

---

# Value Propagation Networks

---

**Nantas Nardelli** \*  
University of Oxford  
nantas@robots.ox.ac.uk

**Gabriel Synnaeve**  
Facebook AI Research  
gab@fb.com

**Zeming Lin**  
Facebook AI Research  
zlin@fb.com

**Pushmeet Kohli** †  
Microsoft Research  
pushmeet@google.com

**Philip H. S. Torr**  
University of Oxford  
philip.torr@eng.ox.ac.uk

**Nicolas Usunier**  
Facebook AI Research  
usunier@fb.com

## Abstract

We present Value Propagation (VProp), a parameter-efficient differentiable planning module built on Value Iteration which can successfully be trained using reinforcement learning to solve unseen tasks, has the capability to generalize to larger map sizes, and can learn to navigate in dynamic environments. Furthermore, we show that the module enables learning to plan when the environment also includes stochastic elements, providing a cost-efficient learning system to build low-level size-invariant planners for a variety of interactive navigation problems. We evaluate on static and dynamic configurations of MazeBase grid-worlds, with randomly generated environments of several different sizes, and on a StarCraft navigation scenario, with more complex dynamics, and pixels as input.

## 1 Introduction

Planning is a key component for artificial agents in a variety of domains. However, a limit of classical planning algorithms is that one needs to know how to search for an optimal – or at least reasonable – solution, for each instantiation of every possible type of plan. As the environment dynamics and states complexity increase, this makes writing planners difficult, cumbersome, or simply entirely impractical. This is among the reasons why “learning to plan” has been an active research area to address these shortcomings [Russell et al., 1995, Kaelbling et al., 1996]. To be useful in practice we propose that methods that enable to learn planners should have at least two properties: they should be *traces free*, i.e. not require traces from an optimal planner, and they should *generalize*, i.e. learn planners that are able to function on plans of the same type but of unseen instance and/or planning horizons.

In Reinforcement Learning (RL), learning to plan can be framed as the problem of finding a policy that maximises the expected return from the environment, where such policy is a greedy function that selects actions that will visit states with a higher value for the agent. This in turns shifts the problem to the one of obtaining good estimates of state values. One of the most commonly used algorithms to solve this problem is Value Iteration (VI), which estimates the values by collecting and propagating the seen rewards until a fixed point is reached. A policy – or a plan – can then be constructed by rolling out the obtained value function on the desired state-action pairs.

When the environment can be represented as an occupancy map (a 2D grid), it is possible to approximate this planning algorithm using a deep convolutional neural network (CNN) to propagate the rewards on the grid cells. This enables one to differentiate directly through the planner steps and

---

\*Work was partly done while at Facebook AI Research.

†Now at Google DeepMind.

perform end-to-end learning of the value function. Tamar et al. [2016] train such models – Value Iteration Networks (VIN) – with a supervised loss on the trace from a search/planning algorithm, with the goal to find the parameters that can solve the shortest path task in such environments by iteratively learning the value function using the convnet. However, this baseline requires good target value estimates, violating our wished trace free property and limiting its usage in interactive, dynamics, settings. Furthermore, it doesn’t take advantage of the model structure to generalise to harder instances of the task. That is what we set out to extend and further study.

In this work we extend the formalization used in VIN to more accurately represent the structure of grid-world-like scenarios, enabling Value Iteration modules to be naturally used within the reinforcement learning framework beyond the scope of the initial work, while also removing some of the limitations and underlying assumptions constraining the original architecture. We show that our models can not only learn to plan and navigate in dynamic environments, but that their hierarchical structure provides a way to generalize to navigation tasks where the required planning horizon and the size of the map are much larger than the ones seen at training time.

Our main contributions include: (1) introducing VProp, a network planning module which successfully learns to solve pathfinding tasks via reinforcement learning, (2) demonstrating the ability to generalize to large unseen maps even when training exclusively on much smaller ones, and (3) showing that our modules can learn to plan in environments with more complex dynamics than a static “grid world”, both in terms of transition function and observation complexity.

## 1.1 Related work

Model-based planning with end-to-end architectures has recently shown promising results on a variety of tasks and environments, often using Deep Reinforcement Learning as the algorithmic framework [Silver et al., 2016, Oh et al., 2017, Weber et al., 2017, Groshev et al., 2017, Farquhar et al., 2017]. 3D and 2D navigation tasks have also been tackled within the RL framework [Mirowski et al., 2016], with methods in some cases building and conditioning on 2D occupancy maps to aid the process of localization and feature grounding [Bhatti et al., 2016, Zhang et al., 2017, Banino et al., 2018].

Other work has furthermore explored the usage of VIN-like architectures for navigation problems: Niu et al. [2017] present a generalization of VIN able to learn modules on more generic graph structures by employing a graph convolutional operator to convolve through each node of the graph. Rehder et al. [2017] demonstrate a method for multi-agent planning in a cooperative setting by training multiple VI modules and composing them into one network, while also adding an orientation state channel to simulate non-holonomic constraints often found in mobile robotics. Gupta et al. [2017] and Khan et al. [2017] propose to tackle partially observable settings by constructing hierarchical planners that use VI modules in a multi-scale fashion to generate plans and condition the model’s belief state.

## 2 Background

We consider the control of an agent in a “grid world” environment, in which entities can interact with each other. The entities have some set of attributes, including a uniquely defined type, which describes how they interact with each other, the immediate rewards of such interactions, and how such interactions affect the next state of the world. The goal is to *learn to plan* through reinforcement learning, that is learning a policy trained on various configurations of the environment that can generalize to arbitrary other configurations of the environment, including larger environments, and ones with a larger number of entities. In the case of a standard navigation task, this boils down to learning a policy which, given an observation of the world, will output actions that take the agent to the goal as quickly as possible. An agent may observe such environments as 2D images of size  $d_x \times d_y$ , with  $d_{\text{pix}}$  input panes, which are then potentially passed through an embedding function  $\Phi$  (such as a 2D convnet) to extract the entities and generates some local embedding based on their positions and features.

### 2.1 Reinforcement Learning

The problem of reinforcement learning is typically formulated in terms of computing optimal policies for a Markov Decision Problem (MDP) [Sutton and Barto, 1998]. An MDP is defined by the tuple

$(S, A, T, R, \gamma)$ , where  $S$  is a finite set of states,  $A$  is the set of actions  $a$  that the agent can take,  $T : s \rightarrow a \rightarrow s'$  is a function describing the state-transition matrix,  $R$  is a reward function, and  $\gamma$  is a discount factor. In this setting, an optimal policy  $\pi^*$  is a distribution over the state-action space that maximises in expectation the discounted sum of rewards  $\sum_k \gamma^k r_k$ , where  $r_k$  is the single-step reward. A standard method to find the optimal policy  $\pi : s \rightarrow a$  is to iteratively compute the value function,  $Q^\pi(s, a)$ , updating it based on rewards received from the environment [Watkins and Dayan, 1992]. Using this framework, we can view learning to plan as a *structured prediction* of rewards with the planning algorithm Value Iteration [Bertsekas, 2012] as inference procedure. Beside value-based algorithms, there exist other types which are able to find optimal policies, such as *policy gradient* methods [Sutton et al., 1999], which directly regress to the policy function  $\pi$  instead of approximating the value function. These methods however suffer from high variance estimates in environments that require many steps. Finally, a third type is represented by the actor-critic algorithms family, which combine the policy gradient methods' advantage of being able to compute the policy directly, with the low-variance performance estimation of value-based RL used as a more accurate feedback signal to the policy estimator [Konda and Tsitsiklis, 2000].

## 2.2 Value Iteration Module

Let us denote by  $s$  the current observation of the environment,  $q^0$  the zero tensor of dimensions  $(A, d_x, d_y)$ , and  $h$  an update function acting on the value embeddings. Then, the *Value Iteration* (VI) module [Tamar et al., 2016] is defined for  $k \geq 1$ :

$$\begin{aligned} \forall(i, j) \in \llbracket d_x \rrbracket \times \llbracket d_y \rrbracket, \quad v_{ij}^k &= \max_{a=1..A} q_{a,i,j}^k, \\ q^k &= h(\Phi(s), v^{k-1}). \end{aligned} \quad (1)$$

Given the agent's position  $(x_0, y_0)$  and the current observation of the environment  $s$ , the control policy  $\pi$  is then defined by  $\pi(s, (x_0, y_0)) = \operatorname{argmax}_{a=1..A} q_{a,x_0,y_0}^K$ .

Since the state space can be identified with the coordinates in a 2-dimensional environment of size  $d_x \times d_y$  then a discounted MDP on that state is defined by the discount factor  $\gamma$  together with:

- a transition probability function  $P = d_x \times d_y \times A \times d_x \times d_y$ , such that the matrix obtained fixing the last three coordinates sum to 1.
- an immediate reward function  $R = A \times d_x \times d_y$ , where  $R_{a,i,j}$  represents the reward obtained by performing action  $a$  in state  $i, j$ .

Given a starting  $Q$ -function  $Q^0$  (with same size as  $R$ ), we then can define a sequence  $V^k, Q^k$  of respectively state-value and state-action functions through:

$$\begin{aligned} \forall(i, j) \in \llbracket d_x \rrbracket \times \llbracket d_y \rrbracket, \quad V_{ij}^k &= \max_{a=1..A} Q_{a,i,j}^k, \\ \forall(a, i, j) \in \llbracket A \rrbracket \times \llbracket d_x \rrbracket \times \llbracket d_y \rrbracket, \quad Q_{a,i,j}^k &= R_{a,i,j} + \gamma \langle P_{::a,i,j}, V^{k-1} \rangle. \end{aligned}$$

It follows that when  $\Phi(s) = R$ ,  $h$  is a linear layer of both of its inputs with  $P$  as parameters for the  $v^k$  input:

$$h_{a,i,j}(\Phi(s), v) = \Phi_{a,i,j}(s) + \gamma \langle P_{::a,i,j}, v \rangle. \quad (2)$$

Thus, a linear VI module with  $K$  iteration has the capacity to represent the application of  $K$  iterations of value iteration in an MDP where the state space is the set of 2D coordinates, requiring in the worst case a number of steps equal to the number of states  $- d_x d_y$  in grid worlds  $-$  to evaluate an entire plan. In practice however, far fewer iterations are needed in the considered tasks: for instance path finding in a grid-world requires planning for only the length of the shortest path, which is much smaller unless the configuration corresponds to a very complicated (and unusual) maze.

We can also see that (2) corresponds to only a very special case of linear VI modules. For instance, if the recurrence uses a fully connected layer with weights  $W \in \mathbb{R}^{A \times d_x \times d_y \times (A+1) \times d_x \times d_y}$  and biases  $b \in \mathbb{R}^{A \times d_x \times d_y}$  so that

$$h_{a,i,j}(\Phi(s), v) = \langle W_{a,i,j}, [\Phi(s); v] \rangle + b_{a,i,j},$$

then equation (2) not only corresponds to a special case with extremely sparse weights, but also exhibits a specific structure where the dot product is non-trivial only on the recurrent part. This relationship ultimately forms the motivation for the development of VI modules [Tamar et al., 2016, Section 3.1].

### 3 Models

Let us reformulate the Value Iteration Network algorithmically such that it is the combination of three components: (a) an embedding function  $\Phi$  of the current observation, (b) the planning algorithm which runs  $K$  iterations on the embedded output, and (c) the control policy  $\pi$  which outputs the final action either via sampling during training or taking the max value at test time. In particular, the original VIN method can be summed up by these three choices:

- (a)  $\bar{r}_{i,j} = \Phi_{\theta}(s)_{i,j},$
- (b)  $v_{i,j}^0 = 0, \quad q_{a,i,j}^k = \sum_{(i',j') \in \mathcal{N}(i,j)} p_{a,i'-i,j'-j}^{(v)} * v_{i',j'}^{k-1} + p_{a,i'-i,j'-j}^{(r)} * \bar{r}_{i',j'}^{k-1}, \quad v_{i,j}^k = \max_a q_{a,i,j}^{k-1},$
- (c)  $\pi = F\left([q_{i',j'}^K, s_{i',j'}]_{(i',j') \in \mathcal{N}(i_0,j_0)}\right).$

where the agent is located at  $(i_0, j_0)$ , the agent’s policy is the output of a function  $F$  which takes the final propagation as input, the action is sampled from  $\pi$ , and  $\mathcal{N}(i, j)$  is the reachable neighbors of  $i, j$ . In a standard grid-world, this simply corresponds to the 8 surrounding locations if the agent is able to move in all the directions. Note that the  $q$  iteration in this case is exactly the convolution formulation described in Tamar et al. [2016], with convolution layer parameters  $p$ . Using this formulation of VIN it now becomes apparent that we can generalize it by improving on any of the elements (a) - (c).

Next we provide two novel alternatives to the Value Iteration Network, both of which can function as drop-in replacements for VIN and be used as differentiable low-level planners in graph-based tasks.

#### 3.1 Value-Propagation Module

A problematic and key limitation of VIN is its planning engine: the recursive VI module (1) encodes the transition probabilities as the weights of the convolution, which are naturally translation-invariant. This means that the transition probabilities at one state, a cell in this case, do not depend on its content or the content of the surrounding ones. This restriction affects the capability of the model to learn any non-trivial dynamics, while also artificially constraining the design of the architecture and other experimental parameters. Based on our formulation of VIN, we can reframe the architecture to include the capacity to represent more complex dynamics. We observe that the transition probabilities must depend on the state we are in, and thus we choose a factorization where  $\bar{r}_{i',j'}^{\text{in}}$  represents the value of *entering* a state,  $\bar{r}_{i,j}^{\text{out}}$  represents the value of *leaving* a state. The reward for making a transition from  $(i, j)$  to  $(i', j')$  takes on the additive form of  $\bar{r}_{i',j'}^{\text{in}} - \bar{r}_{i,j}^{\text{out}}$ , where  $\bar{r}^{\text{in}}$  and  $\bar{r}^{\text{out}}$  are both state dependent encodings.<sup>3</sup> We also introduce  $p_{i,j}$ , which represents the propagation of values into  $(i, j)$ . This results in a factored state dependent transition function, such that  $p(i, j|i', j') = p(i, j)$ , meaning that we only care about the state that the agent ends up in. This conditions the model parameters to emphasize state reachability, which provides an inexpensive and flexible way of modelling complex reward functions.

We define our new architecture, *Value Propagation* (VProp), by these three choices:

- (a)  $\bar{r}_{i,j}^{\text{in}}, \bar{r}_{i,j}^{\text{out}}, p_{i,j} = \Phi_{\theta}(s)_{i,j},$
- (b)  $v_{i,j}^0 = 0, \quad v_{i,j}^{(k)} = \max \left( v_{i,j}^{(k-1)}, \max_{(i',j') \in \mathcal{N}(i,j)} (p_{i,j} v_{i',j'}^{(k-1)} + \bar{r}_{i',j'}^{\text{in}} - \bar{r}_{i,j}^{\text{out}}) \right),$
- (c)  $\pi = F\left([p_{i_0,j_0} v_{i',j'}^{(K)} + \bar{r}_{i',j'}^{\text{in}} - \bar{r}_{i_0,j_0}^{\text{out}}, s_{i',j'}]_{(i',j') \in \mathcal{N}(i_0,j_0)}\right).$

VProp’s model corresponds to a deterministic model in which the reward propagates from adjacent states to current states, capturing the prior that the reachability of a state does not depend on the

<sup>3</sup>Note that while we use a minus sign, we do not impose specific sign constraints on  $\bar{r}^{\text{in}}$  and  $\bar{r}^{\text{out}}$ , since natural environments can easily be represented with both values as positive.

adjacent cell the agent is in, but by the underlying – and less constraining – observed transition dynamics. To represent unachievable states, the model sets their propagation to 0 and  $\bar{r}^{\text{in}}$  to some negative value.<sup>4</sup> Goal states, which are also absorbing for the underlying MDP, are represented with a propagation close 0, and a positive  $\bar{r}^{\text{in}}$ , whereas other types of cell are to have high propagation value bounded by the discount factor, while the cost of their traversal is represented by either a negative  $\bar{r}^{\text{in}}$  or a positive  $\bar{r}^{\text{out}}$ .

### 3.2 Max-Propagation Module

Since both VI module and the VProp’s module described above are both recursive, they should be able to generalize to larger environments than the ones seen during training time by simply increasing the recursion depth. However, the form of the planning procedure means that these models cannot generalize properly across the entire distribution of environment configurations. To see this, let us consider the pathfinding. Even in a simple grid-world with a single goal, there are many sets of parameters of the VI or VProp module that are equally good for pathfinding with a fixed environment size. For instance, in an environment of size  $20 \times 20$  in which shortest paths are of length at most 40, we can trivially check that a VProp module with a  $\bar{r}^{\text{in}}$  value of 1 for the goal entity and 0 otherwise,  $\bar{r}^{\text{out}}$  to  $-0.02$  for empty cells, and  $-1$  for the goal state, and with propagation  $p$  set to 1, everywhere, will effectively create a value map  $v^{(K)}$  which is affinely related to the shortest path to the goal on empty cells. However, if we increase the size of the environment such that paths can be longer than 50, the goal value would become  $1 - 51 \times 0.02$ , which is less than 0, hence stopping the propagation of the path after the 51st step. The model is therefore bound to the capacity of the additive rollout in the value map, which is responsible for both identifying the goal and computing the shortest path.

To solve this problem, we further introduce the Max-Value Propagation module (MVProp), which constrains the network to represent goals with a high base reward, propagates them multiplicatively through cells with lower values but high propagation, and learns reward and propagation maps depending on the content of the individual cells in the same fashion as VProp. More precisely, we define the module as:

$$\begin{aligned}
 \text{(a)} \quad & \bar{r}_{i,j}, p_{i,j} = \Phi_{\theta}(s)_{i,j}, \\
 \text{(b)} \quad & v_{i,j}^0 = \bar{r}_{i,j}, \quad v_{i,j}^{(k)} = \max \left( \bar{r}_{i,j}, \max_{(i',j') \in \mathcal{N}(i,j)} (\bar{r}_{i,j} + p_{i,j} (v_{i',j'}^{(k-1)} - \bar{r}_{i,j})) \right), \\
 \text{(c)} \quad & \pi = F \left( [v_{i',j'}^{(K)}, s_{i',j'}]_{(i',j') \in \mathcal{N}(i_0,j_0)} \right).
 \end{aligned}$$

This propagation system guarantees that values are propagated in the direction of lower-value cells at all times (even at the start of training), and that costs of individual cells are dealt with the  $1 - p$  map. In other words, the path length is propagated multiplicatively, whereas the reward map is used to distinguish between goal cells and other cells. Given this setup, the optimal policy should therefore be able to locally follow the direction of maximum value.

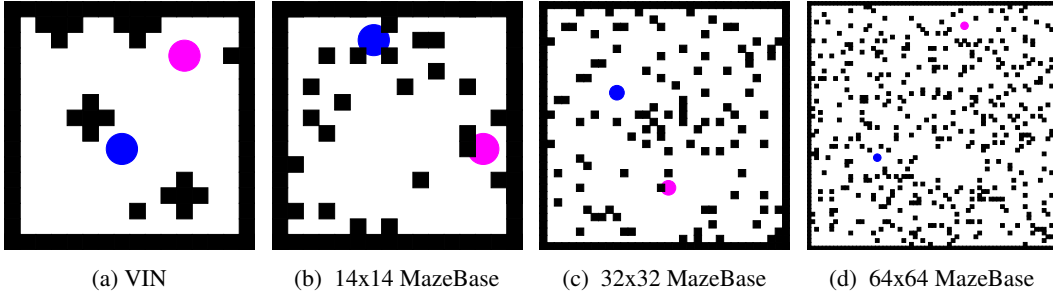


Figure 1: Comparison between a random map of the VIN dataset, and a few random configuration of our training environment. In our custom grid-worlds, the number of blocks increases with size, but their percentage over the total available space is kept fixed. Agent and goal are shown as circles for better visualization, however they still occupy a single cell.

<sup>4</sup>Absorbing states can be represented in the same way.

## 4 Experiments

We focus on evaluating our modules strictly using Reinforcement Learning, since we are interested in moving towards environments that better simulate tasks requiring interaction with the environment. For training, we use an actor-critic architecture with experience replay. We collect transition traces of the form  $(s^t, a^t, r^t, p^t, s^{t+1})$ , where  $s^t$  is the observation at time step  $t$ ,  $a^t$  is the action that was chosen,  $p^t$  is the vector of probabilities of actions as given by the policy, and  $r^t$  is the immediate reward. The architecture contains the policy  $\pi_\theta$  described in the previous sections, together with a value function  $V_w$ , which takes the same input as the softmax layer of the policy, concatenated with the  $3 \times 3$  neighborhood of the agent.  $w$  and  $\theta$  share all their weights until the end of the convolutional recurrence. At training time, given the stochastic policy at time step  $t$  denoted by  $\pi_{\theta^t}$ , we sample a minibatch of  $B$  transitions, denoted  $\mathcal{B}$ , uniformly at random from the last  $L$  transitions, and perform gradient ascent over importance-weighted rewards:

$$\begin{aligned} \theta^{t+1} &\leftarrow \theta^t + \eta \sum_{(s,a,r,p,s') \in \mathcal{B}} \min\left(\frac{\pi_{\theta^t}(s,a)}{p(a)}, C\right) \left(r + \mathbf{1}_{\{s' \neq \emptyset\}} V_{w^t}(s') - V_{w^t}(s)\right) (\nabla_{\theta^t} \log \pi_{\theta^t}(s,a)) \\ &\quad + \lambda \sum_{(s,a,r,p,s') \in \mathcal{B}} \sum_{a'} p(a') (\nabla_{\theta^t} \log \pi_{\theta^t}(s,a')), \\ w^{t+1} &\leftarrow w^t + \eta' \sum_{(s,a,r,p,s') \in \mathcal{B}} \min\left(\frac{\pi_{\theta^t}(s,a)}{p(a)}, C\right) (\nabla_{w^t} V_{w^t}(s) - r - \mathbf{1}_{\{s' \neq \emptyset\}} V_{w^t}(s'))^2, \end{aligned}$$

where  $\mathbf{1}_{\{s' \neq \emptyset\}}$  is 1 if  $s'$  is terminal and 0 otherwise. The capped importance weights  $\min\left(\frac{\pi_{\theta^t}(s,a)}{p(a)}, C\right)$  are standard in off-policy policy gradient (see e.g. [Wang et al., 2016], and references therein). The capping constant ( $C = 10$  in our experiments) controls the variance of the gradients at the expense of some bias. The second term of the update acts as a regularizer and forces the current predictions to be close enough to the ones that were made by the older model. The learning rates  $\eta$ ,  $\lambda$  and  $\eta'$  also control the relative weighting of the different objectives when the weights are shared.

### 4.1 Grid-world setting

Our experimental setting consists of a 2D grid-world of fixed dimensions where all entities are sampled based on some fixed distribution (Figure 1). The agent is allowed to move in all 8 directions at each step, and a terminal state is reached when the agent either reaches the goal or hits one of the walls. We use MazeBase [Sukhbaatar et al., 2015] to generate the configurations of our world and the agent interface for both training and testing phases. Additionally we also evaluate our trained agents on maps uniformly sampled from the  $16 \times 16$  dataset originally used by Tamar et al. [2016], so as to get a direct comparison with the previous work, and to confirm the quality of our baseline. We tested all the models on the other available datasets ( $8 \times 8$  and  $28 \times 28$ ) too, without seeing significant changes in relative performance, so they are omitted from our evaluation. We employ a curriculum where the average length of the optimal path from the starting agent position is bounded by some value which gradually increases after a few training episodes. This makes it more likely to encounter the goal at early stages of training, allowing for easier conditioning over the goal feature. Across all our tests on this setup, both VProp and MVProp greatly outperformed our implementation of VIN. Figure 2 shows rewards obtained during training, averaged across 5 training runs seeded randomly. The original VIN architecture was mostly tested in a fully supervised setting (via imitation learning), where the best possible route was given to the network as target. In the appendix, however, Tamar et al. [2016] claim that VIN can perform in a RL setting, obtaining an 82.5% success rate, versus the 99.3% success rate of the supervised setting on a map of  $16 \times 16$ . The authors do not provide results for the larger  $28 \times 28$  map dataset, nor do they provide learning curves and variance, however overall these results are consistent with the *best* performance we obtained from testing our implementation.

The final average performances of each model against the static-world experiments clearly demonstrate the strength of VProp and MVProp. MVProp in particular very quickly learns a transition function over the MDP dynamics that is distinct enough that it provides good values across even bigger sizes, hence obtaining near-optimal policies over all the sizes during the first thousand training episodes.

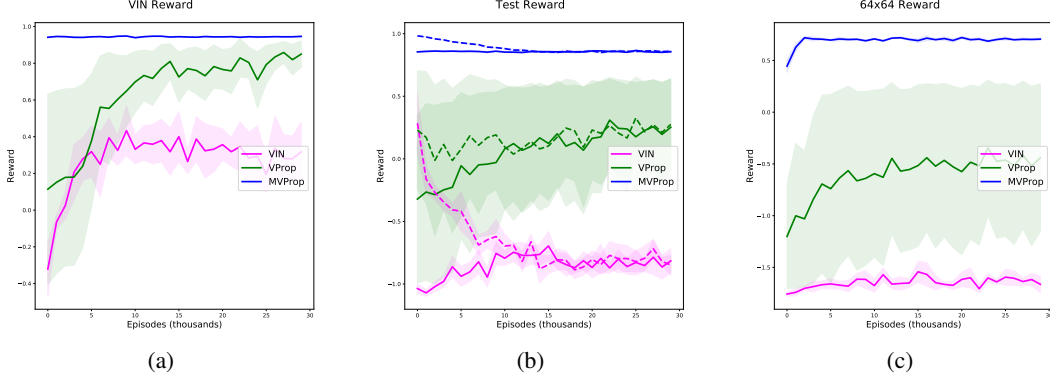


Figure 2: Average, min and max reward of all the models as they train on our curriculum. Note again that in the first two plots the the map size is  $32 \times 32$ . a and c demonstrate performances respectively on the VIN dataset and our generated  $64 \times 64$  maps. b shows performance on evaluation maps constrained by the curriculum settings (segmented line), and without (continuous line).

## 4.2 Tackling dynamic environments

To test the capability of our models to effectively learn non-static environments - that is, relatively more complex transition functions - we propose a set of experiments in which we allow our environment to spawn dynamic adversarial entities. Such entities at each step query some custom policy, which is executed in parallel to the agent's. Examples of these policies might include a  $\epsilon$ -noop strategy, which makes the entity move in random direction with probability  $\epsilon$  or do nothing, a  $\epsilon$ -direction policy, which makes the entity move to a specific direction with probability  $\epsilon$  or do nothing, or strictly adversarial policies such as one that makes such entities try to catch the agent before it can reach the goal. We use the first category of policies to augment our standard path-planning experiments, generating *enemies only* environments where 20% of the space is occupied by agents with  $\epsilon = 0.5$ , and *mixed* environments with the same amount of entities, half consisting of fixed walls, and the remaining of agents with  $\epsilon = 0.2$ . The second type of policies is instead used to generate a deterministic but continuously changing environment which we call *avalanche*, in which the agent is tasked to reach the goal as quickly as possible while avoiding "falling" entities. Finally, we propose a third type of experiments where the latter fully adversarial policy is applied on 1 to 6 enemy entities depending on the size of the environment. Solving this final scenario clearly requires the planning module to learn off extremely sparse positive rewards, and thus creates a strong challenge for all the presented methods.

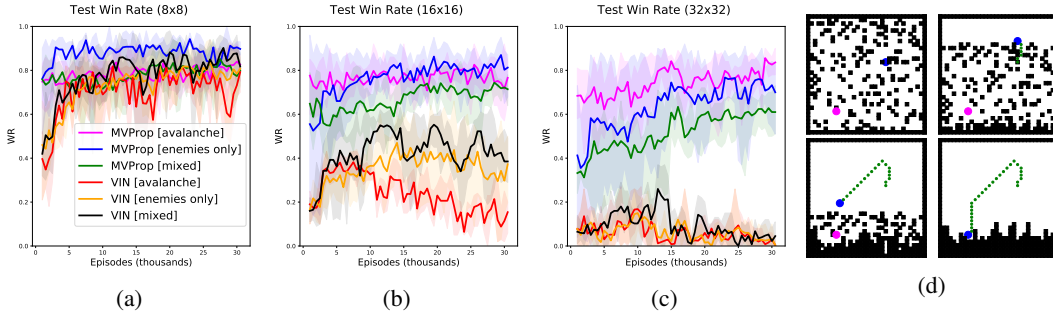


Figure 3: Average, min, and max, test win rate obtained on our dynamic experiments. Each agent was trained on the  $8 \times 8$  instances of the scenario in a similar fashion to the static world experiments. Figure 3d shows an example of policy obtained after training on a avalanche testing configuration. Agent and goal are shown as circles for better visualization, however they still occupy a single cell.

As these new environments are not anymore static, the agent needs to re-plan at every step, forcing us to train on  $8 \times 8$  maps to reduce the time spent rolling-out the recurrent modules. This however allows us to train without curriculum, as the agent is already likely to successfully hit the goal in a

smaller area with a stochastic policy. Figure 3 shows that VProp learns to handle this new complexity in the dynamics, successfully generalising to 32x32 maps, much larger than the ones seen during training (Figure 3d). VIN instead gradually loses performance on any larger sizes, since the added planning complexity can be tackled only when the dynamics are modelled correctly and the decision boundaries are sharp enough.

### 4.3 StarCraft navigation

Finally, we evaluate VProp on a navigation task in *StarCraft: Brood War* where the navigation-related actions have low-level physical dynamics that affect the transition function. In *StarCraft*, it is common to want to plan a trajectory around enemy units, as these have auto-attack behaviours that will interfere or even destroy your own unit if found navigating too close to them. While planning trajectories of the size of a standard *StarCraft* map is outside of the scope of this work, this problem becomes already quite difficult when considering the points of the plan where enemies are close to the planner (sub-)goals’ positions, thus we can test our architecture on instances of these scenarios.

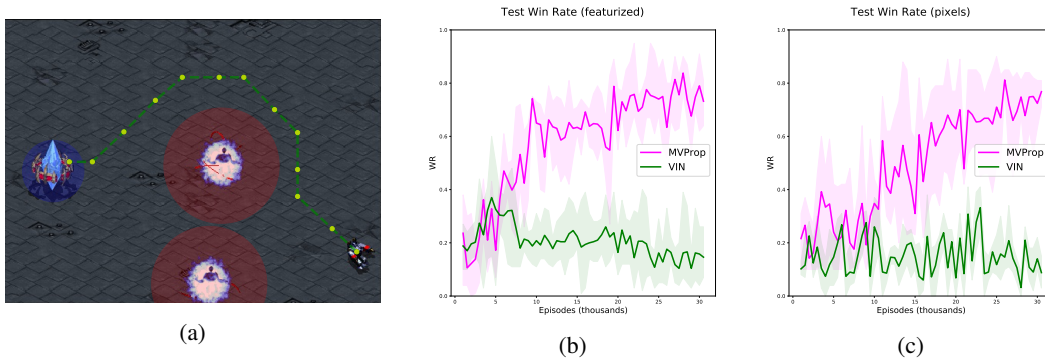


Figure 4: *StarCraft* navigation results. Figure 4a shows a generated trajectory on a random scenario at late stages of training. The red and blue overlays (not shown to the agent) indicate the distance required to interact with each entity.

We use TorchCraft [Synnaeve et al., 2016] to setup the environment and extract the position and type of units randomly spawned in the scenario. The state space is larger than the one used in our previous experiments and positive rewards might be extremely sparse, thus we employ a mixed curriculum to sample the units and their positions, allowing the models to observe positive rewards more often at the early stages of training and speed up training (note that this is required for the VIN baseline to have an a chance at the task [Tamar et al., 2016]). As shown in Figure 4b, the model quickly converges to a good representation of the value, which enables the agent to navigate around the lethal enemies and reach the goal. Compared to the VIN baseline, the ability to accurately learn a model of the state-action transition function allows VProp to learn planning modules for non-trivial environments. We also evaluate on the scenario directly from pixels, by adding two convolutional layers to provide capacity to learn the state features: as expected (Figure 4c) the architecture takes some more time to condition on the entities, but ultimately reaches similar final performances. In both cases, even with the curriculum providing early positive feedback, VIN struggles to condition correctly on the more complex transition function, often resulting in “suicidal” or simply dangerous plans.

## 5 Conclusions

Architectures that try to solve the large but structured space of navigation tasks have much to benefit from employing planners that can be learnt from data, however these need to quickly adapt to local environment dynamics so that they can provide a flexible planning horizon without the need to collect new data. Value Propagation’ performance show that, if the problem is carefully formalized, such planners can be successfully learnt via Reinforcement Learning, and that great generalization capabilities can be expected when these models are correctly applied to 2D path-planning tasks. Furthermore, we have demonstrated that our methods can even generalize when the environments are dynamic or with high-dimensional observation spaces, enabling them to be employed in relatively



complex tasks. In future we expect to test our methods on a variety of tasks that can be embedded as graph-like structures (and for which we have the relevant convolutional operators). We also plan to evaluate the effects of plugging VProp into architectures that are employing VI modules (see Section 1.1), since most of these models could make use of the ability to tackle more complex interactive environments. Finally, VProp could be applied to algorithms used in mobile robotics and visual tracking [Lee et al., 2017, Bordallo et al., 2015], as it can learn to propagate arbitrary value functions and model a wide range of potential functions.

## References

- A. Banino, C. Barry, B. Uria, C. Blundell, T. Lillicrap, P. Mirowski, A. Pritzel, M. J. Chadwick, T. Degris, J. Modayil, et al. Vector-based navigation using grid-like representations in artificial agents. *Nature*, page 1, 2018.
- D. P. Bertsekas. *Dynamic Programming and Optimal Control, Vol. II*. Athena Scientific, 4th edition, 2012.
- S. Bhatti, A. Desmaison, O. Miksik, N. Nardelli, N. Siddharth, and P. H. Torr. Playing doom with slam-augmented deep reinforcement learning. *arXiv preprint arXiv:1612.00380*, 2016.
- A. Bordallo, F. Previtali, N. Nardelli, and S. Ramamoorthy. Counterfactual reasoning about intent for interactive navigation in dynamic environments. In *Intelligent Robots and Systems (IROS), 2015 IEEE/RSJ International Conference on*, pages 2943–2950. IEEE, 2015.
- G. Farquhar, T. Rocktäschel, M. Igl, and S. Whiteson. Treeqn and atreec: Differentiable tree planning for deep reinforcement learning. *arXiv preprint arXiv:1710.11417*, 2017.
- J. Foerster, N. Nardelli, G. Farquhar, P. Torr, P. Kohli, S. Whiteson, et al. Stabilising experience replay for deep multi-agent reinforcement learning. *arXiv preprint arXiv:1702.08887*, 2017.
- E. Groshev, A. Tamar, S. Srivastava, and P. Abbeel. Learning generalized reactive policies using deep neural networks. *arXiv preprint arXiv:1708.07280*, 2017.
- S. Gupta, J. Davidson, S. Levine, R. Sukthankar, and J. Malik. Cognitive mapping and planning for visual navigation. *arXiv preprint arXiv:1702.03920*, 2017.
- L. P. Kaelbling, M. L. Littman, and A. W. Moore. Reinforcement learning: A survey. *Journal of artificial intelligence research*, 4:237–285, 1996.
- A. Khan, C. Zhang, N. Atanasov, K. Karydis, V. Kumar, and D. D. Lee. Memory augmented control networks. *arXiv preprint arXiv:1709.05706*, 2017.
- V. R. Konda and J. N. Tsitsiklis. Actor-critic algorithms. In *Advances in neural information processing systems*, pages 1008–1014, 2000.
- N. Lee, W. Choi, P. Vernaza, C. B. Choy, P. H. Torr, and M. Chandraker. Desire: Distant future prediction in dynamic scenes with interacting agents. *arXiv preprint arXiv:1704.04394*, 2017.
- P. Mirowski, R. Pascanu, F. Viola, H. Soyer, A. Ballard, A. Banino, M. Denil, R. Goroshin, L. Sifre, K. Kavukcuoglu, et al. Learning to navigate in complex environments. *arXiv preprint arXiv:1611.03673*, 2016.
- S. Niu, S. Chen, H. Guo, C. Targonski, M. C. Smith, and J. Kovačević. Generalized value iteration networks: Life beyond lattices. *arXiv preprint arXiv:1706.02416*, 2017.
- J. Oh, S. Singh, and H. Lee. Value prediction network. *arXiv preprint arXiv:1707.03497*, 2017.
- A. Paszke, S. Gross, S. Chintala, G. Chanan, E. Yang, Z. DeVito, Z. Lin, A. Desmaison, L. Antiga, and A. Lerer. Automatic differentiation in pytorch. 2017.
- E. Rehder, M. Naumann, N. O. Salscheider, and C. Stiller. Cooperative motion planning for non-holonomic agents with value iteration networks. *arXiv preprint arXiv:1709.05273*, 2017.

- S. Russell, P. Norvig, and A. Intelligence. A modern approach. *Artificial Intelligence. Prentice-Hall, Egnlewood Cliffs*, 25:27, 1995.
- D. Silver, H. van Hasselt, M. Hessel, T. Schaul, A. Guez, T. Harley, G. Dulac-Arnold, D. Reichert, N. Rabinowitz, A. Barreto, et al. The predictron: End-to-end learning and planning. *arXiv preprint arXiv:1612.08810*, 2016.
- S. Sukhbaatar, A. Szlam, G. Synnaeve, S. Chintala, and R. Fergus. Mazebase: A sandbox for learning from games. *arXiv preprint arXiv:1511.07401*, 2015.
- R. S. Sutton and A. G. Barto. *Reinforcement learning: An introduction*, volume 1. MIT press Cambridge, 1998.
- R. S. Sutton, D. A. McAllester, S. P. Singh, Y. Mansour, et al. Policy gradient methods for reinforcement learning with function approximation. In *NIPS*, volume 99, pages 1057–1063, 1999.
- G. Synnaeve, N. Nardelli, A. Auvolet, S. Chintala, T. Lacroix, Z. Lin, F. Richoux, and N. Usunier. Torchcraft: a library for machine learning research on real-time strategy games. *arXiv preprint arXiv:1611.00625*, 2016.
- A. Tamar, S. Levine, P. Abbeel, Y. WU, and G. Thomas. Value iteration networks. In *Advances in Neural Information Processing Systems*, pages 2146–2154, 2016.
- N. Usunier, G. Synnaeve, Z. Lin, and S. Chintala. Episodic exploration for deep deterministic policies: An application to starcraft micromanagement tasks. *arXiv preprint arXiv:1609.02993*, 2016.
- Z. Wang, V. Bapst, N. Heess, V. Mnih, R. Munos, K. Kavukcuoglu, and N. de Freitas. Sample efficient actor-critic with experience replay. *arXiv preprint arXiv:1611.01224*, 2016.
- C. J. Watkins and P. Dayan. Q-learning. *Machine learning*, 8(3-4):279–292, 1992.
- T. Weber, S. Racanière, D. P. Reichert, L. Buesing, A. Guez, D. J. Rezende, A. P. Badia, O. Vinyals, N. Heess, Y. Li, et al. Imagination-augmented agents for deep reinforcement learning. *arXiv preprint arXiv:1707.06203*, 2017.
- J. Zhang, L. Tai, J. Boedecker, W. Burgard, and M. Liu. Neural slam. *arXiv preprint arXiv:1706.09520*, 2017.

## A Agent setup

All the models and agent code was implemented in PyTorch [Paszke et al., 2017], and will be made available upon acceptance together with the environments. Most of the agents tested shared learning hyperparameters fitted to the VIN baseline to make comparison as fair as possible, and were validated using more than 5 random seeds across all our experiments. The second term in the  $\phi^{t_1}$  update is supposed to play the role of TRPO-like regularization as implemented in Wang et al. [2016], where they use probabilities of an average model instead of the previous probabilities. We implemented an n-step memory replay buffer, observing that keeping only the last 50000 transitions (to avoid trying to fit predictions of a bad model) worked well on our tasks. In all our experiments we used RMSProp rather than plain SGD, with relative weights  $\lambda = \eta = 100.0\eta'$ . We also used a learning rate to 0.001 and mini-batch size of 128, with learning updates set at a frequency of 32 steps. We tested reasonable ranges for all these hyperparameters, but observed no relative significant changes when cross-validated over multiple seeds for most of them.

## B MazeBase setup

The agents were tasked to navigate the maze as fast as possible, as total cost increased with time since *noop* actions were not allowed. Episodes terminated whenever the agent would take any illegal action such as hitting a wall, or when the maximum number of steps (set to roughly three times the length of the shortest path) would be reached. All entities were set with discrete collision boundaries corresponding to the featurised observation. The environments were also constrained so that only one entity could be present in any given cell at time  $t$ . Unless specified otherwise, attempting to walk into walls would yield a reward of  $-1$ , any valid movement would provide a reward of  $-0.01 \times f(a_t, s_t)$ , where  $f(a, s)$  is the cost of moving in the direction specified by action  $a$  in state  $s$ , and reaching the goal would give a positive reward of 1 to the agent. In our experiments we define  $f$  as the L2 distance between the agent position at state  $s_t$ , and the one at  $s_{t+1}$ , to adjust for the real cost of moving diagonally.

For all static experiments, the ratio of un-walkable blocks over total space was fixed to 30%, and the blocks were sampled uniformly within the space, unless specified otherwise. This setting provided environments of decent difficulty, with chunky obstacles as well as harder, more narrow paths. Dynamic environments used different ratios of blocks (and other entities) vs walkable surface:

maps	VIN	VProp	MVProp	VIN	VProp	MVProp
	win rate			distance to optimal path		
<i>v</i> 16x16	63.6% $\pm$ 13.2%	94.4% $\pm$ 5.6%	100%	0.2 $\pm$ 0.2	0.2 $\pm$ 0.2	0.0 $\pm$ .0
32x32	15.6% $\pm$ 5.3%	68.8% $\pm$ 27.2%	100%	0.8 $\pm$ 0.3	0.4 $\pm$ 0.3	0.0 $\pm$ .0
64x64	4.0% $\pm$ 4.1%	53.2% $\pm$ 31.8%	100%	1.5 $\pm$ 0.4	0.5 $\pm$ 0.4	0.0 $\pm$ .0

Table 1: Average performance at the end of training of all tested models on the static grid-worlds with 90% confidence value, across 5 different training runs (with random seeding). *v*16x16 correspond to the maps sampled from VIN’s 16x16 grid test dataset, while the rest of the maps are sampled uniformly from our generator using the same parameters employed at training time. The distance to the optimal path is averaged only for successful episodes.

*avalanche* environments filled between 20% and 30% of the surface with ”falling” entities, *enemies only* spawned 10% (rounded to the bigger integer) of the surface with adversarial agents running A\*, while *mixed* environments employed the same amount of adversarial agents with the addition of static and stochastic entities (with and  $\epsilon$ -greedy stochastic policy) making up for 10% of the surface area, for a total of roughly 20% occupied blocks.

Our VIN models were based off the architecture employed by Tamar et al. [2016] in their grid-world experiments, which we used to also build VProp and MVProp models. The only significant difference between the models (beyond the number of maps used in the recurrency) consisted in VProp and MVProp using 8 input filters and unpadded convolutions. Note that we tested our VIN models using the same setup and saw no significant difference in these tests either.

## C StarCraft setup

As TorchCraft by default runs at a very high framerate, we had to set the amount of skipped frames to 15, to allow us to be able to see relevant changes in the environment after each step; this is roughly double compared to other work done on the same platform [Usunier et al., 2016, Foerster et al., 2017]. The rest of the environment parameters were kept to the default values. All entities were spawned in a similar fashion to the grid-world experiments, with an additional minimal constraint based on each entity’s pixel size. At training time we employed a fixed max-distance curriculum to gradually increase the distance between spawned goal and agent, however we also prevented enemies from spawning during the first 500 episodes so as to allow the stochastic policy to quickly condition on the goal. Without this particular change in the curriculum setting we found learning to be generally more unstable, since the stochastic policy would need to naturally luck out. This could have been fixed by utilising a better exploration strategy, but we considered it outside the scope of this work.

We used a fixed 8x downsampling operator to reduce the size of the raw observation of 480x360 pixels to 60x45. Based on the experiment, this downsampled observation was then either fully featurised in a grid-world fashion, or transformed into greyscale (as it is typically done in other deep reinforcement learning experimental settings). We increased the capacity of the models by adding 2 additional convolutional layers with 32 filters and 7x7 and 5x5 kernels, and max-pooling layers with stride and extent equal to 2 to further reduce the dimensionality before the recurrent step.