XLVIN: eXECUTED LATENT VALUE ITERATION NETS

Anonymous authors
Paper under double-blind review

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

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*) (Tamar et al., 2016; Oh et al., 2017; Racanière et al., 2017; Silver et al., 2017; Niu et al., 2018; Guez et al., 2018; 2019). Instead of training explicit environment models and leveraging planning algorithms, such approaches propose inductive biases in the policy function to enable planning to emerge, while training the policy in a model-free manner. A notable example of this line of research are *value iteration networks* (VINs), which observe that the value iteration (VI) algorithm on a grid-world can be understood as a convolution of state values and transition probabilities followed by max-pooling, which inspired the use of a CNN-based VI module (Tamar et al., 2016). *Generalized value iteration networks* (GVINs), based on graph kernels (Yanardag & Vishwanathan, 2015), lift the assumption that the environment is a grid-world and allow planning on irregular discrete state spaces (Niu et al., 2018), such as graphs.

While such models *can* learn to perform VI, they are in no way constrained or explicitly incentivised to do so. Policies including such planning modules might exploit their capacity for different purposes, potentially finding ways to overfit the training data instead of learning how to perform planning. Further, both VINs and GVINs assume discrete state spaces, incurring loss of information for problems with naturally continuous state spaces. Finally, and most importantly, both approaches require the graph specifying 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.

In this paper, we propose the **eXecuted Latent Value Iteration Network** (XLVIN), an implicit planning policy network which embodies the computation of VIN-like models while addressing all of the above issues. As a result, we are able to 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), all the way towards fully continuous-state environments, consistently outperforming or matching baseline models which lack XLVIN's inductive biases.

To achieve this, we have unified recent concepts from several areas of representation learning:

- Using *contrastive self-supervised representation learning*, we are able to meaningfully infer dynamics of the MDP, even when it is not provided. In particular, we leverage the work of Kipf et al. (2020); van der Pol et al. (2020), which uses the TransE model (Bordes et al.,

2013) to embed states and actions into vector spaces, in such a way that the effect of action embeddings onto the state embeddings are consistent with the true environment dynamics.

- By applying recent advances in *graph representation learning* (Battaglia et al., 2018; Bronstein et al., 2017; Hamilton et al., 2017), we designed a message passing architecture (Gilmer et al., 2017) which can traverse our partially-inferred MDP, without imposing strong structural constraints (i.e., our model is not restricted to grid-worlds).
- We better align our planning module with VI by leveraging recent advances from *neural algorithm execution*, which has shown that GNNs can learn polynomial-time graph algorithms (Xu et al., 2019; Veličković et al., 2019; Yan et al., 2020; Georgiev & Lió, 2020; Veličković et al., 2020), by supervising them to structure their problem solving process according to the specific target algorithm. For example, it has been recently shown that GNNs are capable of executing value iteration (Deac et al., 2020) The resulting latent value estimates can then be used as features for a policy network.

While we focus our discussion on VINs and GVINs, which we directly generalise and with which we share key design concepts (like the VI-based differentiable planning module), there are other lines of research that our approach could be linked to. Significant work has been done on representation learning in RL (Givan et al., 2003; Ferns et al., 2004; 2011; Jaderberg et al., 2017; Ha & Schmidhuber, 2018b; Gelada et al., 2019), often exploiting observed state similarities.

Regarding work in planning in latent spaces (Oh et al., 2017; Farquhar et al., 2018; Hafner et al., 2019; van der Pol et al., 2020), *Value Prediction Networks* (Oh et al., 2017) and *TreeQN* (Farquhar et al., 2018) explore some ideas similar to our work, with important differences; they use explicit planning algorithms, while XLVINs do fully implicit planning in the latent space. However, due to the way in which value estimates are represented, the policy network is capable of melding both model-free and model-based cues robustly. Furthermore, while our VI executor provides a representation that aligns with the predictive needs of value iteration, it can also incorporate additional information if it benefits the performance of the model.

2 BACKGROUND

Value iteration Value iteration is a successive approximation method for finding the optimal value function of a discounted *Markov decision processes* (MDPs) as the fixed-point of the so-called Bellman optimality operator (Puterman, 2014). A discounted MDP is a tuple $(\mathcal{S}, \mathcal{A}, R, P, \gamma)$ where $s \in \mathcal{S}$ are *states*, $a \in \mathcal{A}$ are *actions*, $R : \mathcal{S} \times \mathcal{A} \to \mathbb{R}$ is a reward function, $P : \mathcal{S} \times \mathcal{A} \to \text{Dist}(\mathcal{S})$ is a *transition function* such that $\mathbb{P}(s'|s, a)$ is the conditional probability of transitioning to state s' when the agent executes action a in state s, and $\gamma \in [0, 1]$ is a discount factor which trades off between the relevance of immediate and future rewards. In the infinite horizon discounted setting, an agent sequentially chooses actions according to a stationary Markov *policy* $\pi : \mathcal{S} \times \mathcal{A} \to [0, 1]$ such that $\pi(a|s)$ is a conditional probability distribution over actions given a state. The *return* is defined as $G_t = \sum_{k=0}^{\infty} \gamma^k R(a_{t+k}, s_{t+k})$. Value functions $V^\pi(s, a) = \mathbb{E}_\pi[G_t|s_t = s]$ and $Q^\pi(s, a) = \mathbb{E}_\pi[G_t|s_t = s, a_t = a]$ represent the expected return induced by a policy in an MDP when conditioned on a state or state-action pair respectively. In the infinite horizon discounted setting, we know that there exists an optimal stationary Markov policy π^* such that for any policy π it holds that $V^{\pi^*}(s) \geq V^\pi(s)$ for all $s \in \mathcal{S}$. Furthermore, such optimal policy can be deterministic – *greedy* – with respect to the optimal values. Therefore, in order to find a π^* it suffices to find the unique optimal value function $V^\star$ as the fixed-point of the Bellman optimality operator. Value iteration is in fact the instantiation of the method of successive approximation method for finding the fixed-point of a contractive operator. The optimal value function $V^\star$ is such a fixed-point and satisfies the *Bellman optimality equations* (Bellman, 1966):

$$V^\star(s) = \max_{a \in \mathcal{A}} \left(R(s, a) + \gamma \sum_{s' \in \mathcal{S}} P(s'|s, a) V^\star(s') \right) \quad . \tag{1}$$

TransE The TransE (Bordes et al., 2013) loss for embedding objects and relations can be adapted to RL. State embeddings are obtained by an *encoder* $z : \mathcal{S} \to \mathbb{R}^k$ and the effect of an action in a given state is modelled by a translation model $T : \mathbb{R}^k \times \mathcal{A} \to \mathbb{R}^k$. Specifically, $T(z(s), a)$ is a *translation vector* to be added to $z(s)$ in order to obtain an embedding of the resulting state when

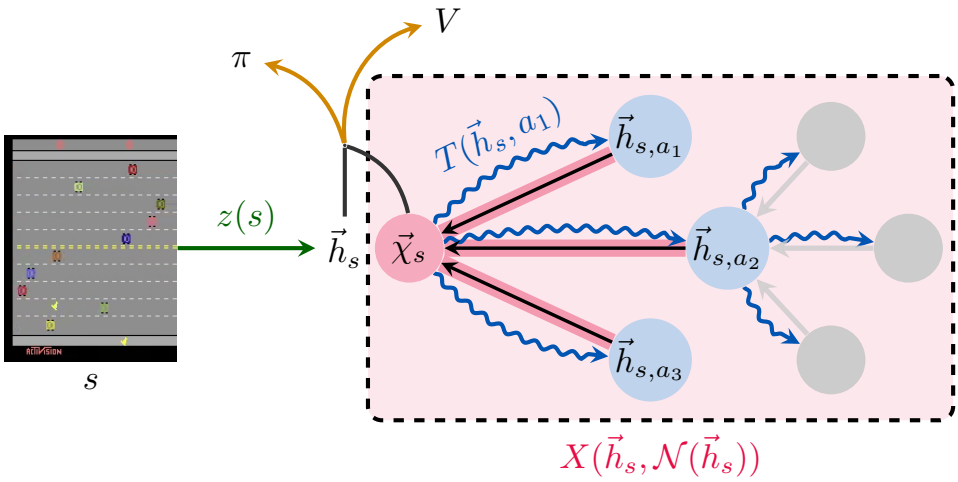

Figure 1: XLVIN model summary. The individual modules are explained (and colour-coded) in Section 3.1, and the dataflow is outlined in Section 3.2.

taking action a in state s.[1] Such embeddings should also be far away from all other states $z(\tilde{s})$. Therefore, the embedding functions are optimized using the following variant of the triplet loss:

$$\mathcal{L} = d(z(s) + T(z(s), a), z(s')) + \max(0, \alpha - d(z(\tilde{s}), z(s'))) \tag{2}$$

where α is the hyperparameter of the hinge loss, (s, a, s') is a transition in a given MDP and $\tilde{s}$ is a negative sample state. Trajectories sampled from the MDP can be used as a training set of *positive* triplets (s, a, s'), whereas negative triplets $(s, a, \tilde{s})$ are obtained by sampling $\tilde{s}$ uniformly at random.

GNNs and algorithmic executors Graph neural networks (GNNs) have been intensively studied as a tool to process graph-structured inputs, and were successfully applied to various of RL tasks (Wang et al., 2018; Klissarov & Precup, 2019; Adhikari et al., 2020). GNN operations are commonly expressed through the *message passing* framework (Gilmer et al., 2017). For each node i in the graph, a set of messages is computed—one message for each neighbouring node j, derived by applying a *message function* M to the relevant node and edge features $(\vec{h}_i, \vec{h}_j, \vec{e}_{ij})$. Incoming messages in a neighbourhood $\mathcal{N}(i)$ are then aggregated through a permutation-invariant operator $\bigoplus$ (such as sum or max), obtaining a summarised message $\vec{m}_i$:

$$\vec{m}_i^{n+1} = \bigoplus_{j \in \mathcal{N}(i)} M_n(\vec{h}_i^n, \vec{h}_j^n, \vec{e}_{ij}^n) \tag{3}$$

The features of node i are then updated through a function U applied to these summarised messages:

$$\vec{h}_i^{n+1} = U_n(\vec{h}_i^n, \vec{m}_i^{n+1}) \tag{4}$$

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 (Veličković et al., 2019). In particular, Xu et al. (2019) establishes *algorithmic alignment* between GNNs and dynamic programming algorithms. Furthermore, Veličković et al. (2019) show that supervising the GNN on the algorithm's intermediate results is highly beneficial for out-of-distribution generalization.

As VI is, in fact, a dynamic programming algorithm, a GNN executor is a suitable choice for learning it. With intermediate supervision, good results on executing VI have emerged on synthetic graphs (Deac et al., 2020), establishing it as a potential planning module in an RL environment.

3 XLVIN Architecture

With all of the above components in place, we can fully specify the computations of the eXecuted Latent Value Iteration Network (XLVIN). Throughout this section, we recommend referring to Figure 1 for a visualisation of the overall model dataflow.

[1] For stochastic MDPs, the resulting embedding should be understood as an *expectation* under $P(s'|s, a)$.

We propose a *policy network*—a function, $\pi_\theta(a|s)$, which for a given state $s \in \mathcal{S}$ specifies a probability distribution of performing each of the actions $a \in \mathcal{A}$ in that state. Here, θ are the neural network parameters, to be optimised using gradient ascent.

3.1 XLVIN MODULES

Encoder The encoder function, $z : \mathcal{S} \to \mathbb{R}^k$, consumes state representations $s \in \mathcal{S}$ and produces flat embeddings, $\vec{h}_s = z(s) \in \mathbb{R}^k$. The design of this component is flexible and may be dependent on the structure present in states. For example, pixel-based environments will necessitate CNN encoders, while environments with flat observations are likely to be amenable to MLP encoders.

Transition The transition function, $T : \mathbb{R}^k \times \mathcal{A} \to \mathbb{R}^k$, models the effects of taking actions, in the *latent* space. Accordingly, it consumes a state embedding $z(s)$ and an action[2] a and produces the appropriate *translation* of the state embedding, to match the embedding of the successor state (in expectation). That is, it is desirable that T satisfies the following property:

$$z(s) + T(z(s), a) \approx \mathbb{E}_{s' \sim P(s'|s,a)} z(s') \tag{5}$$

As T operates in a flat embedding space, it is commonly realised as an MLP.

Executor The executor function, $X : \mathbb{R}^k \times \mathbb{R}^{|\mathcal{A}| \times k} \to \mathbb{R}^k$, generalises the planning modules proposed in VIN (Tamar et al., 2016) and GVIN (Niu et al., 2018). It processes an embedding $\vec{h}_s$ of a given state s, alongside a neighbourhood set $\mathcal{N}(\vec{h}_s)$, which contains (expected) embeddings of states that immediately neighbour s—for example, through taking actions. Hence,

$$\mathcal{N}(\vec{h}_s) \approx \left\{ \mathbb{E}_{s' \sim \mathbb{P}(s'|s,a)} z(s') \right\}_{a \in \mathcal{A}} \tag{6}$$

The executor combines the neighbourhood set features to produce an updated embedding of state s, $\vec{\chi}_s = X(\vec{h}_s, \mathcal{N}(\vec{h}_s))$, which is mindful of the properties and structure of the neighbourhood. Ideally, X would perform operations in the latent space which mimic the one-step behaviour of VI, allowing for the model to meaningfully plan from state s by stacking several layers of X (with K layers allowing for exploring length-K trajectories). Given the relational structure of a state and its neighbours, the executor is commonly realised as a graph neural network (GNN).

Actor & Tail components The actor function, $A : \mathbb{R}^k \times \mathbb{R}^k \to [0, 1]^{|\mathcal{A}|}$ consumes the state embedding $\vec{h}_s$ and the updated state 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 our XLVIN 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) (Schulman et al., 2017), 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.

3.2 OVERVIEW OF XLVIN

Putting all of the above components together, we now present 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:

1. Embed the input state by using the encoder function: $\vec{h}_s = z(s)$.
2. Initialise the depth-0 set of state embeddings to the singleton set, $\mathbb{S}^0 = \{\vec{h}_s\}$, containing only the embedding of s. Accordingly, initialise the set of edges to $\mathbb{E} = \emptyset$.
3. For each graph depth, $k \in [0, K]$:
   (a) Initialise the depth-$(k+1)$ embedding set to $\mathbb{S}^{k+1} = \emptyset$.
   (b) For each state embedding in the previous depth, $\vec{h} \in \mathbb{S}^k$, and action, $a \in \mathcal{A}$:
      i. Compute (expected) neighbour embedding of $\vec{h}$ upon taking action a using the transition model: $\vec{h}' = \vec{h} + T(\vec{h}, a)$.

[2]For discrete action spaces, we use one-hot representations of actions. Extensions of our model to continuous actions, while possible, are deferred to future work.

ii. Attach $\vec{h}'$ to the graph, by $\mathbb{S}^{k+1} = \mathbb{S}^{k+1} \cup \{\vec{h}'\}$, $\mathbb{E} = \mathbb{E} \cup \{(\vec{h}, \vec{h}', a)\}$.

(c) For each state embedding $\vec{h} \in \mathbb{S}^k$, define its neighbourhood as the set of all adjacent embeddings: $\mathcal{N}(\vec{h}) = \{\vec{h}' \mid \exists \alpha \in \mathcal{A}.(\vec{h}, \vec{h}', \alpha) \in \mathbb{E}\}$.

4. Run the execution model over the graph specified by the nodes $\mathbb{S} = \bigcup_{k=0}^{K} \mathbb{S}^k$ and edges $\mathbb{E}$, by repeatedly applying $\vec{h} = X(\vec{h}, \mathcal{N}(\vec{h}))$, for every embedding $\vec{h} \in \mathbb{S}$, for K steps. Denote the updated embedding of s, obtained by the above procedure, as $\vec{\chi}_s$.
5. Finally, use the actor and tail to predict the policy and value functions from the state embedding and updated state embedding of s: $\pi_\theta(s, \cdot) = A(\vec{h}_s, \vec{\chi}_s)$; $V(s) = V(\vec{h}_s, \vec{\chi}_s)$.

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.

3.3 XLVIN Training

As discussed in Section 3.1, our transition function, T, and executor function, X, should ideally respect certain properties (e.g. Equations 5–6). We *pre-train* both of them using established methods in the literature; (Kipf et al., 2020; van der Pol et al., 2020) for TransE and (Deac et al., 2020) for the executor. Due to space constraints, we report detailed pre-training information in Appendix A.

To optimise the neural network parameters θ, we use proximal policy optimisation (PPO)[3] (Schulman et al., 2017). 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 in Equation 2). 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.

4 Experiments

We now proceed to evaluating the proposed XLVIN architecture across three distinct kinds of environments, and a variety of training and testing modes. Our work is centered around the following three research questions, which we will repeatedly refer to throughout this section:

Q1 How do XLVINs compare to (G)VINs when they are applicable (i.e. on fixed, known and discrete MDPs)? Is the graph built within XLVINs as useful as the underlying MDP?

Q2 Do XLVINs effectively **generalise** the VIN model? Do they outperform baseline architectures (without the transition and executor models) across *general* environments?

Q3 Does the XLVINs' environment model help them robustly plan? Are XLVINs amenable to dynamically changing environments (e.g. continual learning) or low-data regimes?

4.1 Experimental setup

We will now categorise our environments into three types—for further details, see Appendix B.

Known MDP In order to compare XLVINs performance against VINs and GVINs (**Q1**) we evaluate on an established environment with a known, fixed and discrete MDP. We use the 8×8 grid-world mazes proposed by Tamar et al. (2016). 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.

[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`.

We utilise this difficulty to 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.

Setting up the task in this way allowed us to evaluate the resilience of the agents to *catastrophic forgetting* (**Q3**)—as the difficulties get harder, the models get further inspired to memorise the training set (see Appendix C for further statistics) We also test *out-of-distribution generalisation* (training on 8×8 mazes, testing on 16×16), allowing for another way to test the agents' robustness (**Q3**).

Given the grid-world structure, our encoder for the maze environment is a three-layer CNN computing 128 latent features and 10 outputs—which we describe in Appendix D. It is important to note that the VIN model of Tamar et al. (2016) performs the aggregation by *slicing* directly the features on the agent's coordinates. This assumes upfront knowledge of where the agent actually is on the map (rather than having to infer it from data) and hence puts VINs in a privileged position. For a possibly fairer comparison (**Q1**), 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.

The transition function is a three-layer MLP with layer normalisation (Ba et al., 2016) 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 Deac et al. (2020). 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 (Ba et al., 2016) applied after every step.

Continuous-space observations The latent-space execution of XLVINs allow us to deploy it in generic environments—especially, continuous-space environments are now supported without having to discretise them. We investigate the performance of XLVIN across these environments (**Q2**), focusing on three classical control environments from the OpenAI Gym: CartPole-v0, Acrobot-v1 and MountainCar-v0. We also address **Q3** by presenting extremely-limited-data scenarios.

The encoder function is now a three-layer MLP with ReLU activations, computing 50 output features and F hidden features, where $F = 64$ for CartPole, $F = 32$ for Acrobot, and $F = 16$ for MountainCar. The same hidden dimension is also used in the transition function, which otherwise matches the one used for mazes.

The executor has been trained from synthetic graphs which are designed to imitate the dynamics of CartPole very crudely—the MDP graphs being binary trees where only certain leaves carry zero reward and are terminal. More details on the graph construction is given in Appendix E. The same trained executor is then deployed across all environments, to demonstrate robustness to synthetic graph construction (**Q3**). In all cases, the XLVIN uses $K = 2$ executor layers.

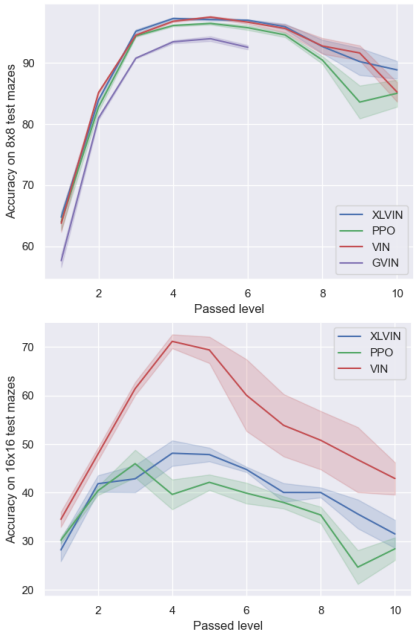

Figure 2: Success rate on 8×8 (**top**) and 16×16 (**bottom**) held-out mazes for XLVIN, PPO, VIN and GVIN, obtained after passing each level of 8×8 train mazes. VIN-mean and VIN-max **failed** to pass difficulty 1.

Before moving on, it is worthy to note that CartPole offers dense and frequent rewards—making it easy for policy gradient methods. We make the task challenging by sampling *only 10 trajectories* at the beginning, and not allowing any further interaction—beyond 100 epochs of training on this dataset. Conversely, Acrobot and MountainCar are both sparse-reward, and known to be challenging for policy gradient. For these environments, we sample 5 trajectories at a time, twenty times during training (for a total of 100 trajectories).

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

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 van der Pol et al. (2020).

Pixel-space Lastly, we investigate how XLVINs perform on high-dimensional pixel-based observations (**Q2**), using the Atari-2600 (Bellemare et al., 2013). 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* (**Q3**) the trained executor from CartPole. We once again evaluate the agents' low-data performance by allowing only 1,000,000 observed transitions (**Q3**). We re-use exactly the environment and encoder from here[4], and run the executor for $K = 2$ layers.

4.2 Results

In 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.

Continual maze The results of our continual maze evaluation are summarised in Figure 2. We observe that, in-distribution, the XLVIN is competitive with all other models (including VIN and GVIN—that fails to pass difficulty 6). It is also the most resilient to level 10, which has only one maze, and is hence very easy to overfit to.

The XLVIN remains competitive with the baseline model 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 mazes.

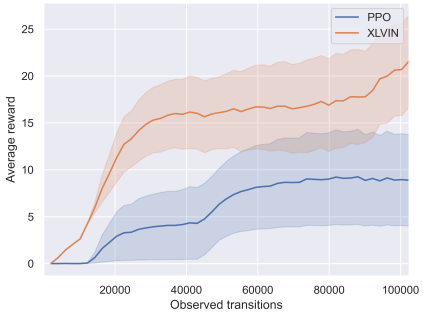

Figure 3: Average reward on Freeway over 1,000,000 processed transitions.

CartPole, Acrobot and MountainCar Results for the continuous-space control environments are provided in Table 1. We find that, for CartPole, the XLVIN model solves the environment from only 10 trajectories, outperforming all the results given in van der Pol et al. (2020) (incl. REINFORCE (Williams, 1992), Autoencoders, World Models (Ha & Schmidhuber, 2018a), DeepMDP (Gelada et al., 2019) and PRAE (van der Pol et al., 2020)), 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. Conversely, the baseline model is unable to get off the ground at all, consistently remaining stuck at the lowest possible score in the environment until timing out.

Freeway Lastly, the average reward of the Freeway agents across the first million transitions can be visualised in Figure 3. The trend can clearly be observed: 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.

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

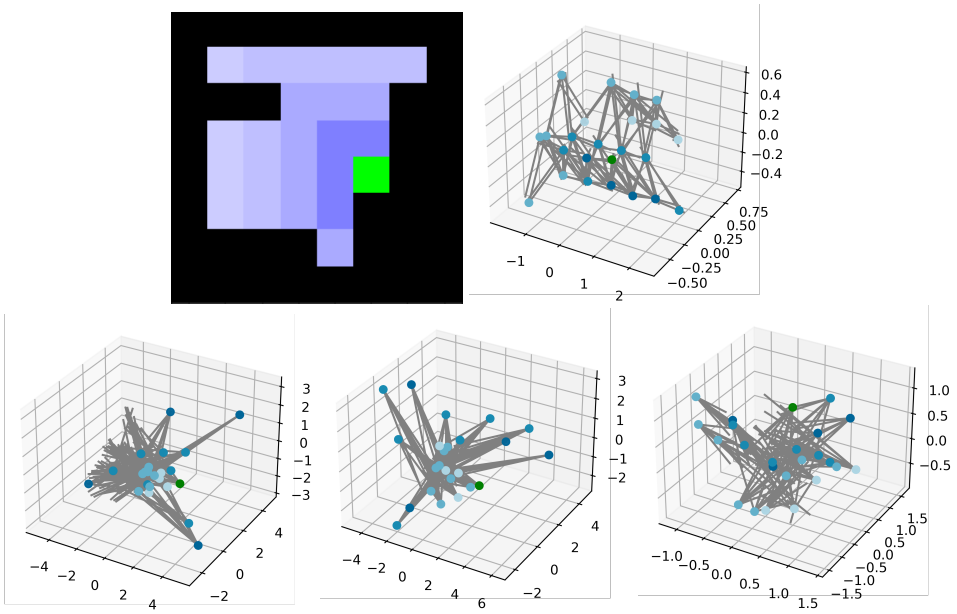

Figure 4: **Top:** A test maze (*left*) and the PCA projection of its TransE state embeddings (*right*), colour-coded by distance to goal (in green). **Bottom:** PCA projection of the XLVIN state embeddings after passing the first (*left*), second (*middle*), and ninth (*right*) level of the continual maze.

4.3 Qualitative results

We conclude with 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 (and hence provide further insight on **Q3**).

At the top row of Figure 4, 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.

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.

5 Conclusions

We presented eXecuted Latent Value Iteration Networks (XLVINs), combining recent progress in self-supervised contrastive learning, graph representation learning and neural algorithm execution to generalise Value Iteration Networks to MDPs which are irregular, continuous, not known, or any of the above. Our results demonstrate that XLVINs are capable of matching or outperforming appropriately constructed baselines, often at low-data or out-of-distribution data regimes. Further, the learnt executors are robust and transferable across environments, despite the fact that they are trained on synthetic graphs that only roughly align with the underlying MLP. XLVINs represent, to the best of our knowledge, one of the first times neural algorithmic executors have been used for implicit planning, and we believe that this is a very promising direction for future RL research.

REFERENCES

Ashutosh Adhikari, Xingdi Yuan, Marc-Alexandre Côté, Mikuláš Zelinka, Marc-Antoine Rondeau, Romain Laroche, Pascal Poupart, Jian Tang, Adam Trischler, and William L Hamilton. Learning dynamic knowledge graphs to generalize on text-based games. *arXiv preprint arXiv:2002.09127*, 2020.

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

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

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.

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.

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.

Richard Bellman. Dynamic programming. *Science*, 153(3731):34–37, 1966.

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

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.

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

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

Gregory Farquhar, Tim Rocktäschel, Maximilian Igl, and Shimon Whiteson. Treeqn and atreec: Differentiable tree-structured models for deep reinforcement learning. In *ICLR*, 2018.

Norm Ferns, Prakash Panangaden, and Doina Precup. Metrics for finite markov decision processes. In *AAAI*, pp. 950–951, 2004.

Norm Ferns, Prakash Panangaden, and Doina Precup. Bisimulation metrics for continuous markov decision processes. *SIAM J. Comput.*, 40(6):1662–1714, 2011.

Carles Gelada, Saurabh Kumar, Jacob Buckman, Ofir Nachum, and Marc G. Bellemare. Deepmdp: Learning continuous latent space models for representation learning. In *ICML*, volume 97, pp. 2170–2179, 2019.

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

Justin Gilmer, Samuel S Schoenholz, Patrick F Riley, Oriol Vinyals, and George E Dahl. Neural message passing for quantum chemistry. *arXiv preprint arXiv:1704.01212*, 2017.

R. Givan, T. Dean, and M. Greig. Equivalence notions and model minimization in Markov decision processes. *Artificial Intelligence Journal*, 147:163–224, 2003.

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, pp. 1817–1826. PMLR, 2018.

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, pp. 2464–2473, 2019.

David Ha and Jürgen Schmidhuber. World models. *arXiv preprint arXiv:1803.10122*, 2018a.

David Ha and Jürgen Schmidhuber. Recurrent world models facilitate policy evolution. In *NeurIPS*, pp. 2455–2467, 2018b.

Danijar Hafner, Timothy P. Lillicrap, Ian Fischer, Ruben Villegas, David Ha, Honglak Lee, and James Davidson. Learning latent dynamics for planning from pixels. In *ICML*, volume 97, pp. 2555–2565, 2019.

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

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

Max Jaderberg, Volodymyr Mnih, Wojciech Marian Czarnecki, Tom Schaul, Joel Z. Leibo, David Silver, and Koray Kavukcuoglu. Reinforcement learning with unsupervised auxiliary tasks. In *ICLR*, 2017.

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

Martin Klissarov and Doina Precup. Graph convolutional networks as reward shaping functions. In *ICLR Workshop on Representation Learning on Graphs and Manifolds*, 2019.

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.

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

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

Junhyuk Oh, Satinder Singh, and Honglak Lee. Value prediction network. In *NIPS*, pp. 6118–6128, 2017.

Martin L Puterman. *Markov decision processes: discrete stochastic dynamic programming*. John Wiley & Sons, 2014.

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*, pp. 5690–5701, 2017.

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

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, pp. 3191–3199, 2017.

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

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

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 (eds.), *NIPS*, pp. 2146–2154, 2016.

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

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

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

Tingwu Wang, Renjie Liao, Jimmy Ba, and Sanja Fidler. Nervenet: Learning structured policy with graph neural networks. In *International Conference on Learning Representations*, 2018.

Ronald J Williams. Simple statistical gradient-following algorithms for connectionist reinforcement learning. *Machine learning*, 8(3-4):229–256, 1992.

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.

Yujun Yan, Kevin Swersky, Danai Koutra, Parthasarathy Ranganathan, and Milad Hashemi. Neural Execution Engines, 2020. URL `https://openreview.net/forum?id=rJg7BA4YDr`.

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

A PRE-TRAINING OF TRANSITION AND EXECUTOR FUNCTIONS

The transition condition of Equation 5 perfectly aligns with the TransE loss (Bordes et al., 2013), hence we pre-train T by optimising the loss function of Equation 2, using transitions sampled in the environment using a random policy—not unlike the prior work by Kipf et al. (2020); van der Pol et al. (2020) 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 Deac et al. (2020). 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 Deac et al. (2020), we generate synthetic discrete MDPs that align with the target MDP as much as possible[5]—finding that useful transfer still occurs, echoing the findings of Deac et al. (2020).

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).

B ENVIRONMENTS UNDER STUDY

We provide a visual overview of all five environments considered in Figure 5.

[5]If no prior knowledge about the environment is known, one might resort to generic graph distributions, such as Erdős-Rényi (Erdős & Rényi, 1960) or Barabási-Albert (Albert & Barabási, 2002).

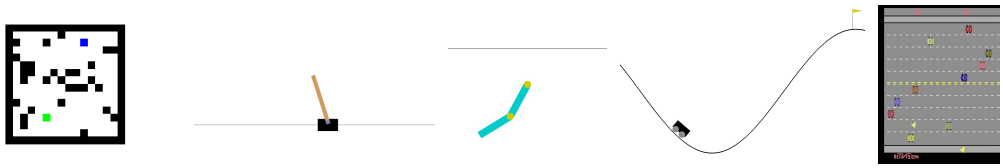

Figure 5: The five environments considered within our evaluation: 8×8 and 16×16 mazes (Tamar et al., 2016) (known grid-like MDP), continuous control environments (CartPole-v0, Acrobot-v1, MountainCar-v0) and Atari Freeway (pixel-based space).

Maze The maze environment with randomly generated obstacles from Tamar et al. (2016). 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 Barto et al. (1983). 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 Sutton (1996). 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 Moore (1990). 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 Mnih et al. (2015). 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 6. Namely, we can observe that the distribution of 8×8 maze difficulties follows a

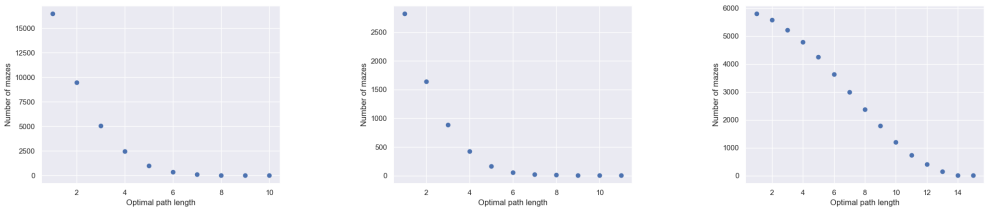

Figure 6: Number of mazes with the optimal path of given length from: 8×8 train dataset (left), 8×8 test dataset (middle) and 16×16 test dataset (right).

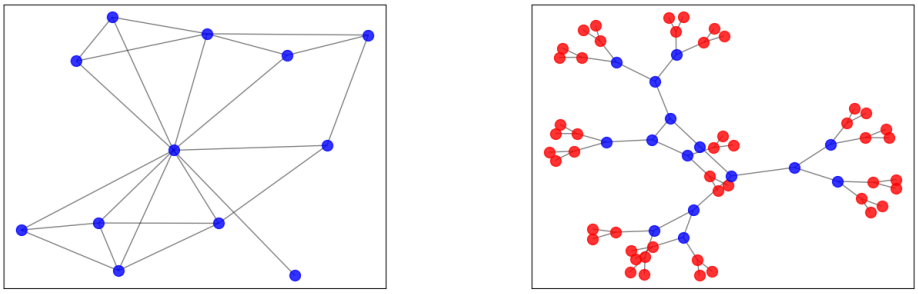

Figure 7: Synthetic graphs constructed for pre-training the GNN executor: Maze (**left**) and CartPole (**right**)

power-law, whereas the 16×16 maze difficulty counts decay linearly. This poses an additional 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 (Ioffe & Szegedy, 2015) and the ReLU activation. We then aggregate all positions by performing global average pooling[6] (Springenberg et al., 2014). 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 (Tamar et al., 2016).

E SYNTHETIC GRAPHS

Figure 7 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. More-

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

over, 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.