Learning Generalized Reactive Policies Using Deep Neural Networks

Edward Groshev,[†] Aviv Tamar,[†] Maxwell Goldstein,[‡]
Siddharth Srivastava,[*§] Pieter Abbeel[†]
[†] Department of Computer Science, University of California, Berkeley CA 94720
[‡] Department of Computer Science, Princeton University, Princeton, NJ 08544
[§] School of Computing, Informatics and Decision Systems Engineering, Arizona State University, Tempe, AZ 85281

Abstract

We present a new approach to learning for planning, where knowledge acquired while solving a given set of planning problems is used to plan faster in related, but new problem instances. We show that a deep neural network can be used to learn and represent a *generalized reactive policy* (GRP) that maps a problem instance and a state to an action, and that the learned GRPs efficiently solve large classes of challenging problem instances. In contrast to prior efforts in this direction, our approach significantly reduces the dependence of learning on handcrafted domain knowledge or feature selection. Instead, the GRP is trained from scratch using a set of successful execution traces. We show that our approach can also be used to automatically learn a heuristic function that can be used in directed search algorithms. We evaluate our approach using an extensive suite of experiments on two challenging planning problem domains and show that our approach facilitates learning complex decision making policies and powerful heuristic functions with minimal human input. Video results available at goo.gl/Hpy4e3.

Introduction

In order to help with day to day chores such as organizing a cabinet or arranging a dinner table, robots need to be able plan: to reason about the best course of action that could lead to a given objective. Unfortunately, planning is well known to be a challenging computational problem: plan-existence for deterministic, fully observable environments is PSPACE-complete when expressed using rudimentary propositional representations (Bylander 1994). Such results have inspired multiple approaches for reusing knowledge acquired while planning across multiple problem instances (in the form of triangle tables (Fikes, Hart, and Nilsson 1972), learning control knowledge for planning (Yoon, Fern, and Givan 2008), and constructing generalized plans that solve multiple problem instances (Srivastava, Immerman, and Zilberstein 2011; Hu and De Giacomo 2011) with the goal of faster plan computation on a new problem instance.

In this work, we present an approach that unifies the principles of imitation learning (IL) and generalized planning for learning a *generalized reactive policy* (GRP) that predicts the action to be taken, given an observation of the planning problem instance and the current state. The GRP is represented as a deep neural network (DNN). We use an off-the-shelf planner to plan on a set of training problems, and train the DNN to learn a GRP that imitates and generalizes the behavior generated by the planner. We then evaluate the learned GRP on a set of unseen test problems from the same domain. We show that the learned GRP successfully generalizes to unseen problem instances including those with larger state spaces than were available in the training set. This allows our approach to be used in end-to-end systems that learn representations as well as executable behavior purely from observations of successful executions in similar problems.

We also show that our approach can generate representation-independent heuristic functions for a given domain, to be used in arbitrary directed search algorithms such as A* (Hart, Nilsson, and Raphael 1968). Our approach can be used in this fashion when stronger guarantees of completeness and classical notions of "explainability" are desired. Furthermore, in a process that we call "leapfrogging", such heuristic functions can be used in tandem with directed search algorithms to generate training data for much larger problem instances, which in turn can be used for training more general GRPs. This process can be repeated, leading to GRPs that solve larger and more difficult problem instances with iteration.

While recent work on DNNs has illustrated their utility as function representations in situations where the input data can be expressed in an image-based representation, we show that DNNs can also be effective for learning and representing GRPs in a broader class of problems where the input is expressed using a graph data structure. For the purpose of this paper, we restrict our attention to deterministic, fully observable planning problems. We evaluate our approach on two planning domains that feature different forms of input representations. The first domain is Sokoban (see Figure 1). This domain represents problems where the execution of a plan can be accurately expressed as a sequence of images. This category captures a number of problems of interest in household robotics including setting the dinner table. This problem has been described as the most challenging problem

[*]Part of the work was done while this author was at United Technologies Research Center

in the literature on learning for planning (Fern, Khardon, and Tadepalli 2011).

Our second test domain is the traveling salesperson problem (TSP), which represents a category of problems where execution is *not* efficiently describable through a sequence of images. This problem is challenging for classical planners as valid solutions need to satisfy a plan-wide property (namely a Hamiltonian cycle, which does not revisit any nodes). Our experiments with the TSP show that using graph convolutions (Dai et al. 2017) DNNs can be used effectively as function representations for GRPs in problems where the grounded planning domain is expressed as a graph data structure.

Our experiments reveal that several architectural components are required to learn GRPs in the form of DNNs: (1) A *deep* network. (2) Structuring the network to receive as input pairs of current state and goal observations. This allows us to 'bootstrap' the data, by training with *all pairs* of states in a demonstration trajectory. (3) Predicting plan length as an auxiliary training signal can improve IL performance. In addition, the plan length can be effectively exploited as a heuristic by standard planners.

We believe that these observations are general, and will hold for many domains. For the particular case of Sokoban, using these insights, we were able to demonstrate a 97% success rate in one object domains, and an 87% success rate in two object domains. In Figure 1 we show an example test domain, and a non-trivial solution produced by our learned DNN.

Related Work

The interface of planning and learning (Fern, Khardon, and Tadepalli 2011) has been investigated extensively in the past. The works of Khadron (Khardon 1999), Martin and Geffner (Martín and Geffner 2004), and Yoon et al. (Yoon, Fern, and Givan 2002) learn policies represented as decision lists on the logical problem representation, which needs to be hand specified. On the other hand, the literature on generalized planning (Srivastava, Immerman, and Zilberstein 2011; Hu and De Giacomo 2011) has focused on computing iterative generalized plans that solve broad classes of problem instances, with strong formal guarantees of correctness. While all of these strive to reuse knowledge made available during planning, the selection of a good *representation* for expressing the data as well as the learned functions or generalized plans is handcrafted. Feature sets and domain descriptions in these approaches are specified by experts using formal languages such as PDDL (Fox and Long 2003). Similarly, approaches such as case-based planning (Spalzzi 2001), approaches for extracting macro actions (Fikes, Hart, and Nilsson 1972; Scala, Torasso, and others 2015) and for explanation based plan generalization (Shavlik 1989; Kambhampati and Kedar 1994) rely on curated vocabularies and domain knowledge for representing the appropriate concepts necessary for efficient generalization of observations and the instantiation of learned knowledge. Our approach requires as input only a set of successful plans and their executions—our neural network architecture is able to learn a reactive policy that predicts the best action to execute based on the current state of the environment without any additional representational expressions. The current state is expressed either as an image (Sokoban) or as an instance of the graph data structure (TSP).

Neural networks have previously been used for learning heuristic functions (Ernandes and Gori 2004). Recently, deep convolutional neural networks (DNNs) have been used to automatically extract expressive features from data, leading to state-of-the-art learning results in image classification (Krizhevsky, Sutskever, and Hinton 2012), natural language processing (Sutskever, Vinyals, and Le 2014), and control (Mnih et al. 2015), among other domains. The phenomenal success of DNNs for across various disciplines motivates us to investigate whether DNNs can learn useful representations in the learning for planning setting as well. Indeed, one of the contributions of our work is a general convolutional DNN architecture that is suitable for learning to plan.

Imitation learning has been previously used with DNNs to learn policies for tasks that involve short horizon reasoning such as path following and obstacle avoidance (Pomerleau 1989; Ross, Gordon, and Bagnell 2011; Tamar et al. 2016; Pfeiffer et al. 2016), focused robot skills (Mülling et al. 2013; Nair et al. 2017), and recently block stacking (Duan et al. 2017). From a planning perspective, the Sokoban domain considered here is considerably more challenging than block stacking or navigation between obstacles. In (Tamar et al. 2016), a value iteration planning computation was embedded within the network structure, and demonstrated successful learning on 2D gridworld navigation. Due to the curse of dimensionality, it is not clear how to extend that work to planning domains with much larger state spaces, such as the Sokoban domain considered here. In that work the state space was a 2D grid world with local connectivity, making value iteration tractable. However, for Sokoban, the state must include the position of both the agent and the objects, making it much larger than a 2D grid world. While one can construct such a state space, running value iteration on it would be too slow. Another alternative is to try to embed the Sokoban problem in some 2D grid world and run VI on it. This method performs significantly worse than our proposed method. Concurrently with our work, Weber et al. (Weber et al. 2017) proposed a DNN architecture that combines model based planning with model free components for reinforcement learning, and demonstrated results on the Sokoban domain. In comparison, our IL approach requires significantly less training instances of the planning problem (over 3 orders of magnitude) to achieve similar performance in Sokoban.

The 'one-shot' techniques in (Duan et al. 2017), however, are complimentary to this work. The impressive Alpha-Go-Zero (Silver et al. 2017) program learned a DNN policy for Go using reinforcement learning and self play. Key to its success is the natural curriculum in self play, which allows reinforcement learning to gradually explore more complicated strategies. A similar self-play strategy was essential for Tesauro's earlier Backgammon agent (Tesauro 1995). For the goal-directed planning problems we consider here, it is not clear how to develop such a curriculum strategy, although our leapfrogging idea takes a step in that direction. Extending

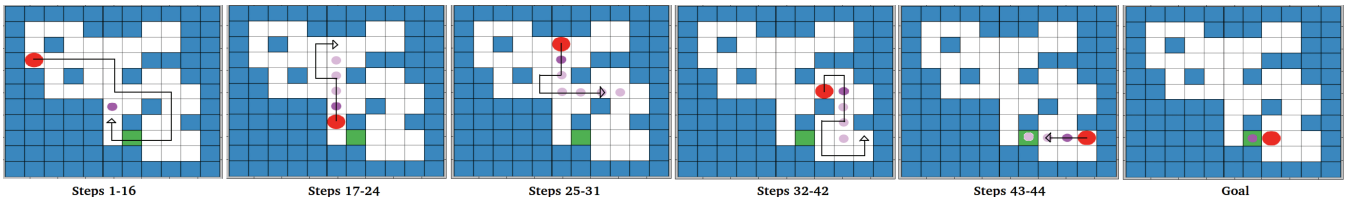


Figure 1: The Sokoban domain (best viewed in color). In Sokoban the agent (red dot) needs to push around movable objects (purple dots) between unmovable obstacles (blue squares) to a goal position (green square). In this figure we show a challenging Sokoban instance with one object. From left to right, we plot several steps in the shortest plan for this task: arrows represent the agent's path, and light purple dots show the resulting object movement. This 44 step trajectory was produced by our learned DNN policy. Note that it demonstrates reasoning about dead ends that may happen many steps after the initial state.

our work to reinforcement learning is a direction for future research.

Our approach thus offers two major advantages over prior efforts: (1) in situations where successful plan executions can be observed, e.g. by observing humans solving problems, our approach significantly reduces the effort required in designing domain representations; (2) in situations where guarantees of success are required, and domain representations are available, our approach provides an avenue for automatically generating a representation-independent heuristic function, which can be used with arbitrary guided search algorithms.

Formal Framework

We assume the reader is familiar with the formalization of deterministic, fully observable planning domains and planning problems in languages such as PDDL (Fox and Long 2003; Helmert 2009) and present the most relevant concepts here. A planning problem domain can be defined as a tuple $K = \langle \mathcal{R}, \mathcal{A} \rangle$, where $\mathcal{R}$ is a set of binary *relations*; and $\mathcal{A}$ is a set of *parameterized actions*. Each action in $\mathcal{A}$ is defined by a set of preconditions categorizing the states on which it can be applied, and the set of instantiated relations that will changed to true or false as a result of executing that action. A planning problem instance associated with a planning domain can be defined as $\Pi = \langle \mathcal{E}, s_0, G \rangle$, where $\mathcal{E}$ is a set of entities, s_0 is an initial state and G is a set of goal conditions. Relations in $\mathcal{R}$ instantiated with entities from $\mathcal{E}$ define the set of *grounded fluents*, $\mathcal{F}$. Similarly, actions in $\mathcal{A}$ instantiated with appropriately entities in $\mathcal{E}$ define the set of *grounded actions*, denoted as $\mathcal{A}[\mathcal{E}]$. The initial state, s_0, for a given planning problem is a complete truth valuation of fluents in $\mathcal{F}$; the goal condition, G, is a truth valuation of a subset of the grounded fluents in $\mathcal{F}$.

As an example, the discrete move action could be represented as follows:

$$\mathit{Move}(\text{loc1}, \text{loc2}) : \begin{cases} \mathit{pre} : \mathit{RobotAt}(\mathit{loc1}), \\ \mathit{eff} : \neg \mathit{RobotAt}(\text{loc1}), \mathit{RobotAt}(\text{loc2}). \end{cases}$$

We introduce several additional notations to the planning problem, to make the connection with imitation learning clearer. Given a planning domain and a planning problem instance, we denote by $S = 2^{\mathcal{F}}$ the state space of the planning problem. A state $s \in S$ corresponds to the values of each fluent in $\mathcal{F}$. The task in planning is to find a sequence of grounded actions, $a_0, \ldots, a_n$ – the so called *plan* – such that $a_n(\ldots(a_0(s_0))\ldots) \models G$.

In Sokoban, the domain represents the legal movement actions and the notion of movement on a bounded grid, a problem instance represents the exact grid layout (denoting which cell-entities are blocked), the starting locations of the objects and the agent, and the goal locations of the objects.

We denote by $o(\Pi, s)$ the *observation* for a problem instance Π when the state is s. For example, o can be an image of the current game state (Figure 1) for Sokoban. We let $\tau = \{s_0, o_0, a_0, s_1, \ldots, s_g, o_g\}$ denote the state-observation-action trajectory implied by the plan. The plan length is the number of states in τ.

Our objective is to learn a generalized behavior representation that efficiently solves multiple problem instances for a domain. More precisely, given a domain K, and a problem instance Π, let $\mathcal{O}_{K,\Pi}$ be the set of possible observations of states from Π. Given a planning problem domain $K = \langle \mathcal{R}, \mathcal{A} \rangle$ we define a *generalized reactive policy (GRP)* as a function mapping observations of problem instances and states to actions: $GRP_K : \cup_\Pi \{\mathcal{O}_{K,\Pi}\} \rightarrow \cup_\Pi \{\mathcal{A}[\mathcal{E}_\Pi]\}$, where $\mathcal{E}_\Pi$ is the set of entities defined by the problem Π and the unions range over all possible problem instances associated with K. Further, GRP_K is constrained so that the observations from every problem instance are mapped to the grounded actions for that problem instance $(\forall \Pi \quad GRP_K(\mathcal{O}_{K,\Pi}) \subseteq \mathcal{A}[\mathcal{E}_\Pi])$. This effectively generalizes the concept of a policy to functions that can map states from multiple problem instances of a domain to action spaces that are legal within those instances.

Imitation Learning In imitation learning (IL), demonstrations of an expert solving a problem are given in the form of observation-action trajectories $D_{\text{imitation}} = \{o_0, a_0, o_1, \ldots, o_T, a_T\}$. The goal is to find a policy – a mapping from observation to actions $a = \mu(o)$, which imitates the expert. A straightforward IL approach is *behavioral cloning* (Pomerleau 1989), in which supervised learning is used to learn μ from the data.

Learning Generalized Reactive Policies

We assume we are given a set D_{train} of N_{train} problem instances $\{\Pi_1, \ldots, \Pi_{N_{\text{train}}}\}$, which will be used for learning a GRP, and a set D_{test} of N_{test} problem instances that will

be used for evaluating the learned model. We also assume that the training and test problem instances are similar in some sense, so that relevant knowledge can be extracted from the training set to improve performance on the test set. Concretely, both training and test instances come from the same distribution.

Our approach consists of two stages: a data generation stage and a policy training stage.

Data generation We generate a random set of problem instances D_{train}. For each $\Pi \in D_{\text{train}}$, we run an off-the-shelf planner to generate a plan and corresponding trajectory τ, and then add the observations and actions in τ to $D_{\text{imitation}}$. In our experiments we used the Fast-Forward (FF) planner (Jörg Hoffman 2001), though any other PDDL planner can be used instead.

Policy training Given the generated data $D_{\text{imitation}}$, we use IL to learn a GRP μ. The learned policy μ maps an observation to action, and therefore can be readily deployed to any test problem in D_{test}.

One may wonder why such a naive approach would even learn to produce the complex decision making ability that is required to solve unseen instances in D_{test}. Indeed, as we show in our experiments, naive behavioral cloning with standard shallow neural networks fails on this task. One of the contributions of this work is the investigation of DNN representations that make this simple approach succeed.

Data Bootstrapping

In the IL literature (e.g., (Pomerleau 1989; Ross, Gordon, and Bagnell 2011)), the policy is typically structured as a mapping from the observation of a state to an action. However, GRPs need to consider the problem instance while generating an action to be executed since different problem instances may have different goals. Although this seems to require more data, we present an approach for "data bootstrapping" that mitigates the data requirements.

Recall that our training data $D_{\text{imitation}}$ consists of N_{train} trajectories composed of observation-action pairs. This means that the number of training samples for a policy mapping state-observations to actions is equal to the number of observation-action pairs in the training data. However, since GRPs use the goal condition in their inputs (captured by a problem instance),*any pair* of observations from successive states $(o(\Pi, s_i), o(\Pi, s_j))$ and the intermediate trajectory in an execution in D_{train} can be used as a sample for training the policy by setting s_j as a goal condition for the intermediate trajectory. Our reasoning for this data bootstrapping technique is based on the following fact:

Proposition 1. *For a planning problem* Π *with initial state* s_0 *and goal state* s_g*, let* $\tau_{opt} = \{s_0, s_1, \ldots, s_g\}$ *denote the shortest plan from* s_0 *to* s_g*. Let* $\mu_{opt}(s)$ *denote an optimal policy for* Π *in the sense that executing it from* s_0 *generates the shortest path* τ_{opt} *to* s_g*. Then,* μ_{opt} *is also optimal for a problem* Π *with the initial and goal states replaced with any two states* $s_i, s_j \in \tau_{opt}$ *such that* $i < j$*.*

Proposition 1 underlies classical planning methods such as triangle tables (Fikes, Hart, and Nilsson 1972). Here, we exploit it to design our DNN to take as input *both* the *current observation* and a *goal observation*. For a given trajectory of length T, the bootstrap can potentially increase the number of training samples from T to $(T-1)^2/2$. In practice, for each trajectory $\tau \in D_{\text{imitation}}$, we uniformly sample $n_{\text{bootstrap}}$ pairs of observations from τ. In each pair, the first observation is treated as the current observation, while the last observation is treated as the goal observation[1]. This results in $n_{\text{bootstrap}} + T$ training samples for each trajectory τ, which are added to a bootstrap training set $D_{\text{bootstrap}}$ to be used instead of $D_{\text{imitation}}$ for training the policy. [2]

Network Structure

We propose a general structure for a convolutional network that can learn a GRP.

Our network is depicted in Figure 2. The current state and goal state observations are passed through several layers of convolution which are shared between the action prediction network and the plan length prediction network. There are also skip connections from the input layer to to every convolution layer.

The shared representation is motivated by the fact that both the actions and the overall plan length are integral parts of a plan. Having knowledge of the actions makes it easy to determine plan length and vice versa, knowledge about the plan length can act as a template for determining the actions. The skip connections are motivated by the fact that several planning algorithms can be seen as applying a repeated computation, based on the planning domain, to a latent variable. For example, greedy search expands the current node based on the possible next states, which are encoded in the domain; value iteration is a repeated modification of the value given the reward and state transitions, which are also encoded in the domain. Since the network receives no other knowledge about the domain, other than what's present in the observation, we hypothesize that feeding the observation to every conv-net layer can facilitate the learning of similar planning computations. We note that in value iteration networks (Tamar et al. 2016), similar skip connections were used in an explicit neural network implementation of value iteration.

For planning in graph domains, we propose to use graph convolutions, similar to the work of (Dai et al. 2017). The graph convolution can be seen as a generalization of an image convolution, where an image is simply a grid graph. Each node in the graph is represented by a feature vector, and linear operations are performed between a node and its neighbors, followed by a nonlinear activation. A detailed description is provided in the supplementary material. For the TSP problem with n nodes, we map a partial Hamiltonian path P of the graph to a feature representation as follows. For each node, the features are represented as a 3-dimensional binary vector.

[1] In our experiments, we used the FF planner, which does not necessarily produce shortest plans. However, Proposition 1 can be extended to satisficing plans.

[2] Note that for the Sokoban domain, goal observations in the test set (i.e., real goals) do not contain the robot position, while the goal observations in the bootstrap training set include the robot position. However, this inconsistency had no effect in practice, which we verified by explicitly removing the robot from the observation.

The first element is 1 if the node has been visited in P, the second element is 1 if it is the current location of the agent, and the third element is 1 if the node is the terminal node. For a Hamiltonian cycle the terminal node is the start node. The state is then represented as a collection of feature vectors, one for each node. In the TSP every Hamiltonian cycle is of length n, so predicting the plan length in this case is trivial, as we encode the number of visited cities in the feature matrix. Therefore, we omit the plan-length prediction part of the network.

Generalization to Different Problem Sizes

A primary challenge in learning for planning is finding representations that can generalize across different problem sizes. For example, we expect that a good policy for Sokoban should work well on the instances it was trained on, 9×9 domains for example, as well as on larger instances, such as 12×12 domains. A convolution-based architecture naturally addresses this challenge.

However, while the convolution layers can be applied to any image/graph size, the number of inputs to the fully connected layer is strictly tied to the problem size. This means that the network architecture described above is fixed to a particular grid dimension. To remove this dependency, we employ a trick used in fully convolutional networks (Long, Shelhamer, and Darrell 2015), and keep only a $k \times k$ window of the last convolution layer, centered around the current agent position. This modification makes our DNN applicable to any grid dimension. Note that since the window is applied *after* the convolution layers, the receptive field can be much larger than $k \times k$. In particular, a value of $k = 1$ worked well in our experiments. For the graph architectures, a similar trick is applied, where the decision at a particular node is a function of the convolution result of its neighbors, and the same convolution weights are used across different graph sizes.

Experiments

Here we report our experiments on learning for planning with DNNs. Our focus is on the following questions:

1. What makes a good DNN architecture for learning a GRP?
2. Can a useful planning heuristic be extracted from the GRP?

The first question aims to show that recent developments in the representation learning community, such as deep convolutional architectures, can be beneficial for planning. The second question has immediate practical value – a good heuristic can decrease planning costs. However, it also investigates a deeper premise. If a useful heuristic can indeed be extracted from the GRP, it means that the GRP has learned some underlying structure in the problem. In the domains we consider, such structure is hard to encode manually, suggesting that the data-driven DNN approach can be promising.

To investigate these questions, we selected two test domains representative of very different classes of planning problems. We used the *Sokoban* domain to represent problems where plan execution can be captured as a set of images, and the goal takes the form of achieving a state property (objects at their target locations). We used the *traveling salesperson problem* as an exemplar for problems where plan execution is not easy to capture as a set of images and the goal features a temporal property.

Sokoban For Sokoban, we consider two difficulty levels: moving a single object as described in Figure 1, and a harder task of moving two objects. We generated training data using a Sokoban random level generator[3].

For imitation learning, we represent the policy with the DNNs as described in Network Structure section and optimize using Adam (Kingma and Ba 2014) (step size 0.001). When training with data bootstrapping, we selected $n_{\text{bootstrap}} = T$ for generating $D_{\text{bootstrap}}$. Unless stated otherwise, the training set used in all Sokoban experiments was comprised of 45k observation-action trajectories from 9k different obstacle configurations.

To evaluate policy performance on the Sokoban domain we use execution success rate. Starting from the initial state, we execute the learned policy deterministically and track whether or not the goal state is reached. We evaluate performance both on test domains of the same size the GRPs were trained on, 9×9 grids, and also on larger problems. We explicitly verified that *none of the test domains appeared in the training set*.

Videos of executions of our learned GRPs for Sokoban are available at goo.gl/Hpy4e3.

TSP For TSP, we consider two different graph distributions. The first is the space of complete graphs with edge weights sampled uniformly in $[0, 1]$. The second, which we term *chord graphs*, is generated by first creating an n-node graph in the form of a cycle, and then adding $2n$ undirected chords between randomly chosen pairs of nodes, with a uniformly sampled weight in $[0, 1]$. The resulting graphs are guaranteed to contain Hamiltonian cycles. However, in contrast to the complete graphs, finding such a Hamiltonian cycle is not trivial. Our results for the chord graphs are similar to the complete graphs, and for space constraints, we present them in the supplementary material. Training data was generated using the TSP solver in Google Optimization Tools[4].

As before, we train the DNN using Adam. We found it sufficient to use only 1k observation-action trajectories for our TSP domain. The metric used is average relative cost[5], defined as the ratio between the cycle cost of the learned policy and the Google solver, averaged over all initial nodes in each test domain. We also compare the DNN policy against a greedy policy which always picks the lowest-cost edge

[3]The Sokoban data-set from the learning for planning competition contains only 60 training domains, which is not enough to train a DNN. Our generator works as follows: we assume the room dimensions are a multiple of 3 and partition the grid into 3x3 blocks. Each block is filled with a randomly selected and randomly rotated pattern from a predefined set of 17 different patterns. To make sure the generated levels are not too easy and not impossible, we discard the ones containing open areas greater than 3x4 and discard the ones with disconnected floor tiles. For more details we refer the reader to Taylor et al. (Taylor and Parberry 2011).

[4]https://developers.google.com/optimization

[5]For the complete graphs, all policies always succeeded in finding a Hamiltonian cycle. For the chord graphs, we report success rates in the supplementary material.

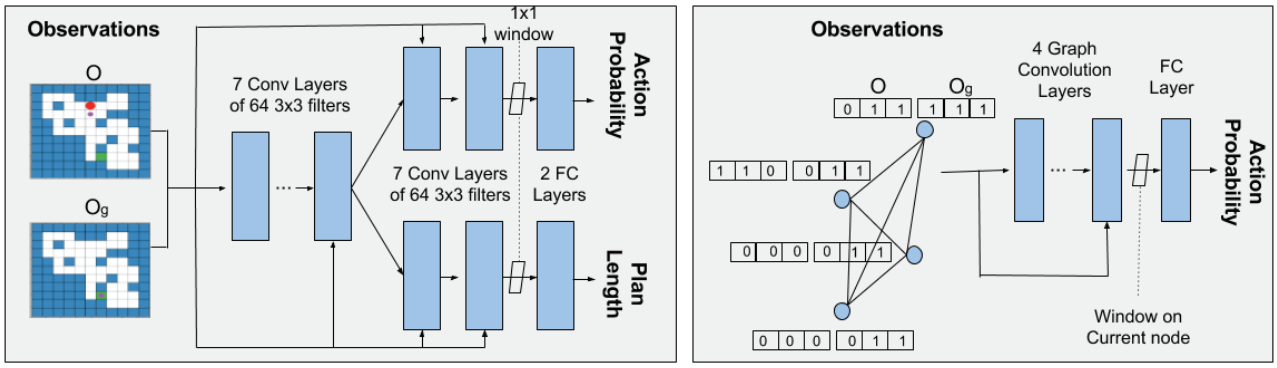


Figure 2: Network architecture. The architecture on the left is used for Sokoban, while the one on the right is used for the TSP. A pair of current and goal observations are passed in to a shared conv-net. This shared representation is input to an action prediction conv-net and a plan length prediction conv-net. Skip connections from the input observations to all conv-layers are added. For the TSP network, we omitted the plan length prediction, as the features directly encode the number of nodes visited, making the prediction trivial. All activation functions are ReLU's and the final one is a SoftMax. In both architectures, after the last convolution layer, we apply a $k \times k$ window around the agents location to ensure a constant size feature vector is passed to the fully connected layers. This effectively decouples the architecture from the problem size and allows the receptive field to be greater than the $k \times k$ window.

leading to an unvisited node.

As in the Sokoban domain, we evaluate performance on test domains with graphs of the same size as the training set, 4 node graphs, and on larger graphs with up-to 11 nodes.

Evaluation of Learned GRPs

Here we evaluate performance of the learned GRPs on previously unseen test problems. Our results suggest that the GRP can learn a well-performing planning-like policy for challenging problems. In the Sokoban domain, on 9×9 grids, the learned GRP in the best performing architecture (14 layers, with bootstrapping and a shared representation) can solve one-object Sokoban with 97% success rate, and two-object Sokoban with 87% success rate. Figure 1 shows a trajectory that the policy predicted in a challenging one-object domain from the test set. Two-object trajectories are difficult to illustrate using images; we provide a video demonstration at goo.gl/Hpy4e3. We observed that the GRP effectively learned to select actions that avoid dead ends far in the future, as Figure 1 demonstrates. The most common failure mode is due to cycles in the policy, and is a consequence of using a deterministic policy. Due to space constraints, further analysis of failure modes is given in the supplementary material. The learned GRP can thus be used to solve new planning problem instances with a high chance of success. In domains where simulators are available, a planner can be used as a fallback if the policy fails in simulation.

Figure **??** shows the performance of the GRP policy on complete graphs of sizes $4 - 11$, when trained on graphs of the same size (respectively). For both the GRP and the greedy policy, the cost increases approximately linearly with the graph size. For the greedy policy, the rate of cost increase is roughly twice the rate for the GRP, showing that the GRP learned to perform some type of lookahead planning.

Investigation of Network Structure

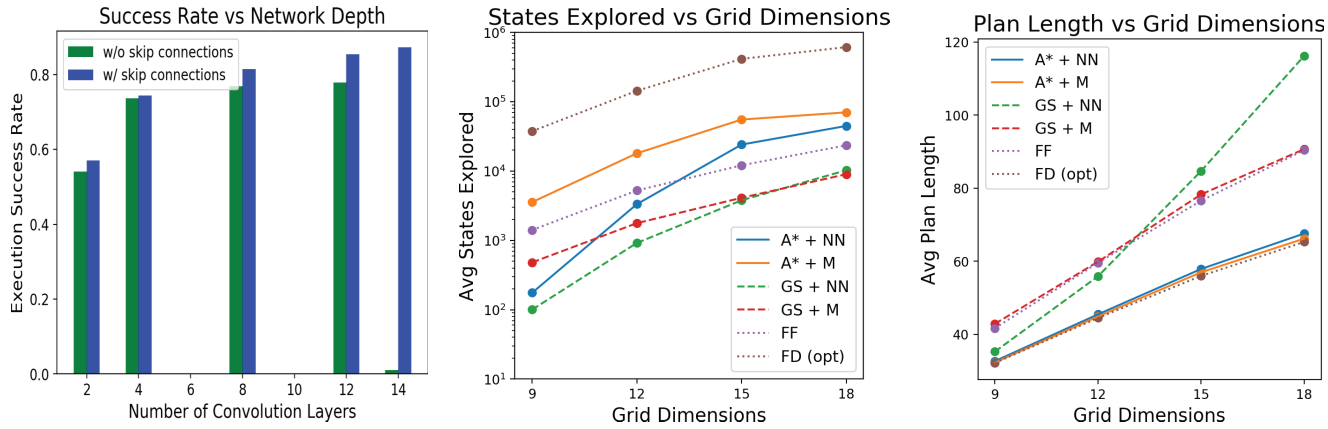
We performed ablation experiments to tease out the important ingredients for a successful GRP. Our results suggest that deeper networks improve performance.

In Figure 3a we plot execution success rate on two-object Sokoban, for different network depths, and with or without skip connections. The results show that deeper networks perform better, with skip connections resulting in a consistent advantage. In the supplementary material we show that a deep network significantly outperformed a shallow network with the same number of parameters, further establishing this claim. The improved results for the deeper networks suggest that for learning GRP's – **the deeper the network the better**. We note a related observation in the context of a DNN representation of the value iteration planning algorithm in (Tamar et al. 2016). However, in our experiments the performance levels off after 14 layers. We attribute this to the general difficulty of training deep DNNs due to gradient propagation, as evident in the failure of training the 14 layer architecture without skip connections, Figure 3a.

We also investigated the benefit of having a shared representation for both action and plan length prediction, compared to predicting each with a separate network. The ablation results are presented in Table 1. Interestingly, the plan length prediction improves the accuracy of the action prediction.

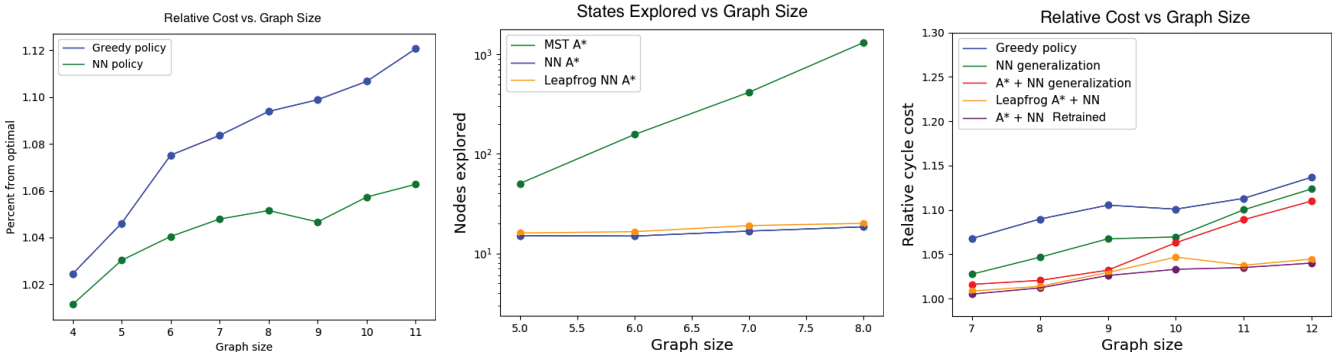
GRP as a Heuristic Generator

We now show that the learned GRPs can be used to extract *representation independent heuristics* for use with arbitrary guided search algorithms. To our knowledge, there are no other approaches for computing such heuristics without using hand-curated domain vocabularies or features for learning and/or expressing them. However, to evaluate the quality of our learned heuristics, we compared them with a few well-known heuristics that are either handcrafted or com-



(a) (b) (c)

Figure 3: Sokoban results. (a) Investigating DNN depth and skip connections. We plot the success rate for deterministic execution in two-object Sokoban. Deeper networks show improved success rates and skip connections improve performance consistently. We were unable to successfully train a 14 layer deep network without skip connections. (b,c) Performance of learned heuristic. The GRP was trained only on 9x9 instances, and evaluated (as a heuristic, see text for more details) on larger instances. (b) shows number of states explored (i.e., planning speed) and (c) shows plan length (i.e., planning quality). A* with the learned heuristic produced nearly optimal plans with an order of magnitude reduction in the number of states explored.



(a) (b) (c)

Figure 4: TSP results. (a) Performance (average relative cost; see text for details) for GRPs trained and tested on problems of sizes $4-11$, respectively. We compare the GRP with a greedy policy. (b,c) Performance of learned heuristic. The GRP was trained on 4-node graphs, and evaluated (as a heuristic, see text for more details) on larger instances. (b) shows number of states explored (i.e., planning speed). We compare with the minimum spanning tree heuristic, which is admissible for TSP. (c) shows average relative cost (i.e., planning quality) compared to plans from the Google solver. Note that up to a graph of size 9, the performance of A* with GRP heuristic (labeled A*+NN generalization) was within 5% of optimal, while requiring orders of magnitude less computation than the MST heuristic. We also present results for the leapfrogging algorithm (see text for details), and additionally compare to a baseline of retraining the GRP with optimal data for each graph size. Note that the leapfrogging results are very close to the results obtained with retraining, although optimal data was only given for the smallest graph size. This shows that the GRP heuristic can be used for generating reliable training data for domains of larger size than trained on.

puted using handcrafted representations. We found that the representation-independent GRP heuristic was competitive, and remains effective on larger problems than the GRP was trained on. For the Sokoban domain, the plan-length prediction can be directly used as a heuristic function. This approach can be used for state-property based goals in problems where execution can be captured using images. For the TSP domain, we used a heuristic that is inversely proportional to the probability of selecting the next node to visit, as the number of steps required to create a complete cycle is not discriminative. Full details are given in the supplementary material.

We investigated using the GRP as a heuristic in greedy search and A* search (Hart, Nilsson, and Raphael 1968). We use two performance measures: the number of states explored during search and the length of the computed plan. The first measure corresponds to planning speed since evaluating less nodes translates to faster planning. The second measure rep-

	w/ bootstrap	w/o bootstrap	
Predict plan length	2.211	2.481	ℓ_1 norm
Predict plan length	**2.205**	2.319	ℓ_1 norm
& actions	**0.844**	0.818	Succ Rate
Predict actions	0.814	0.814	Succ Rate

Table 1: Benefits of bootstrapping and having a shared representation. To evaluate accuracy of the plan length prediction, we measure the average ℓ_1 loss (absolute difference). To evaluate action prediction we measure the success rate on execution. Best performance was obtained with using bootstrapping and the shared representation. For this experiment the training set contained 25k observation-action trajectories.

resents plan quality.

Sokoban We compare performance in Sokoban to the Manhattan heuristic[6] in Figure 3b. In the same figure we evaluate generalization of the learned heuristic to larger, never before seen, instances as well as the performance of two state-of-the-art planners: Fast Forward (FF, (Jörg Hoffman 2001)) and Fast Downward (FD, (Helmert 2006))[7]. The GRP was trained on 9×9 domains, and evaluated on new problem instances of similar size or larger. During training, we chose the window size $k = 1$ to influence learning a problem-instance-size-invariant policy. As seen in Figure 3b the learned GRP heuristic *significantly outperforms the Manhattan heuristic* in both greedy search and A* search, on the 9x9 problems. As the size of the test problems increases, the learned heuristic shines when used in conjunction with A*, consistently expanding fewer nodes than the Manhattan heuristic. Note that even though the GRP heuristic is not guaranteed to be admissible, when used with A*, the plan quality is very close to optimal, while exploring an order of magnitude less nodes than the conventional alternatives.

TSP We trained the GRP on 6-node complete graphs and evaluated the GRP, used either directly as a policy or as a heuristic within A*, on graphs of larger size. Figure 4(b-c) shows generalization performance of the GRP, both in terms of planning speed (number of nodes explored) and in terms of plan quality (average relative cost). We compare both to a greedy policy, and to A* with the minimum spanning tree (MST) heuristic. Note that the GRP heuristic is significantly more efficient than MST, while not losing much in terms of plan quality, especially when compared to the greedy policy.

Leap-Frogging Algorithm

The effective generalization of the GRP heuristic to larger problem sizes motivates a novel algorithmic idea for learning to plan on iteratively increasing problem sizes, which we term *leap-frogging*. The idea is that, we can use a 'general and optimal' planner, such as FD, to generate data for a small domain, of size d. We then train a GRP using this data, and use the resulting GRP heuristic in A* to *quickly* solve planning problems from a larger domain $d' > d$. These solutions can then be used as new data for training another GRP on the domain size d'. Thus, we can iteratively apply this procedure to solve problems of larger and larger sizes, while only requiring the slow 'general' planner to be applied in the smallest domain size.

In Figure 4c we demonstrate this idea in the TSP domain. We used the solver to generate training data for a graph with 4 nodes. We then evaluate the GRP heuristic trained using leapfrogging on larger domains, and compare with a GRP heuristic that was only trained on the 4-node graph. Note that we significantly improve upon the standard GRP heuristic, while using the same initial optimal data obtained from the slow Google solver. We also compare with a GRP heuristic that was re-trained with optimal data for each graph size. Interestingly, this heuristic performed only slightly better than the GRP trained using leap-frogging, showing that the generalization of the GRP heuristic is effective enough to produce reliable new training data.

Conclusion

We presented a new approach in learning for planning, based on imitation learning from execution traces of a planner. We used deep convolutional neural networks for learning a generalized policy, and proposed several network designs that improve learning performance in this setting, and are capable of generalization across problem sizes. In addition, we showed that our networks can be used to extract a heuristic for off-the-shelf planners, which led to significant improvements over standard heuristics that do not leverage learning.

Our results on the challenging Sokoban domain suggest that DNNs have the capability to extract powerful features from observations, and the potential to learn the type of 'visual thinking' that makes some planning problems easy for humans but very hard for automatic planners. The leapfrogging results, suggest a new approach for planning – when facing a large and difficult problem, first solve simpler instances of the problem and learn a DNN heuristic that aids search algorithms in solving larger instances. This heuristic can be used to generate data for training a new DNN heuristic for larger instances, and so on. Our preliminary results suggest this approach to be promising.

There is still much to explore in employing deep networks for planning. While representations for images based on deep conv-nets have become standard, representations for other modalities such as graphs and logical expressions are an active research area (Dai et al. 2017; Kansky et al. 2017). We believe that the results presented here will motivate future research in representation learning for planning.

References

Bylander, T. 1994. The computational complexity of propositional strips planning. *Artificial Intelligence* 69(1-2):165–204.

[6]The Manhattan heuristic is only admissible in one-object Sokoban. We tried Euclidean distance and Hamiltonian distance. However, Manhattan distance had the best trade-off between performance and computation time.

[7]FD uses an anytime algorithm, so we constrained the planning time to be no more than 5 minutes per instance. For the problem instances we evaluated, FD always found the optimal solution.

Dai, H.; Khalil, E. B.; Zhang, Y.; Dilkina, B.; and Song, L. 2017. Learning combinatorial optimization algorithms over graphs. *arXiv preprint arXiv:1704.01665*.

Duan, Y.; Andrychowicz, M.; Stadie, B.; Ho, J.; Schneider, J.; Sutskever, I.; Abbeel, P.; and Zaremba, W. 2017. One-shot imitation learning. *arXiv preprint arXiv:1703.07326*.

Ernandes, M., and Gori, M. 2004. Likely-admissible and sub-symbolic heuristics. In *Proceedings of the 16th European Conference on Artificial Intelligence*, 613–617. IOS Press.

Fern, A.; Khardon, R.; and Tadepalli, P. 2011. The first learning track of the international planning competition. *Machine Learning* 84(1):81–107.

Fikes, R. E.; Hart, P. E.; and Nilsson, N. J. 1972. Learning and executing generalized robot plans. *Artificial Intelligence* 3:251 – 288.

Fox, M., and Long, D. 2003. PDDL2. 1: An extension to PDDL for expressing temporal planning domains. *J. Artif. Intell. Res.(JAIR)* 20:61–124.

Hart, P. E.; Nilsson, N. J.; and Raphael, B. 1968. A formal basis for the heuristic determination of minimum cost paths. *IEEE transactions on Systems Science and Cybernetics* 4(2):100–107.

Helmert, M. 2006. The fast downward planning system. *Journal of Artificial Intelligence (JAIR)* 26:191–246.

Helmert, M. 2009. Concise finite-domain representations for pddl planning tasks. *Artificial Intelligence* 173(5):503 – 535.

Hu, Y., and De Giacomo, G. 2011. Generalized planning: Synthesizing plans that work for multiple environments. In *IJCAI Proceedings-International Joint Conference on Artificial Intelligence*.

Jörg Hoffman. 2001. FF: The fast-forward planning system. *AI Magazine* 22:57–62.

Kambhampati, S., and Kedar, S. 1994. A unified framework for explanation-based generalization of partially ordered and partially instantiated plans. *Artificial Intelligence* 67(1):29–70.

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

Khardon, R. 1999. Learning action strategies for planning domains. *Artificial Intelligence* 113(1):125 – 148.

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

Krizhevsky, A.; Sutskever, I.; and Hinton, G. E. 2012. Imagenet classification with deep convolutional neural networks. In *Advances in neural information processing systems*, 1097–1105.

Long, J.; Shelhamer, E.; and Darrell, T. 2015. Fully convolutional networks for semantic segmentation. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, 3431–3440.

Martín, M., and Geffner, H. 2004. Learning generalized policies from planning examples using concept languages. *Applied Intelligence* 20(1):9–19.

Mnih, V.; Kavukcuoglu, K.; Silver, D.; Rusu, A. A.; Veness, J.; Bellemare, M. G.; Graves, A.; Riedmiller, M.; Fidjeland, A. K.; Ostrovski, G.; et al. 2015. Human-level control through deep reinforcement learning. *Nature* 518(7540):529–533.

Mülling, K.; Kober, J.; Kroemer, O.; and Peters, J. 2013. Learning to select and generalize striking movements in robot table tennis. *The International Journal of Robotics Research* 32(3):263–279.

Nair, A.; Chen, D.; Agrawal, P.; Isola, P.; Abbeel, P.; Malik, J.; and Levine, S. 2017. Combining self-supervised learning and imitation for vision-based rope manipulation. *arXiv preprint arXiv:1703.02018*.

Pfeiffer, M.; Schaeuble, M.; Nieto, J.; Siegwart, R.; and Cadena, C. 2016. From perception to decision: A data-driven approach to end-to-end motion planning for autonomous ground robots. *arXiv preprint arXiv:1609.07910*.

Pomerleau, D. A. 1989. Alvinn: An autonomous land vehicle in a neural network. In *Advances in Neural Information Processing Systems*, 305–313.

Ross, S.; Gordon, G. J.; and Bagnell, D. 2011. A reduction of imitation learning and structured prediction to no-regret online learning. In *AISTATS*.

Scala, E.; Torasso, P.; et al. 2015. Deordering and numeric macro actions for plan repair. In *IJCAI*, 1673–1681.

Shavlik, J. W. 1989. Acquiring recursive concepts with explanation-based learning. In *IJCAI*, 688–693.

Silver, D.; Schrittwieser, J.; Simonyan, K.; Antonoglou, I.; Huang, A.; Guez, A.; Hubert, T.; Baker, L.; Lai, M.; Bolton, A.; et al. 2017. Mastering the game of go without human knowledge. *Nature* 550(7676):354–359.

Spalzzi, L. 2001. A survey on case-based planning. *Artificial Intelligence Review* 16(1):3–36.

Srivastava, S.; Immerman, N.; and Zilberstein, S. 2011. A new representation and associated algorithms for generalized planning. *Artificial Intelligence* 175(2):615–647.

Sutskever, I.; Vinyals, O.; and Le, Q. V. 2014. Sequence to sequence learning with neural networks. In *Advances in neural information processing systems*, 3104–3112.

Tamar, A.; Levine, S.; Abbeel, P.; WU, Y.; and Thomas, G. 2016. Value iteration networks. In *Advances in Neural Information Processing Systems*, 2146–2154.

Taylor, J., and Parberry, I. 2011. Procedural generation of sokoban levels. In *Proceedings of the International North American Conference on Intelligent Games and Simulation*.

Tesauro, G. 1995. Temporal difference learning and td-gammon. *Communications of the ACM* 38(3):58–68.

Weber, T.; Racanière, S.; Reichert, D. P.; Buesing, L.; Guez, A.; Rezende, D. J.; Badia, A. P.; Vinyals, O.; Heess, N.; Li, Y.; et al. 2017. Imagination-augmented agents for deep reinforcement learning. *arXiv preprint arXiv:1707.06203*.

Yoon, S.; Fern, A.; and Givan, R. 2002. Inductive policy selection for first-order MDPs. In *Proceedings of the Eigh-*

teenth conference on Uncertainty in artificial intelligence, 568–576. Morgan Kaufmann Publishers Inc.

Yoon, S.; Fern, A.; and Givan, R. 2008. Learning control knowledge for forward search planning. *Journal of Machine Learning Research* 9(Apr):683–718.

Appendix

Graph Convolution Network

Consider a graph $\mathcal{G} = (V, \mathcal{E})$ with adjacency matrix A where V has N nodes and $\mathcal{E}$ is the weighted edge set with weight matrix W. Suppose that each node $v \in V$ has a corresponding feature $x_v \in \mathbb{R}^m$ and consider a parametric function $f_\theta : \mathbb{R}^{2m} \to \mathbb{R}^m$ parameterized by $\theta \in \mathbb{R}^f$. Let $\mathcal{N}_i : V \to 2^V$ denote a function mapping a vertex to its ith degree neighborhood. The propagation rule is given by the following equation

$$H_v = \sigma\left(\sum_{u \in \mathcal{N}(v)} A_{uv} f_\theta(x_u, x_v)\right) \tag{1}$$

where σ is the ReLU function. Consider a graph $\mathcal{G}$ of size n, with each vertex having feature vector of size C encoded in the feature matrix $X \in \mathbb{R}^{n \times C}$. In the TSP experiments, we use the propagation rule where the ij entry of the next layer is given by

$$H_{ij} = \sigma\left(\sum_{s \in \mathcal{N}(i)} A_{si}[x_s, x_i, W_{si}]^T \Theta_j + b_j\right) \tag{2}$$

Here, W is the weight matrix of $\mathcal{G}$, A is the adjacency matrix, and $\Theta \in \mathbb{R}^{(2C+1) \times C'}$ is the matrix of weights that we learn and $b \in \mathcal{R}^{C'}$ is a learned bias vector. Θ_j is the jth column of Θ.

In the networks we used for the TSP domain, the initial feature vector is of size $C = 6$. We then applied 4 convolution layers of size $C = 26$. We then applied a convolution of size $C = 1$, corresponding to a fully connected layer. Thus, $j = 1$ in H_{ij} for all i in the last convolution layer.

The final layer of the network is a softmax over H_{i1}, and we select the node i with the highest score that is also connected to the current node.

Relation to Image Convolution In the next proposition we show that this graph-based propagation rule can be seen as a generalization of a standard 2-D convolution, when applied to images (grid graphs). Namely, we show that there exists features for a grid graph and parameters Θ for which the above propagation rule reduces to a standard 2-D convolution.

Proposition 2. *When $\mathcal{G}$ is a grid graph, for a particular choice of f_θ the above propagation rule reduces to the traditional convolutional network. In particular, for a filter of size n, choosing f_θ as a polynomial of degree $2(N-1)$ and $\theta \in \mathbb{R}^{N^2}$ works.*

Proof. For each node v, consider its representation as $v = (v_x, v_y)$ where (v_x, v_y) are the grid coordinates of the vertex.

Num Params	Deep-8	Wide-2	Wide-1	
556288	0.068	0.092	0.129	error rate
	0.83	0.62	0.38	succ rate

Table 2: Comparison of deep vs. shallow networks. The deep network has 8 convolution layers with 64 filter per layer. The shallow networks contain 2 and 1 layers respectively with 256 and 512 filters per layer respectively. Clearly, deeper networks outperform shallow networks while containing an equal number of parameters.

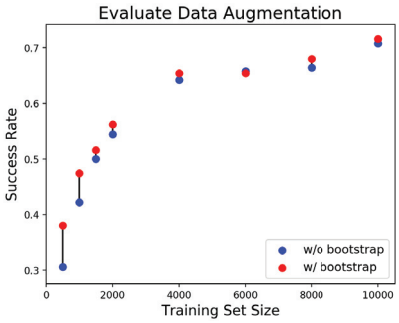


Figure 5: This shows the affect of data bootstrapping on the performance of two-object Sokoban, as a function of the dataset size. Smaller datasets benefit more from data augmentation.

Let $a := \frac{n-1}{2}$. We first transform the coordinates to center them around v by transforming $u \to (u_x - v_x, u_y - v_y)$ so that u lies in the set $[-a, a] \times [-a, a]$.

We wish to design a polynomial g that takes the value $\theta_{i,j}$ at location (i, j). We show that it is possible to do with a degree $2(n-1)$ polynomial by construction. The polynomial g is given by

$$g(x, y) := \sum_{i=-a}^{a} \sum_{j=-a}^{a} \theta_{i,j} \prod_{s=-a, s \neq i}^{a} (s + y) \prod_{t=-a, t \neq j}^{a} (t + x) \tag{3}$$

To see why this is correct, note that for any $(s, t) \in [-a, a] \times [-a, a]$ there is exactly one polynomial inside the summands that does not have either of the terms $(i + u_y)$ or $(j + u_x)$ appearing in its factorization. Indeed, by construction this term is the polynomial corresponding to $\theta_{i,j}$ so that $g(i, j) = C\theta_{i,j}$ for some constant C.

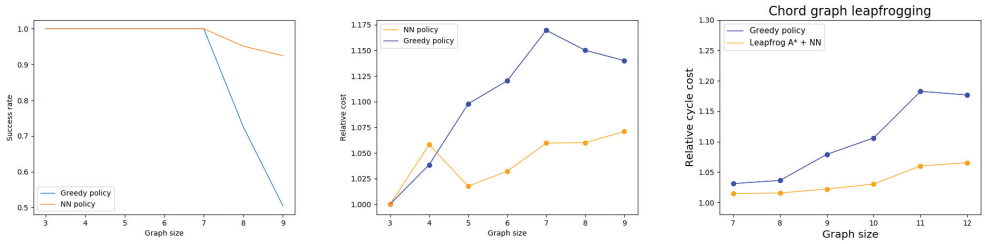
The polynomial inside the summands is of degree $(n-1) + (n-1) = 2(n-1)$, so g is of degree $2(n-1)$. Letting p_u denote th pixel value at node u, setting

$$f_\theta(x_u, x_v) := p_u g(x_u - x_v) \tag{4}$$

completes the proof. □

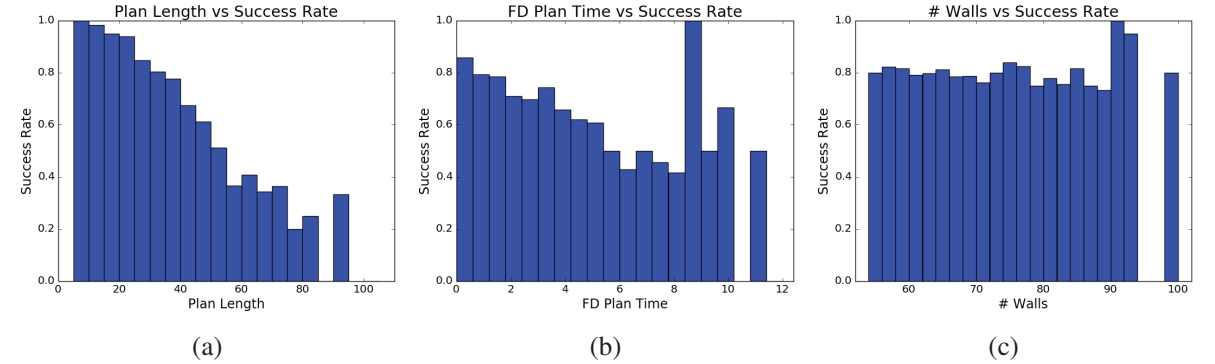
TSP domain heuristic

We can use the graph convolution network as a heuristic inside A-star search. Given a feature encoding of a partial cycle P, we can compute the probability p_i of moving to

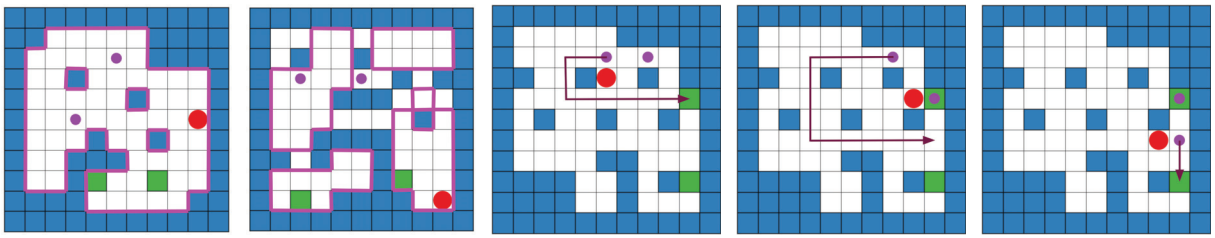


(a) (b) (c)

Figure 6: Chord-graph TSP results. (a) Success rate of neural network policy on chord graphs of size $3-9$, respectively. Note that the agent is only allowed to visit each node once, so the agent may visit a node with no un-visited neighbors which is a dead end. We also show the success rate of the greedy policy. (b) Performance of neural network policy on chord graphs of size 3-9. (c) Leapfrogging algorithm results on chord graphs of size 7-12. We compare to a baseline greedy policy



(a) (b) (c)



(d) (e) (f) (g) (h)

Figure 7: Analysis of Failure Modes. (a-c): Success rate vs features of the domain. Plan length (a) seems to be the main factor in determining success rate. Longer plans fail more often. While there is some relationship between planning time and success rate (b), planning time is not always an accurate indicator, as explained in (d,e). The number of walls (c) does not affect success rate. (d,e): Domains containing large open rooms results in a high branching factor and thus produce the illusion of difficulty while still having a simple underlying policy. The domain in (d) took FD significantly longer time to solve, 8.6 seconds compared to 1.6 seconds for the domain in (e), although it has a shorter optimal solution, 51 steps compared to 65 steps. This is since the domain in (e) can be broken up into small regions which are all connected by hallways, a configuration that reduces the branching factor and thus the overall planning speed. (f-h): Demonstration of the 2nd failure mode in Section . From the start state, the policy moves the first object using the path shown in (f). It proceeds to move the next object using the path in (g). As the game state approaches (h) it becomes clear that the current domain is no longer solvable. The lower object needs to be pushed down but is blocked by the upper object, which can no longer be moved out of the way. In order to solve this level, the first object must ether be moved to the bottom goal or must be moved after the second object has been placed at the bottom goal. Both solutions require a look-ahead consisting of 20+ steps.

any node i. We then use the quantity $(N-v)(1-p_i)/2$ as the heuristic, where N is the total number of nodes and v is the number of visited nodes in the current partial path. Multiplying by $(N-v)/2$ puts the output of the heuristic on the same scale as the current cost of the partial path.

Deep VS Shallow Networks

Here we present another experiment to further establish the claim that the depth of the network improves performance and not necessarily the number of parameters in the network. In Table 2 we compare deep networks against shallow net-

works containing the same number of parameters. Note that we evaluate based on two different metrics. The first metric is classification error on the next action, which shows whether or not the action matches what the planner would have done. The second metrics is execution success rate, as defined above.

Evaluation of Bootstrap Performance

We briefly summarize the evaluation of data bootstrapping in the Sokoban domain. Table 1 shows the success rate and plan length prediction error for architectures with and without the bootstrapping. As can be observed, the bootstrapping resulted in better use of the data, and led to improved results.

While investigating the performance of data bootstrapping with respect to training set size, we observed that a non-uniform sampling performed better on smaller datasets. For each $\tau \in D_{\text{imitation}}$, we sampled an observation $\hat{o}$ from a distribution that is linearly increasing in time, such that observations near the goal have higher probability. The performance of this bootstrapping strategy is shown in Figure 5. As should be expected, performance improvement due to data augmentation is more significant for smaller data sets.

Analysis of Failure Modes

While investigating the failure modes of the learned GRP in the Sokoban domain, we noticed that there were two primary failure modes. The first failure mode is due to cycles in the policy, and is a consequence of using a deterministic policy. For example, when the agent is between two objects a deterministic policy may oscillate, moving back and fourth between the two. We found that a stochastic policy significantly reduces this type of failure. However, stochastic policies have some non-zero probability of choosing actions that lead to a dead end (e.g., pushing the box directly up against a wall), which can lead to different failures. The second failure mode was the inability of our policy to foresee long term dependencies between the two objects. An example of such a case is shown in Figure 7 (f-h), where deciding which object to move first requires a look-ahead of more than 20 steps. A possible explanation for this failure is that such scenarios are not frequent in the training data. This is less a limitation of our approach and more a limitation of the neural network, more specifically the depth of the neural network.

Additionally, we investigated whether the failure cases can be related to specific features in the task. Specifically, we considered the task plan length (computed using FD), the number of walls in the domain, and the planning time with the FD planner (results are similar with other planners). Intuitively, these features are expected to correlate with the difficulty of the task. In Figure 7 (a-c) we plot the success rate vs. the features described above. As expected, success rate decreases with plan length. Interestingly, however, several domains that required a long time for FD were 'easy' for the learned policy, and had a high success rate. Further investigation revealed that these domains had large open areas, which are 'hard' for planners to solve due to a large branching factor, but admit a simple policy. An example of one such domain is shown in Figure 7 (d-e). We also note that the number of walls had no visible effect on success rate – it is the configuration of the walls that matters, and not their quantity.