Ramakrishnan, D. Smith, et al. Planning under continuous time and resource uncertainty: A challenge for AI. In *UAI*, 2002.

M. M. Bronstein, J. Bruna, Y. LeCun, A. Szlam, and P. Vandergheynst. Geometric deep learning: going beyond Euclidean data. *Sig. Proc. Mag.*, 2017.

J. Bruna, W. Zaremba, A. Szlam, and Y. LeCun. Spectral networks and locally connected networks on graphs. *arXiv:1312.6203*, 2013.

O. Buffet and D. Aberdeen. FF+FPG: Guiding a policy-gradient planner. In *ICAPS*, 2007.

O. Buffet and D. Aberdeen. The factored policy-gradient planner. *AIJ*, 2009.

O. Buffet and J. Hoffmann. All that glitters is not gold: Using landmarks for reward shaping in FPG. In *ICAPS workshops*, 2010.

T. Bylander. The computational complexity of propositional STRIPS planning. *AIJ*, 1994.

J. Cai, R. Shin, and D. Song. Making neural programming architectures generalize via recursion. *arXiv:1704.06611*, 2017.

D.-A. Clevert, T. Unterthiner, and S. Hochreiter. Fast and accurate deep network learning by exponential linear units (ELUs). *ICLR*, 2016.

A. Coles, A. Coles, A. G. Olaya, S. Jiménez, C. L. López, S. Sanner, and S. Yoon. A survey of the seventh international planning competition. *AI Mag.*, 2012.

T. de la Rosa, S. J. Celorrio, and D. Borrajo. Learning relational decision trees for guiding heuristic planning. In *ICAPS*, 2008.

T. de la Rosa, S. Jiménez, R. Fuentetaja, and D. Borrajo. Scaling up heuristic planning with relational decision trees. *JAIR*, 2011.

M. Defferrard, X. Bresson, and P. Vandergheynst. Convolutional neural networks on graphs with fast localized spectral filtering. In *NIPS*, 2016.

T. G. Dietterich. Ensemble methods in machine learning. *Multiple classifier systems*, 2000.

C. Domshlak and Z. Feldman. To UCT, or not to UCT? (position paper). In *SoCS*, 2013.

D. K. Duvenaud, D. Maclaurin, J. Iparraguirre, R. Bombarell, T. Hirzel, A. Aspuru-Guzik, and R. P. Adams. Convolutional networks on graphs for learning molecular fingerprints. In *NIPS*, 2015.

P. Felzenszwalb, D. McAllester, and D. Ramanan. A discriminatively trained, multiscale, deformable part model. In *CVPR*, 2008.

A. Fern, S. Yoon, and R. Givan. Approximate policy iteration with a policy language bias. In *NIPS*, 2004a.

A. Fern, S. W. Yoon, and R. Givan. Learning domain-specific control knowledge from random walks. In *ICAPS*, 2004b.

R. E. Fikes and N. J. Nilsson. STRIPS: A new approach to the application of theorem proving to problem solving. *AIJ*, 1971.

M. Fox and D. Long. PDDL2.1: An extension to pddl for expressing temporal planning domains. *JAIR*, 2003.

H. Geffner and B. Bonet. A concise introduction to models and methods for automated planning. 2013.

I. Georgievski and M. Aiello. HTN planning: Overview, comparison, and beyond. *AIJ*, 2015.

X. Glorot and Y. Bengio. Understanding the difficulty of training deep feedforward neural networks. In *AISTATS*, 2010.

A. Graves, G. Wayne, and I. Danihelka. Neural Turing machines. *arXiv:1410.5401*, 2014.

A. Graves, G. Wayne, M. Reynolds, T. Harley, I. Danihelka, A. Grabska-Barwińska, S. G. Colmenarejo, E. Grefenstette, T. Ramalho, J. Agapiou, et al. Hybrid computing using a neural network with dynamic external memory. *Nature*, 2016.

C. Gretton. Gradient-based relational reinforcement learning of temporally extended policies. In *ICAPS*, 2007.

C. Gretton and S. Thiébaux. Exploiting first-order regression in inductive policy selection. In *UAI*, 2004.

E. Groshev, A. Tamar, S. Srivastava, and P. Abbeel. Learning generalized reactive policies using deep neural networks. *arXiv:1708.07280*, 2017.

E. A. Hansen and S. Zilberstein. LAO$*$: A heuristic search algorithm that finds solutions with loops. *AIJ*, 2001.

P. Haslum and H. Geffner. Admissible heuristics for optimal planning. In *AIPS*, 2000.

M. Helmert. The Fast Downward planning system. *JAIR*, 2006.

M. Helmert and C. Domshlak. Landmarks, critical paths and abstractions: what's the difference anyway? In *ICAPS*, 2009.

E. Helms, R. D. Schraft, and M. Hagele. rob@work: Robot assistant in industrial environments. In *Intl. Workshop on Robot and Human Interactive Communication*, 2002.

M. Henaff, J. Bruna, and Y. LeCun. Deep convolutional networks on graph-structured data. *arXiv:1506.05163*, 2015.

G. E. Hinton. Distributed representations, 1984.

S. Hochreiter and J. Schmidhuber. Long short-term memory. *Neural computation*, 1997.

J. Hoffmann. FF: The Fast-Forward planning system. *AI Mag.*, 2001.

S. Ioffe and C. Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift. In *ICML*, 2015.

A. Jain, A. R. Zamir, S. Savarese, and A. Saxena. Structural-RNN: Deep learning on spatio-temporal graphs. In *CVPR*, 2016.

S. Jiménez, T. de la Rosa, S. Fernández, F. Fernández, and D. Borrajo. A review of machine learning for automated planning. *Knowl. Eng. Rev.*, 2012.

K. Kansky, T. Silver, D. A. Mély, M. Eldawy, M. Lázaro-Gredilla, X. Lou, N. Dorfman, S. Sidor, S. Phoenix, and D. George. Schema networks: Zero-shot transfer with a generative causal model of intuitive physics. *arXiv:1706.04317*, 2017.

S. Kearnes, K. McCloskey, M. Berndl, V. Pande, and P. Riley. Molecular graph convolutions: moving beyond fingerprints. *Journal of Computer-Aided Molecular Design*, 2016.

T. Keller and P. Eyerich. PROST: Probabilistic planning based on UCT. In *ICAPS*, 2012.

R. Khardon. Learning action strategies for planning domains. *AIJ*, 1999.

D. Kingma and J. Ba. Adam: A method for stochastic optimization. *arXiv:1412.6980*, 2014.

T. N. Kipf and M. Welling. Semi-supervised classification with graph convolutional networks. *arXiv:1609.02907*, 2016.

L. Kocsis and C. Szepesvári. Bandit based Monte-Carlo planning. In *ECML*, 2006.

A. Kolobov, Mausam, and D. S. Weld. LRTDP versus UCT for online probabilistic planning. In *AAAI*, 2012a.

A. Kolobov, D. Weld, et al. A theory of goal-oriented MDPs with dead ends. *arXiv:1210.4875*, 2012b.

V. R. Konda and J. N. Tsitsiklis. Actor-critic algorithms. In *NIPS*, 2000.

M. Krajňanskỳ, J. Hoffmann, O. Buffet, and A. Fern. Learning pruning rules for heuristic search planning. In *ECAI*, 2014.

A. Krizhevsky, I. Sutskever, and G. E. Hinton. Imagenet classification with deep convolutional neural networks. In *NIPS*, 2012.

Y. LeCun, Y. Bengio, et al. Convolutional networks for images, speech, and time series. *The handbook of brain theory and neural networks*, 1995.

Y. LeCun, Y. Bengio, and G. Hinton. Deep learning. *Nature*, 2015.

Y. LeCun et al. Generalization and network design strategies. *Connectionism in perspective*, 1989.

M. Li, T. Zhang, Y. Chen, and A. J. Smola. Efficient mini-batch training for stochastic optimization. In *KDD*, 2014.

M. Lindauer, H. H. Hoos, F. Hutter, and T. Schaub. Autofolio: An automatically configured algorithm selector. *JAIR*, 2015.

I. Little and S. Thiébaux. Probabilistic planning vs. replanning. In *ICAPS workshops*, 2007.

D. Long, H. Kautz, B. Selman, B. Bonet, H. Geffner, J. Koehler, M. Brenner, J. Hoffmann, F. Rittinger, C. R. Anderson, et al. The AIPS-98 planning competition. *AI Mag.*, 2000.

M. Martin and H. Geffner. Learning generalized policies in planning using concept languages. In *KRR*, 2000.

Mausam and A. Kolobov. *Planning with Markov Decision Processes*. Morgan & Claypool, 2012.

D. McDermott, M. Ghallab, A. Howe, C. Knoblock, A. Ram, M. Veloso, D. Weld, and D. Wilkins. PDDL—the Planning Domain Definition Language. Technical report, Yale Center for Computational Vision and Control, 1998.

T. Mikolov, I. Sutskever, K. Chen, G. S. Corrado, and J. Dean. Distributed representations of words and phrases and their compositionality. In *NIPS*, 2013.

A. Milan, S. H. Rezatofighi, R. Garg, A. R. Dick, and I. D. Reid. Data-driven approximations to NP-hard problems. In *AAAI*, 2017.

V. Mnih, K. Kavukcuoglu, D. Silver, A. Graves, I. Antonoglou, D. Wierstra, and M. Riedmiller. Playing Atari with deep reinforcement learning. *arXiv:1312.5602*, 2013.

V. Mnih, A. P. Badia, M. Mirza, A. Graves, T. Lillicrap, T. Harley, D. Silver, and K. Kavukcuoglu. Asynchronous methods for deep reinforcement learning. In *ICML*, 2016.

C. Muise, S. McIlraith, J. A. Baier, and M. Reimer. Exploiting n-gram analysis to predict operator sequences. In *ICAPS*, 2009.

M. A. H. Newton, J. Levine, M. Fox, and D. Long. Learning macro-actions for arbitrary planners and domains. In *ICAPS*, 2007.

M. Niepert, M. Ahmed, and K. Kutzkov. Learning convolutional neural networks for graphs. In *ICML*, 2016.

S. Niu, S. Chen, H. Guo, C. Targonski, M. C. Smith, and J. Kovačević. Generalized value iteration networks: Life beyond lattices. *arXiv:1706.02416*, 2017.

J. L. Obes, C. Sarraute, and G. Richarte. Attack planning in the real world. *arXiv:1306.4044*, 2013.

A. Porco, A. Machado, and B. Bonet. Automatic reductions from PH into STRIPS or how to generate short problems with very long solutions. In *ICAPS*, 2013.

J. R. Quinlan. Induction of decision trees. *Machine learning*, 1986.

S. Reed and N. de Freitas. Neural programmer-interpreters. *arXiv:1511.06279*, 2015.

S. Richter and M. Westphal. The LAMA planner: Guiding cost-based anytime planning with landmarks. *JAIR*, 2010.

S. Richter, M. Westphal, and M. Helmert. Lama 2008 and 2011. In *International Planning Competition*, pages 117–124, 2011.

R. L. Rivest. Learning decision lists. *Machine learning*, 1987.

S. Russell and P. Norvig. *Artificial Intelligence: A Modern Approach*. Prentice-Hall, Egnlewood Cliffs, 1995.

J. Seipp, S. Sievers, and F. Hutter. Fast downward Cedalion. *IPC*, 2014.

D. I. Shuman, S. K. Narang, P. Frossard, A. Ortega, and P. Vandergheynst. The emerging field of signal processing on graphs: Extending high-dimensional data analysis to networks and other irregular domains. *Sig. Proc. Mag.*, 2013.

H. T. Siegelmann and E. D. Sontag. Turing computability with neural nets. *Applied Mathematics Letters*, 1991.

D. Silver, A. Huang, C. J. Maddison, A. Guez, L. Sifre, G. Van Den Driessche, J. Schrittwieser, I. Antonoglou, V. Panneershelvam, M. Lanctot, et al. Mastering the game of Go with deep neural networks and tree search. *Nature*, 2016.

J. Slaney and S. Thiébaux. Blocks world revisited. *AIJ*, 2001.

N. Srivastava, G. E. Hinton, A. Krizhevsky, I. Sutskever, and R. Salakhutdinov. Dropout: a simple way to prevent neural networks from overfitting. *JMLR*, 2014.

M. Steinmetz and J. Hoffmann. State space search nogood learning: Online refinement of critical-path dead-end detectors in planning. *AIJ*, 2017.

M. Steinmetz, J. Hoffmann, and S. I. Campus. Search and learn: On dead-end detectors, the traps they set, and trap learning. In *IJCAI*, 2017.

N. Stephenson. *Snow Crash*. Bantam Books, 1992.

M. Stolle and D. Precup. Learning options in reinforcement learning. In *SARA*, 2002.

R. S. Sutton and A. G. Barto. *Reinforcement learning: An introduction*. MIT Press, 1998.

A. Tamar, Y. Wu, G. Thomas, S. Levine, and P. Abbeel. Value iteration networks. In *NIPS*, 2016.

S. Toyer, F. Trevizan, S. Thiébaux, and L. Xie. Action schema networks: Generalised policies with deep learning. *arXiv:1709.04271*, 2017.

F. Trevizan, S. Thiébaux, and P. Haslum. Occupation measure heuristics for probabilistic planning. In *ICAPS*, 2017.

F. W. Trevizan and M. M. Veloso. Short-sighted stochastic shortest path problems. In *ICAPS*, 2012.

F. W. Trevizan and M. M. Veloso. Finding objects through stochastic shortest path problems. In *AAMAS*, 2013.

M. Vallati, L. Chrpa, M. Grześ, T. L. McCluskey, M. Roberts, S. Sanner, et al. The 2014 International Planning Competition: Progress and trends. *AI Mag.*, 2015.

V. Vanhoucke, A. Senior, and M. Z. Mao. Improving the speed of neural networks on CPUs. In *NIPS workshops*, 2011.

A. Vezhnevets, V. Mnih, S. Osindero, A. Graves, O. Vinyals, J. Agapiou, et al. Strategic attentive writer for learning macro-actions. In *NIPS*, 2016.

O. Vinyals, M. Fortunato, and N. Jaitly. Pointer networks. In *NIPS*, 2015.

J. Virseda, D. Borrajo, and V. Alcázar. LLAMA: Learning LAMA. In *IPC*, 2014.

C. C. White III and H. K. Eldeib. Markov decision processes with imprecise transition probabilities. *Operations Research*, 1994.

Y. Wu, M. Schuster, Z. Chen, Q. V. Le, M. Norouzi, W. Macherey, M. Krikun, Y. Cao, Q. Gao, K. Macherey, et al. Google's neural machine translation system: Bridging the gap between human and machine translation. *arXiv:1609.08144*, 2016.

Y. Xu, A. Fern, and S. W. Yoon. Discriminative learning of beam-search heuristics for planning. In *IJCAI*, 2007.

Y. Xu, A. Fern, and S. Yoon. Learning linear ranking functions for beam search with application to planning. *JMLR*, 2009.

Y. Xu, A. Fern, and S. W. Yoon. Iterative learning of weighted rule sets for greedy search. In *ICAPS*, 2010.

S. Yoon, A. Fern, and R. Givan. Inductive policy selection for first-order MDPs. In *UAI*, 2002.

S. Yoon, A. Fern, and R. Givan. Discrepancy search with reactive policies for planning. In *AAAI workshops*, 2006a.

S. W. Yoon, A. Fern, and R. Givan. Learning heuristic functions from relaxed plans. In *ICAPS*, 2006b.

S. W. Yoon, A. Fern, and R. Givan. Using learned policies in heuristic-search planning. In *IJCAI*, 2007.

H. L. Younes and M. L. Littman. PPDDL1.0: an extension to PDDL for expressing planning domains with probabilistic effects, 2004.

H. L. Younes, M. L. Littman, D. Weissman, and J. Asmuth. The first probabilistic track of the international planning competition. *JAIR*, 2005.

W. Zaremba and I. Sutskever. Reinforcement learning neural Turing machines (revised). *arXiv:1505.00521*, 2015.

M. D. Zeiler and R. Fergus. Stochastic pooling for regularization of deep convolutional neural networks. *arXiv:1301.3557*, 2013.

M. D. Zeiler and R. Fergus. Visualizing and understanding convolutional networks. In *ECCV*, 2014.

G. Zhu. Real-time elective admissions planning for health care providers. Master's thesis, University of Waterloo, 2013.