XLVIN: eXecuted Latent Value Iteration Nets

Anonymous Author(s)
Affiliation
Address
email

Abstract

Value Iteration Networks (VINs) have emerged as a popular method to perform implicit planning within deep reinforcement learning, enabling performance improvements on tasks requiring long-range reasoning and understanding of environment dynamics. This came with several limitations, however: the model is not explicitly incentivised to perform meaningful planning computations, the underlying state space is assumed to be discrete, and the Markov decision process (MDP) is assumed fixed and known. We propose eXecuted Latent Value Iteration Networks (XLVINs), which combine recent developments across contrastive self-supervised learning, graph representation learning and neural algorithmic reasoning to alleviate *all* of the above limitations, successfully deploying VIN-style models on generic environments. XLVINs match the performance of VIN-like models when the underlying MDP is discrete, fixed and known, and provide significant improvements to model-free baselines across three general MDP setups.

1 Introduction and Background

Planning is an important aspect of reinforcement learning (RL) algorithms, and planning algorithms are usually characterised by explicit modelling of the environment. Recently, several approaches explore *implicit planning* (also called *model-free planning*) [25, 19, 20, 22, 18, 12, 11], which proposes inductive biases in the policy function to enable planning to emerge, while training the policy in a model-free manner. *Value iteration networks* (VINs) interpret the value iteration (VI) algorithm on a grid-world as a convolution across state values followed by max-pooling, yielding a CNN-based VI module [25]. *Generalized value iteration networks* (GVINs), based on graph kernels [31], lift the assumption that the environment is a grid-world and allow planning on irregular discrete state spaces [18], such as graphs. While highly influential, both VINs and GVINs assume discrete state spaces, incurring loss of information for problems with naturally continuous state spaces. Most importantly, both approaches require the underlying Markov decision process (MDP) to be known in advance and are inapplicable if it is too large to be stored in memory, or in other ways inaccessible.

We propose the **eXecuted Latent Value Iteration Network** (XLVIN), an implicit planning policy network which embodies the computation of VIN-like models while addressing the above issues. We seamlessly run XLVINs with minimal configuration changes on environments from MDPs with known structure (such as grid-worlds), through pixel-based ones (such as Atari), to continuous-state environments, outperforming baselines lacking XLVIN's inductive biases. XLVINs unify emerging concepts from several areas of representation learning: *contrastive self-supervised learning* [15, 26], *graph representation learning* [4, 7, 13], and *neural algorithm execution* [29, 28, 30, 10, 27].

Value iteration is a method for finding optimal value functions of *Markov decision processes* (MDPs). For an MDP with states $s \in \mathcal{S}$, actions $a \in \mathcal{A}$, transition model $p : \mathcal{S} \times \mathcal{A} \times \mathcal{S} \to [0, 1]$ and reward model $r : \mathcal{S} \times \mathcal{A} \to \mathbb{R}$, value iteration performs the following update:

$$v^{(t+1)}(s) = \max_{a \in \mathcal{A}_s} r(s, a) + \gamma \sum_{s' \in \mathcal{S}} p(s'|s, a) v^{(t)}(s') \tag{1}$$

Submitted to 34th Conference on Neural Information Processing Systems (NeurIPS 2020). Do not distribute.

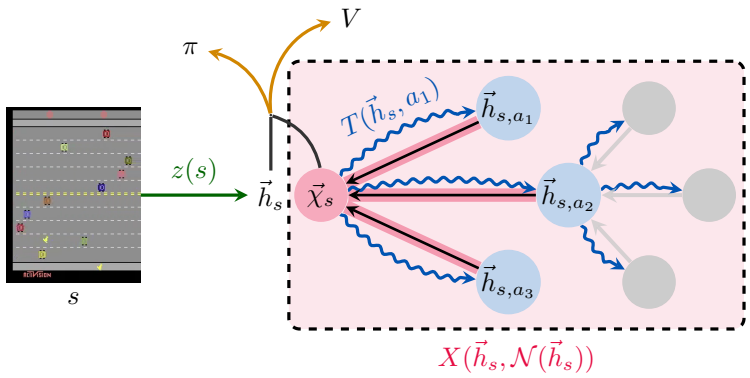

Figure 1: XLVIN model summary. Its modules are explained (and colour-coded) in Section 2.

where $v^{(t)} : \mathcal{S} \to \mathbb{R}$ is the estimate of v^*, the optimal discounted cumulative return, at step $t \in \mathbb{N}$ of the algorithm, and $\gamma \in [0, 1)$ is a discount factor.

An important research direction explores the use of neural networks for learning to execute algorithms—which was recently extended to algorithms on graph-structured data [28]. In particular, [29] establishes *algorithmic alignment* between GNNs and dynamic programming algorithms. As VI is, in fact, a dynamic programming algorithm, a graph neural network executor is a suitable choice for learning it. With intermediate supervision, good results on executing VI have emerged on synthetic graphs [8], establishing it as a potential planning module in an RL environment.

2 XLVIN Architecture

Throughout this section, we recommend referring to Figure 1 for a visualisation of the overall model dataflow. We start by presenting the modules of XLVIN in turn.

The **encoder**, $z : \mathcal{S} \to \mathbb{R}^k$, embeds states $s \in \mathcal{S}$ into flat embeddings, $\vec{h}_s = z(s)$.

The **transition** function, $T : \mathbb{R}^k \times \mathcal{A} \to \mathbb{R}^k$, models the effects of taking actions. From a state embedding $z(s)$ and an action a, it produces a *translation* of the embedding, to match the successor-state embedding (in expectation). It is desirable that T satisfies the following:

$$z(s) + T(z(s), a) \approx \mathbb{E}_{s' \sim P(s'|s,a)} z(s') \quad (2)$$

The **executor**, $X : \mathbb{R}^k \times \mathbb{R}^{|\mathcal{A}| \times k} \to \mathbb{R}^k$, generalises the planning modules proposed in VIN [25] and GVIN [18]. It processes an embedding $\vec{h}_s$ of state s, alongside a neighbourhood set $\mathcal{N}(\vec{h}_s)$, which contains (expected) embeddings of states that neighbour s—e.g., through taking actions. Hence,

Algorithm 1: XLVIN forward prop

Input : Input state s, executor depth K
Output: Policy function $\pi_\theta(a|s)$,
state-value function $V(s)$

$\vec{h}_s = z(s)$, $\mathbb{S}^0 = \{\vec{h}_s\}$, $\mathbb{E} = \emptyset$
for $k \in [0, K]$ **do**
  $\mathbb{S}^{k+1} = \emptyset$
  for $\vec{h} \in \mathbb{S}^k$, $a \in \mathcal{A}$ **do**
    $\vec{h}' = \vec{h} + T(\vec{h}, a)$
    $\mathbb{S}^{k+1} = \mathbb{S}^{k+1} \cup \{\vec{h}'\}$
    $\mathbb{E} = \mathbb{E} \cup \{(\vec{h}, \vec{h}', a)\}$
  end
  for $\vec{h} \in \mathbb{S}^k$ **do**
    $\mathcal{N}(\vec{h}) = \{\vec{h}' \mid \exists \alpha. (\vec{h}, \vec{h}', \alpha) \in \mathbb{E}\}$
  end
end
$\vec{\chi}_s = X(\vec{h}_s, \mathcal{N}(\vec{h}_s))$
$\pi_\theta(s, \cdot) = A(\vec{h}_s, \vec{\chi}_s)$; $V(s) = V(\vec{h}_s, \vec{\chi}_s)$

$$\mathcal{N}(\vec{h}_s) \approx \left\{\mathbb{E}_{s' \sim \mathbb{P}(s'|s,a)} z(s')\right\}_{a \in \mathcal{A}} \quad (3)$$

The executor combines the neighbourhood set features to produce an updated embedding of state s, $\vec{\chi}_s = X(\vec{h}_s, \mathcal{N}(\vec{h}_s))$. Ideally, X should mimic the one-step behaviour of VI, allowing for the model to meaningfully plan from state s by stacking several layers of X.

The **actor**, $A : \mathbb{R}^k \times \mathbb{R}^k \to [0, 1]^{|\mathcal{A}|}$ consumes the state embedding $\vec{h}_s$ and the updated embedding $\vec{\chi}_s$, producing action probabilities $\pi_\theta(a|s) = A\left(\vec{h}_s, \vec{\chi}_s\right)_a$, specifying the policy to be followed

by the agent. Lastly, note that we may also have additional **tail** networks which have the same input as A. We train XLVINs using proximal policy optimisation (PPO) [21], which necessitates a state-value function: $V : \mathbb{R}^k \times \mathbb{R}^k \to \mathbb{R}$. Hence, we also include V as a component of our model.

We present, in Algorithm 1, a step-by-step algorithm used by XLVINs to derive a policy $\pi_\theta(a|s)$, for a given input state s and executor depth K.

The entire procedure is end-to-end differentiable, does not impose any assumptions on the structure of the underlying MDP, and has the capacity to perform computations directly aligned with value iteration. Hence our model can be considered as a generalisation of VIN-like methods to settings where the MDP is not provided or otherwise difficult to obtain. For details on how to train the modules to be mindful of presented constraints, consult Appendix A.

3 Experiments

We categorise our experimental environments into three types: known MDP, continuous observations, and pixel-space. For further details, see Appendix B. In all results that follow, we use **"PPO"** to denote our baseline model-free agent; it has no transition/executor model, but otherwise matches the XLVIN hyperparameters.

Known MDP In order to compare XLVINs performance against VINs and GVINs we evaluate on an established environment with a known, fixed and discrete MDP. We use the 8×8 grid-world mazes proposed by [25]. The observation for this environment consists of the maze image, the starting position of the agent and the goal position. Every maze is associated with a *difficulty* level, equal to the length of the shortest path between the start and the goal.

We formulate the *continual maze* task: the agent is, initially, trained to solve only mazes of difficulty 1. Once the agent reaches 95% success rate on the last 1,000 sampled episodes of difficulty d, it automatically advances to difficulty $d + 1$ (without observing difficulty d again). If the agent fails to reach 95% within 1,000,000 trajectories, it is considered to have *failed*. At each passed difficulty, the agent is evaluated by computing its success rate on held-out test mazes (see Appendix C for further statistics). We also test *out-of-distribution generalisation* (training on 8×8 mazes, testing on 16×16). Our encoder is a three-layer CNN computing 128 latent features and 10 outputs—which we describe in Appendix D. The VIN baseline [25] *slices* the features on the agent's coordinates, which assumes upfront knowledge of where the agent actually is on the map, putting VINs in a privileged position. For a fairer comparison, we also attempted to train **VIN-mean** and **VIN-max**—VIN models where the slicing is replaced by global average pooling or global max pooling.

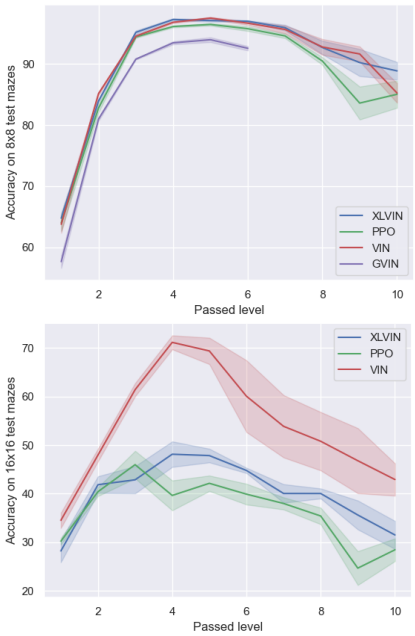

Figure 2: Success rate on 8×8 (**top**) and 16×16 (**bottom**) held-out mazes for XLVIN, PPO, VIN and GVIN, obtained after passing each level of 8×8 train mazes. VIN-mean and VIN-max **failed** to pass difficulty 1.

The transition function is a three-layer MLP with layer normalisation [2] after the second layer, for all environments. For mazes, it computes 128 hidden features and applies ReLU. The executor is, for all environments, identical to the message passing executor of [8]. For mazes, we exploit the fact that the MDP is known: we train the executor exactly on the graph structures generated from the training mazes; please see Appendix E for further details. We apply the executor until depth $K = 4$, with layer normalisation [2] applied after every step.

The results of our continual maze evaluation are summarised in Figure 2. In-distribution, the XLVIN is competitive with all other models. XLVINs remain competitive with the baseline when evaluated out-of-distribution. There is a gap to VINs, which can be attributed to the previously mentioned slicing—corroborated by the fact both VIN-mean and -max *failed* to pass even level 1 of the 8×8

Table 1: Mean scores for CartPole-v0, Acrobot-v1 and MountainCar-v0 after training, averaged over 100 episodes and five seeds. Baseline CartPole results reprinted from [26].

CartPole-v0	100 trajectories	**Only 10 trajectories**
REINFORCE	23.84 $\pm$ 0.88	-
WM-AE	114.47 $\pm$ 17.32	-
LD-AE	154.73 $\pm$ 50.49	-
DMDP-H ($J = 0$)	72.81 $\pm$ 20.16	-
PRAE, $J = 5$	**171.53** $\pm$ 34.18	-
PPO	-	104.6 $\pm$ 48.5
XLVIN	-	**195.2** $\pm$ 5.0

Acrobot-v1	Score
PPO	-500.0 $\pm$ 0.0
XLVIN	**-245.4** $\pm$ 48.4

MountainCar-v0	Score
PPO	-200.0 $\pm$ 0.0
XLVIN	**-168.9** $\pm$ 24.7

mazes. We also performed a qualitative study into the node embeddings learnt by the encoder (and transition model), hoping to elucidate the mechanism in which XLVIN organises its plan—please see Appendix F.

Continuous-space observations The latent-space execution of XLVINs allow us to deploy it in generic environments, including continuous-space environments. We investigate the performance of XLVIN on classical control environments: CartPole-v0, Acrobot-v1 and MountainCar-v0.

The executor has been trained from synthetic graphs which are designed to imitate the dynamics of CartPole very crudely—more details on the graph construction are given in Appendix E. The same trained executor is then deployed across all environments, to demonstrate robustness to synthetic graph construction. In all cases, the XLVIN uses $K = 2$ executor layers.

We make the CartPole task challenging by sampling *only 10 trajectories* at the beginning, and not allowing any further interaction—beyond 100 epochs of training on this dataset. For Acrobot and MountainCar, considering they are both sparse-reward, we sample 5 trajectories at a time, twenty times during training (a total of 100 trajectories).

Results are provided in Table 1. For CartPole, the XLVIN model solves the environment from only 10 trajectories, outperforming all the results given in [26], while using 10$\times$ fewer samples. Further, our model is capable of solving the Acrobot and MountainCar environments from only 100 trajectories, in spite of sparse rewards.

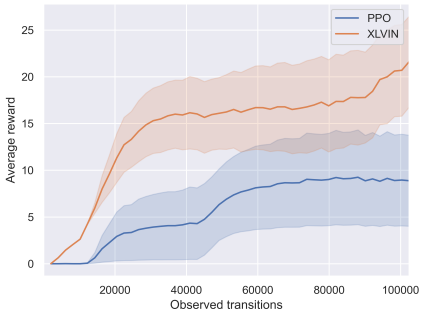

Figure 3: Average reward on Freeway over 1,000,000 processed transitions.

Pixel-space Lastly, we investigate how XLVINs perform on high-dimensional pixel-based observations, using the Atari-2600 [5]. We focus on the Freeway game, which could require short-range planning for avoiding traffic, and is known to be challenging for policy gradient methods given reasonably sparse rewards. Further, its "up-and-down" structure aligns it somewhat to environments like CartPole, and we successfully *re-use* the trained executor from CartPole. We once again evaluate the agents' low-data performance by allowing only 1,000,000 observed transitions. We re-use exactly the environment and encoder from here[1], and run the executor for $K = 2$ layers.

The average reward can be visualised in Figure 3. From the inception of the training, the XLVIN model explores and exploits better, consistently remaining ahead of the baseline model despite the sparse reward structure. The fact that its executor achieved so, while being transferred from a CartPole inspired graph (Appendix E), is a further statement to the model's robustness.

[1] `https://github.com/ikostrikov/pytorch-a2c-ppo-acktr-gail/blob/master/a2c_ppo_acktr/model.py#L169-L195`

References

[1] Réka Albert and Albert-László Barabási. Statistical mechanics of complex networks. *Reviews of modern physics*, 74(1):47, 2002.

[2] Jimmy Lei Ba, Jamie Ryan Kiros, and Geoffrey E Hinton. Layer normalization. *arXiv preprint arXiv:1607.06450*, 2016.

[3] Andrew G Barto, Richard S Sutton, and Charles W Anderson. Neuronlike adaptive elements that can solve difficult learning control problems. *IEEE transactions on systems, man, and cybernetics*, (5):834–846, 1983.

[4] Peter W Battaglia, Jessica B Hamrick, Victor Bapst, Alvaro Sanchez-Gonzalez, Vinicius Zambaldi, Mateusz Malinowski, Andrea Tacchetti, David Raposo, Adam Santoro, Ryan Faulkner, et al. Relational inductive biases, deep learning, and graph networks. *arXiv preprint arXiv:1806.01261*, 2018.

[5] Marc G Bellemare, Yavar Naddaf, Joel Veness, and Michael Bowling. The arcade learning environment: An evaluation platform for general agents. *Journal of Artificial Intelligence Research*, 47:253–279, 2013.

[6] Antoine Bordes, Nicolas Usunier, Alberto García-Durán, Jason Weston, and Oksana Yakhnenko. Translating embeddings for modeling multi-relational data. In *NIPS*, pages 2787–2795, 2013.

[7] Michael M Bronstein, Joan Bruna, Yann LeCun, Arthur Szlam, and Pierre Vandergheynst. Geometric deep learning: going beyond euclidean data. *IEEE Signal Processing Magazine*, 34(4):18–42, 2017.

[8] Andreea Deac, Pierre-Luc Bacon, and Jian Tang. Graph neural induction of value iteration. *arXiv preprint arXiv:2009.12604*, 2020.

[9] Paul Erdős and Alfréd Rényi. On the evolution of random graphs. *Publ. Math. Inst. Hung. Acad. Sci*, 5(1):17–60, 1960.

[10] Dobrik Georgiev and Pietro Lió. Neural bipartite matching. *arXiv preprint arXiv:2005.11304*, 2020.

[11] Arthur Guez, Mehdi Mirza, Karol Gregor, Rishabh Kabra, Sébastien Racanière, Theophane Weber, David Raposo, Adam Santoro, Laurent Orseau, Tom Eccles, Greg Wayne, David Silver, and Timothy P. Lillicrap. An investigation of model-free planning. In *ICML*, volume 97, pages 2464–2473, 2019.

[12] Arthur Guez, Theophane Weber, Ioannis Antonoglou, Karen Simonyan, Oriol Vinyals, Daan Wierstra, Rémi Munos, and David Silver. Learning to search with mctsnets. In *ICML*, volume 80, pages 1817–1826. PMLR, 2018.

[13] William L Hamilton, Rex Ying, and Jure Leskovec. Representation learning on graphs: Methods and applications. *arXiv preprint arXiv:1709.05584*, 2017.

[14] Sergey Ioffe and Christian Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift. *arXiv preprint arXiv:1502.03167*, 2015.

[15] Thomas N. Kipf, Elise van der Pol, and Max Welling. Contrastive learning of structured world models. In *ICLR*, 2020.

[16] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A Rusu, Joel Veness, Marc G Bellemare, Alex Graves, Martin Riedmiller, Andreas K Fidjeland, Georg Ostrovski, et al. Human-level control through deep reinforcement learning. *nature*, 518(7540):529–533, 2015.

[17] Andrew William Moore. Efficient memory-based learning for robot control. 1990.

[18] Sufeng Niu, Siheng Chen, Hanyu Guo, Colin Targonski, Melissa C. Smith, and Jelena Kovacevic. Generalized value iteration networks: Life beyond lattices. In *AAAI*, pages 6246–6253. AAAI Press, 2018.

[19] Junhyuk Oh, Satinder Singh, and Honglak Lee. Value prediction network. In *NIPS*, pages 6118–6128, 2017.

[20] Sébastien Racanière, Theophane Weber, David P. Reichert, Lars Buesing, Arthur Guez, Danilo Jimenez Rezende, Adrià Puigdomènech Badia, Oriol Vinyals, Nicolas Heess, Yujia Li, Razvan Pascanu, Peter W. Battaglia, Demis Hassabis, David Silver, and Daan Wierstra. Imagination-augmented agents for deep reinforcement learning. In *NIPS*, pages 5690–5701, 2017.

[21] John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. Proximal policy optimization algorithms. *arXiv preprint arXiv:1707.06347*, 2017.

[22] David Silver, Hado van Hasselt, Matteo Hessel, Tom Schaul, Arthur Guez, Tim Harley, Gabriel Dulac-Arnold, David P. Reichert, Neil C. Rabinowitz, André Barreto, and Thomas Degris. The predictron: End-to-end learning and planning. In *ICML*, volume 70, pages 3191–3199, 2017.

[23] Jost Tobias Springenberg, Alexey Dosovitskiy, Thomas Brox, and Martin Riedmiller. Striving for simplicity: The all convolutional net. *arXiv preprint arXiv:1412.6806*, 2014.

[24] Richard S Sutton. Generalization in reinforcement learning: Successful examples using sparse coarse coding. In *Advances in neural information processing systems*, pages 1038–1044, 1996.

[25] Aviv Tamar, Sergey Levine, Pieter Abbeel, Yi Wu, and Garrett Thomas. Value iteration networks. In Daniel D. Lee, Masashi Sugiyama, Ulrike von Luxburg, Isabelle Guyon, and Roman Garnett, editors, *NIPS*, pages 2146–2154, 2016.

[26] Elise van der Pol, Thomas Kipf, Frans A. Oliehoek, and Max Welling. Plannable approximations to mdp homomorphisms: Equivariance under actions. In *AAMAS 2020*, 2020.

[27] Petar Veličković, Lars Buesing, Matthew C Overlan, Razvan Pascanu, Oriol Vinyals, and Charles Blundell. Pointer graph networks. *arXiv preprint arXiv:2006.06380*, 2020.

[28] Petar Veličković, Rex Ying, Matilde Padovano, Raia Hadsell, and Charles Blundell. Neural execution of graph algorithms. *arXiv preprint arXiv:1910.10593*, 2019.

[29] Keyulu Xu, Jingling Li, Mozhi Zhang, Simon S Du, Ken-ichi Kawarabayashi, and Stefanie Jegelka. What can neural networks reason about? *arXiv preprint arXiv:1905.13211*, 2019.

[30] Yujun Yan, Kevin Swersky, Danai Koutra, Parthasarathy Ranganathan, and Milad Hashemi. Neural Execution Engines, 2020.

[31] Pinar Yanardag and SVN Vishwanathan. Deep graph kernels. In *Proceedings of the 21th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 1365–1374, 2015.

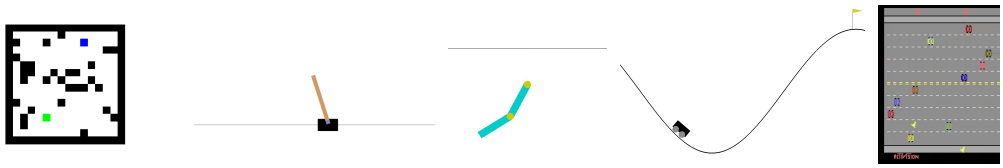

Figure 4: The five environments considered within our evaluation: 8×8 and 16×16 mazes [25] (known grid-like MDP), continuous control environments (CartPole-v0, Acrobot-v1, MountainCar-v0) and Atari Freeway (pixel-based space).

A XLVIN (pre-)training details

Our transition function, T, and executor function, X, should ideally respect certain properties (e.g. Equations 2–3). We *pre-train* both of them using established methods in the literature; TransE [15, 26] for the transition function and neural algorithm execution [8] for the executor.

The transition condition of Equation 2 perfectly aligns with the TransE loss [6], hence we pre-train T by optimising its loss function:

$$\mathcal{L} = d(z(s) + T(z(s), a), z(s')) + \max(0, \alpha - d(z(\tilde{s}), z(s'))) \tag{4}$$

using transitions sampled in the environment using a random policy—not unlike the prior work by [15, 26] that also trains transition functions in this way.

The desirable behaviour of the executor is to align with VI, and hence we pre-train X to be predictive of the value updates performed by VI, following the work of [8]. Note that standard VI procedure requires access to a fully-specified MDP, over which we can generate VI trajectories to use as training data. When an MDP is not available, following the remarks of [8], we generate synthetic discrete MDPs that align with the target MDP as much as possible[2]—finding that useful transfer still occurs, echoing the findings of [8].

To summarise in brief, the executor function training proceeds as follows:

1. First, generate a dataset of MDPs which, as much as possible, mimics in some way the characteristics of the true MDP we'd like to deploy on (e.g. determinism, number of actions, etc.). If no such knowledge is available upfront, resort to a generic graph distribution like Erdős-Rényi or Barabási-Albert.
2. Then, execute the Value Iteration algorithm on these MDPs, keeping track of intermediate values $V_t(s)$ at each iteration t, until convergence.
3. Supervise the executor network—a graph neural network operating over the transitions of the MDP as edges—to receive $V_t(s)$—and all other parameters of the MDP—as inputs, and predict $V_{t+1}(s)$ as outputs (optimise using mean-squared error).

To optimise the neural network parameters θ, we use proximal policy optimisation (PPO)[3] [21]. Note that the PPO gradients also flow into parameters of T and X, which has the potential to displace them from the properties required by the above, leading to either poorly constructed graphs (that don't respect the underlying MDP) or lack of VI-aligned computation. For the transition model, we resolve this by periodically re-training on the TransE loss during training (with a multiplicative factor of 0.001 to the loss). As we have no such easy way of retraining the executor, X, without knowledge of the underlying MDP, we instead opt to *freeze* the parameters of X after pre-training, treating them as constants rather than parameters to be optimised.

B Environments under study

We provide a visual overview of all five environments considered in Figure 4.

[2] If no prior knowledge about the environment is known, one might resort to generic graph distributions, such as Erdős-Rényi [9] or Barabási-Albert [1].

[3] We use the publicly available PyTorch PPO implementation and hyperparameters from the following repository by Ilya Kostrikov: `https://github.com/ikostrikov/pytorch-a2c-ppo-acktr-gail`.

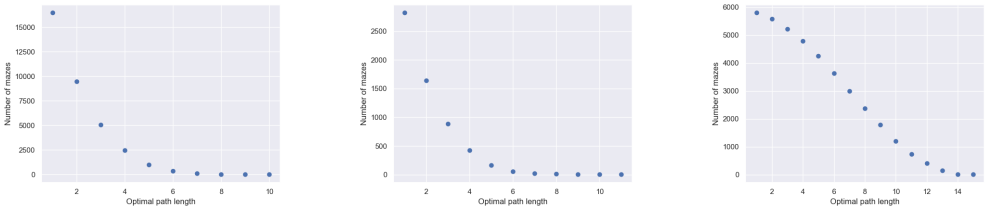

Figure 5: Number of mazes with the optimal path of given length from: 8×8 train dataset (left), 8×8 test dataset (middle) and 16×16 test dataset (right).

Maze The maze environment with randomly generated obstacles from [25]. Observations include a map of the maze (pointing out all obstacles), an indicator of the agent's location, and an indicator of the goal location. Actions correspond to moving in one of the eight principal directions, incurring a reward of -0.01 every move (to encourage shorter solutions), -1 for hitting an obstacle (which terminates the episode), and 1 for hitting the goal (which also terminates the episode).

CartPole The CartPole environment is a classic example of continuous control, first proposed by [3]. The goal is to keep the pole connected by an un-actuated joint to a cart in an upright position. Observations are four-dimensional vectors indicating the cart's position and velocity as well as pole's angle from vertical and pole's velocity at the tip. Actions correspond to staying still, or pushing the engine forwards or backwards. The agent receives a fixed reward of $+1$ for every timestep that the pole remains upright. The episode ends when the pole is more than 15 degrees from the vertical, the cart moves more than 2.4 units from the center or by timing out (at 200 steps), at which point the environment is considered solved.

Acrobot The Acrobot system includes two joints and two links, where the joint between the two links is actuated. Initially, the links are hanging downwards, and the goal is to swing the end of the lower link up to a given height. The environment was first proposed by [24]. The observations—specifying in full the Acrobot's configuration—constitute a six-dimensional vector, and the agent is able to swing the Acrobot using three distinct actions. The agent receives a fixed negative reward of -1 until either timing out (at 500 steps) or swinging the acrobot up, when the episode terminates.

MountainCar The MountainCar environment is an example of a challenging, sparse-reward, continuous-space environment first proposed by [17]. The objective is to make a car reach the top of the mountain, but its engine is too weak to go all the way uphill, so the agent must use gravity to their advantage by first moving in the opposite direction and gathering momentum. Observations are two-dimensional vectors indicating the car's position and velocity. Actions correspond to staying still, or pushing the engine forward or backward. The agent receives a fixed negative reward of -1 until either timing out (at 200 steps) or reaching the top, when the episode terminates.

Freeway Freeway is a game for the Atari 2600, published by Activision in 1981, where the goal is to help the chicken cross the road (by only moving vertically upwards or downwards) while avoiding cars. It is a standard part of the Atari Learning Environment and the OpenAI Gym.

Observations in this environment are the full framebuffer of the Atari console while playing the game, which has been appropriately preprocessed as in [16]. Actions correspond to staying still, moving upwards or downwards. Upon colliding with a car, the chicken will be set back a few lanes, and upon crossing a road, it will be teleported back at the other side to cross the road again (which is also the only time when it receives a positive reward of $+1$). The game automatically times out after a fixed number of transitions.

C Data distribution of mazes

We provide an overview of simple count-based statistics of the maze datasets, stratified by difficulty, in Figure 5. Namely, we can observe that the distribution of 8×8 maze difficulties follows a power-law, whereas the 16×16 maze difficulty counts decay linearly. This poses an additional

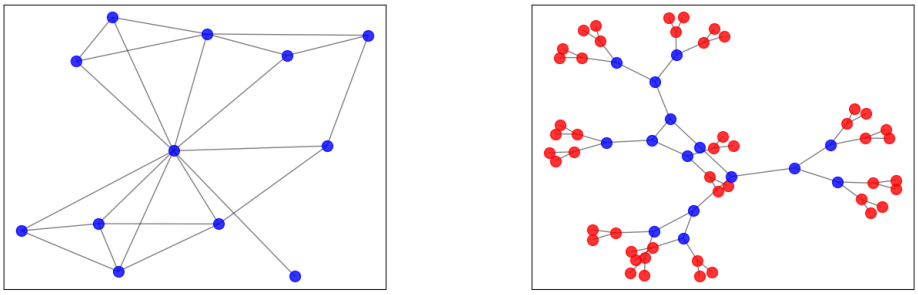

Figure 6: Synthetic graphs constructed for pre-training the GNN executor: Maze (**left**) and CartPole (**right**)

challenge when extrapolating out-of-distribution, as the distributions of the two testing datasets vary drastically—and what worked well for one's performance measure may not necessarily work well for the other.

D Encoder on continual maze

Given the grid-world structure, our encoder for the maze environment is a CNN. We stack three convolutional layers computing 128 features, of kernel size 3×3, each followed by batch normalisation [14] and the ReLU activation. We then aggregate all positions by performing global average pooling[4] [23]. Finally, the aggregated representation is passed to a two-layer MLP with hidden dimension 128 and output dimension 10, with a ReLU activation in-between. The output dimension was chosen to be comparable with VIN [25].

E Synthetic graphs

Figure 6 presents two synthetic graphs used for pretraining the GNN executor. The one for mazes (left) emphasises the terminal node as a node to which all nodes are connected; all other nodes have a maximum of eight neighbours, corresponding to the possible action types.

The graph used for CartPole is, in fact, a binary tree, where red nodes represent nodes with reward 0, and blue nodes have reward 1. This aligns with the idea that going further from the root, which is equivalent with taking repeated left (or right) steps, leads to being more likely to fail the episode.

The CartPole graph is, in fact, also used for pre-training the executor for the other two continuous-observation environments (MountainCar, Acrobot) and for Freeway. Primarily, the similar action space of the environments is a possible supporting argument of the observed transferability. Moreover, MountainCar and Acrobot can be related to a inverted reward graph of CartPole, with more aggressive combinations left/right steps often bringing a higher chance of success.

F Qualitative study

We performed a qualitative study into the node embeddings learnt by the encoder (and transition model), hoping to elucidate the mechanism in which XLVIN organises its plan—please see Appendix F.

At the top row of Figure 7, we (*left*) colour-coded a specific 8×8 test maze by proximity to the goal state, and (*right*) visualised the 3D PCA projections of the "pure-TransE" embeddings of these states (prior to any PPO training), along with the edges induced by its transition model. Such a model merely seeks to accurately organise the data, rather than optimise for returns: hence, it has no trouble organising the data in a grid-like structure.

[4]Note that we could not have performed the usual flatten operation, in order to generalise to 16×16 mazes.

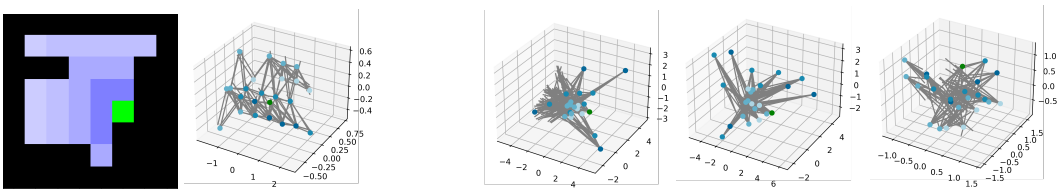

Figure 7: **Left:** A test maze (*left*) and the PCA projection of its TransE state embeddings (*right*), colour-coded by distance to goal (in green). **Right:** PCA projection of the XLVIN state embeddings after passing the first (*left*), second (*middle*), and ninth (*right*) level of the continual maze.

At the bottom row, we visualise how these embeddings and transitions evolve as the agent keeps solving levels; at level one (*left*), the embedder learnt to distinguish all 1-step neighbours of the goal, by putting them on opposite parts of the projection space. At level two (*middle*), the two-step neighbours have also been partitioned away and spread out.

This process does not keep going, because the agent would quickly lose capacity. Instead, by the time it passes level nine (*right*), grid structure emerges again, but now the states become partitioned by proximity: nearer neighbours of the goal are closer to the goal embedding. In a way, the XLVIN agent is learning to reorganise the grid; this time in a way that respects shortest-path proximity.