

Université de Montréal

**Reasoning with Structure : Graph Neural Networks
Algorithms and Applications**

par

Andreea-Ioana Deac

Département d'informatique et de recherche opérationnelle
Faculté des arts et des sciences

Thèse présentée en vue de l'obtention du grade de
Philosophiæ Doctor (Ph.D.)
en informatique

29 août 2023

Université de Montréal

Faculté des arts et des sciences

Cette thèse intitulée

**Reasoning with Structure : Graph Neural
Networks Algorithms and Applications**

présentée par

Andreea-Ioana Deac

a été évaluée par un jury composé des personnes suivantes :

Pierre-Luc Bacon

(président-rapporteur)

Jian Tang

(directeur de recherche)

Sarath Chandar Anbil Parthipan

(membre du jury)

Xavier Bresson

(examineur externe)

(représentant du doyen de la FESP)

Résumé

L'avènement de l'apprentissage profond a permis à l'apprentissage automatique d'exceller dans le traitement d'images et de texte. Donnant lieu à de nombreux succès dans les domaines d'applications tels que la vision par ordinateur ou le traitement du langage naturel. Cependant, il demeure un grand nombre de problèmes d'intérêt dont les données d'entrées ne peuvent être exprimées sous l'un de ces deux formats sans perte d'informations potentiellement cruciales pour leur résolution. C'est dans l'optique de répondre à ce besoin qu'a été développée la branche de l'apprentissage profond géométrique (GDL), qui s'intéresse aux espaces de représentations plus générales, mieux adaptées aux données dont la **structure** sous-jacente ne correspond pas au format de chaîne de caractères unidimensionnel (texte) ou bidimensionnel (images).

Dans cette thèse, nous nous concentrerons plus particulièrement sur les **graphes**. Les graphes sont des structures de données omniprésentes, sous-jacentes à pratiquement toutes les tâches d'intérêt, y compris celles portant sur les données naturelles (par exemple les molécules), les relations entre entités (par exemple les réseaux de transport et les placements de puces), ou encore la liaison de concepts dans les processus de **raisonnement** (par exemple les algorithmes et autres constructions théoriques).

Alors que les architectures modernes de réseaux de neurones de graphes (GNNs) dits expressifs peuvent obtenir des résultats impressionnants sur des benchmarks comme susmentionnés, leur application pratique est toujours en proie à de nombreux problèmes et lacunes, que cette thèse abordera. Les considérations issues de ces **applications** préparerons le terrain pour les chapitres suivants, qui se concentreront sur la résolution des limites des réseaux de neurones de graphes en proposant de nouveaux **algorithmes** d'apprentissage de graphes. Tout d'abord, nous porterons notre attention sur l'amélioration des réseaux de neurones de graphes pour les données qui nécessitent des interactions à longue portée, en construisant des modèles généraux pour compléter leur graphe de calcul. Viennent ensuite les réseaux de neurones de graphes pour les données hétérophiles, où les arêtes ont tendance à connecter des nœuds de différentes classes ; dans ce cas, nous proposerons une modification

particulière du graphe de calcul destinée à améliorer l'homophilie atténuée le problème. Dans un troisième temps, nous tirerons parti d'une caractéristique avantageuse des réseaux de neurones de graphes - leur alignement avec la programmation dynamique. Elle permet aux réseaux de neurones de graphes d'exécuter des algorithmes, sur la base desquels nous proposons une nouvelle classe de planificateurs implicites pour la prise de décision. Enfin, nous capitalisons sur l'utilité de l'apprentissage profond géométrique dans l'apprentissage par renforcement et l'étendrons au-delà des GNNs, en tirant parti des réseaux de neurones à rotation équivariante dans les agents basés sur des modèles.

Mots-clés : Apprentissage profond, Apprentissage de la représentation de graphes, Oversquashing, Hétérophilie, Raisonnement algorithmique, Apprentissage par renforcement, Interactions moléculaires.

Abstract

Since the deep learning revolution, machine learning has excelled at tasks based on images and text, many successes being possible under the umbrella of the computer vision and natural language processing fields. However, much remains that cannot be expressed in these forms without losing information. For these cases, the field of geometric deep learning was developed, covering the space of more general representations, for data whose underlying **structure** doesn't match the single-dimensional string of characters (text) or 2-D shape (images) format.

In this thesis, I will particularly focus on **graphs**. Graphs are ubiquitous data structures underlying virtually all tasks of interest, including natural inputs such as molecules, entity relations for example transportation networks and chip placements, or concept linking in **reasoning** processes, including algorithms and other theoretical constructs.

While modern expressive graph neural network architectures can achieve impressive results on benchmarks like these, their practical application is still plagued with many issues and shortcomings, which this thesis will address. The considerations from these **applications** will set the scene for the following chapters, which focus on tackling the limitations of graph neural networks by proposing new graph learning **algorithms**. Firstly, I focus on improving graph neural networks for data that requires long-range interactions by building general templates to complement their computation graph. This is followed by graph neural networks for heterophilic data, where the edges tend to connect nodes from different classes; in this case, a specialised modification of the computation graph meant to improve homophily alleviates the problem. In the third article, I leverage a strength of graph neural networks – their alignment with dynamic programming. This enables graph neural networks to execute algorithms, based on which I propose a new class of implicit planners for decision making. Lastly, I capitalise on the utility of geometric deep learning in reinforcement learning and extend it beyond GNNs, leveraging rotation-equivariant neural networks in model-based agents.

Keywords: Deep learning, Graph representation learning, Oversquashing, Heterophily, Algorithmic reasoning, Reinforcement learning, Molecular interactions.

Contents

Résumé	5
Abstract	7
List of Tables	13
List of Figures	15
List of acronyms and abbreviations	19
Remerciements	21
Introduction	23
Chapter 1. Background and Motivation	27
1.1. Graph Representation Learning	27
1.1.1. Graph neural networks	28
1.1.2. Expressive power of graph neural networks	29
1.2. Neural Algorithmic Reasoning	32
1.3. Geometric Deep Learning	33
1.3.1. Groups and Representations	34
1.3.2. Equivariance and Invariance	34
1.4. Reinforcement Learning	36
1.4.1. Model-based RL	39
1.5. Graph Neural Networks for Real-world applications	41
1.5.1. Graph task types	42
1.5.2. Motivating modulators: Applied examples	43
1.5.2.1. Large-scale learning on graphs	43
1.5.2.2. Molecular interactions	44
1.6. Summary	48

Chapter 2. Graph Neural Networks for Long-Range Interaction: Prologue to the first article	51
2.1. Article Details	51
2.2. Context	51
2.3. Modulator	52
2.4. Contributions and Research Impact	52
Chapter 3. Expander Graph Propagation	53
3.1. Introduction	53
3.2. Related work	55
3.3. Theoretical background	57
3.4. Local structure of the Cayley graphs, and the utility of negative curvature....	60
3.5. Expander graph propagation	63
3.6. Empirical evaluation	64
3.7. Conclusion	66
Chapter 4. Graph Neural Networks for Heterophilic data: Prologue to the second article	69
4.1. Article Details	69
4.2. Context	69
4.3. Modulator	70
4.4. Contributions and Research Impact	70
Chapter 5. Evolving Computation Graphs	71
5.1. Introduction	71
5.2. Background	74
5.3. Evolving Computation Graphs	77
5.4. Experiments	78
5.4.1. Qualitative studies	80

5.5.	Related work	81
5.6.	Limitations and further work.....	83
5.7.	Conclusions.....	83
Chapter 6. Graph Neural Networks for Reinforcement Learning: Prologue to the third article		85
6.1.	Article Details	85
6.2.	Context.....	85
6.3.	Modulator	87
6.4.	Contributions and Research Impact	87
Chapter 7. Neural Algorithmic Reasoners are Implicit Planners		89
7.1.	Introduction	89
7.2.	Background and related work	91
7.3.	XLVIN Architecture	95
7.3.1.	XLVIN modules.....	95
7.3.2.	XLVIN Training	97
7.4.	Experiments	98
7.4.1.	Experimental setup	98
7.4.2.	Results	99
7.4.3.	Qualitative results	101
7.5.	Conclusions.....	104
Chapter 8. Geometric Deep Learning for Reinforcement Learning: Prologue to the fourth article		105
8.1.	Article Details	105
8.2.	Context.....	105
8.3.	Modulator	106
8.4.	Contributions and Research Impact	106
Chapter 9. Equivariant MuZero.....		107

9.1. Introduction	107
9.2. Background.....	108
9.3. Equivariant MuZero	111
9.4. Experiments and results	115
9.5. Limitations and future work	117
9.6. Conclusions.....	117
Chapter 10. General Conclusions.....	119
References	121
Appendix A. Appendix of Chapter 2.....	139
A.1. Proof of Proposition 3.4.1	139
A.2. Proof of Theorem 3.4.3.....	140
A.3. Cayley graph at infinity is quasi-isometric to a tree	142
A.4. Mixing time properties of expander graphs	143
A.5. Additional experimental details and ablations.....	144
Appendix B. Appendix of Chapter 4.....	147
B.1. Training information	147
Appendix C. Appendix of Chapter 6.....	151
C.1. Alternate rendition of XLVIN dataflow	151
C.2. Additional description of the Execute function.....	151
C.3. Environments under study.....	152
C.4. Additional CartPole results.....	155
C.5. Synthetic graphs	155
C.6. Maze results.....	156
C.7. Pixel-based world models.....	157
C.8. Compute details.....	158
C.9. Potential societal impact	158

List of Tables

1	Summary of notation introduced in this section.....	28
2	A summary of principal approaches to handling global context in graph representation learning (Section 3.2). “(✓)” indicates that a criterion <i>may</i> be satisfied, depending on the method’s tradeoffs. Our proposal, the expander graph propagation (EGP) method, satisfies all four criteria.....	56
3	Comparative evaluation performance on the four datasets studied. Our baseline model is a GIN [1], using exactly the same implementation as in [2]. See Appendix A.5 for ablations.....	66
4	ECG performance on datasets proposed in [3]. We report accuracy for <code>roman-empire</code> and <code>amazon-ratings</code> and ROC AUC for <code>minesweeper</code> , <code>tolokers</code> , and <code>questions</code>	80
5	Statistics of the original heterophilous graphs and of the evolutionary computation graph obtained from MLP and BGRL.....	82
6	Our models are trained to execute the value iteration algorithm by predicting the values of each MDP state. We show the mean-squared error of the predicted values against ground-truth. Taking the action with the highest expected predicted value using the Bellman equation, we can also predict policies. We then compute the policy accuracy with respect to the ground-truth optimal policy. We test different GNN architectures, noting the MPNN is generally robust to aggregator type, while the less aligned version using attention under-performs in terms of policy accuracy.....	86
7	Mean scores for low-data <code>CartPole-v0</code> , <code>Acrobot-v1</code> , <code>MountainCar-v0</code> and <code>LunarLander-v2</code> , averaged over 100 episodes and five seeds.....	100
8	Statistics of the three graph classification datasets studied in our evaluation....	144

9	<p>Comparative ablation performance of various propagation templates on <code>ogbg-molhiv</code>, <code>ogbg-molpcba</code> and <code>ogbg-ppa</code>. Our baseline model is a GIN [1], using exactly the same implementation as in [2]. All models have <i>exactly</i> the same number of parameters—we only modify the connectivity in certain layers depending on the scheme. N.B. The fully-connected graph, used in the FA approach [4] can be seen as a dense expander graph, i.e. a special case of EGP. 'OOT' indicates that the method failed to approach baseline performance within five days of training time (while not converging within this time), and 'OOM' indicates out-of-memory (on a V100 GPU). 145</p>
10	<p>The best-performing hyperparameters for each GNN propagation rule in our experiments. The only experiment where the baseline model outperforms ECG is the SAGE propagation layer on <code>amazon-ratings</code>; hence, the hyperparameters k and p_{de} are irrelevant. 148</p>
11	<p>Detailed breakdown of model performance on the datasets proposed by Platonov et al. [3]. ResNet, GCN, SAGE, GAT-sep and GT-sep are the baselines, while all the other models are variants of ECG. Red marks the best performance on each dataset for each of the considered GNN architectures and the corresponding ECGs. Accuracy is reported for <code>roman-empire</code> and <code>amazon-ratings</code>, and ROC AUC is reported for <code>minesweeper</code>, <code>tolokers</code>, and <code>questions</code>. 149</p>
12	<p>Mean scores for CartPole-v0 after training, averaged over 100 episodes and five seeds. Baseline CartPole results reprinted from [5]. 155</p>

List of Figures

1	High-level overview of the proposed framework. The encoder-decoder framework is a general machine learning paradigm, with encode-process-decode being commonly used as a way to process graph-structured inputs. Our proposal instead focuses on the graph \mathcal{G} underlying the task. For any task, we need to define the functions of 1) the modulator, whose purpose is to build the graph \mathcal{G} and 2) the processor, which derives the auxiliary measures to be used by the decoder.	24
2	Diagram of a <i>homophilic</i> graph (left) and a <i>heterophilic</i> graph (right). Node colours represent classes, and edges are coloured blue if they connect nodes of the same class, and red otherwise. Homophily (edges mostly blue) often arises in the presence of <i>community structures</i> ; for example, when classifying documents in a citation network, documents with similar topics will tend to cite one another more frequently than documents with different topics. Heterophily (edges mostly red), instead, can arise if the graph is misaligned to the task, or if the network has adversarial actors; for example, in a fraud detection setting, the fraudulent actors are very rare, and will focus on creating many more connections with legitimate nodes than with other fraudsters, to avoid detection.	31
3	Picture from Xu et al. [6] indicating the alignment between Graph Neural Networks and Bellman-Ford, a dynamic programming algorithm for finding shortest paths.	32
4	A frame from the Chaser environment in the ProcGen suite [7] and its 90°, 180° and 270° rotations.	34
5	Commutative diagram displaying \mathfrak{G} -equivariance of function f , as described in Equation 1.3.1 – general case to the left, then for permutation and rotation respectively.	35
6	Left portrays model-free agents, while right shows a general model-based agent. .	39
7	Picture from [8] portraying how MuZero plans (A), acts (B) and trains (C).	41
8	The high-level overview of molecular interaction tasks – we form a bipartite graph with the two entities or types of entities of interest, allowing for all pairwise	

	interactions: atoms of drug pairs, drug-protein and amino acids in antibody-antigen complex.....	45
9	A high-level overview of our drug-drug side-effect prediction model. The next-level features of atom i of drug x , $^{(d_x)}h_i^t$, are derived by combining its <i>input features</i> , $^{(d_x)}h_i^{t-1}$, its <i>inner message</i> , $^{(d_x)}m_i^t$, computed using message passing, and its <i>outer message</i> , $^{(d_x)}n_i^t$, computed using co-attention over the second drug, d_y	45
10	The Para-EPMP architecture (left) and the Epi-EPMP multitask architecture (right). The output feature dimension for each layer is the first term in the bracket. For the convolutional layers, the second term is the kernel size.....	47
11	Qualitative example on the 4jr9 pdb. We plot on the residuals the binding probability as increasing intensity colors: blue for the antibody and red for the antigen. The left figure shows the results of the $E(n)$ -EPMP on the residual graph, while the right sides displays the predictions of the surface-based method.	48
12	Left: The Cayley graph of $SL(2, \mathbb{Z}_3)$, constructed using our method. It has $ V = 24$ nodes and it is 4-regular (implying $ E = 2 V $), hence it is sparse. Despite its sparsity, it is highly interconnected: any node is reachable from any other node by no more than 4 hops. Hence, it can serve as a strong “template” for globally propagating node features with a GNN. Right: The Cayley graph of $SL(2, \mathbb{Z}_5)$, constructed in an analogous way (with $ V = 120$ nodes). A 2-hop neighbourhood of one node (in red) is highlighted, demonstrating its tree-like local structure.....	55
13	A simplified illustration of Evolving Computation Graphs. Step 1: nodes in a graph, G , are embedded using a pre-trained weak classifier. Step 2: Based on these embeddings, a nearest-neighbour graph, G^{EGC} , is generated. This graph is likely to have improved propagation and homophily properties (illustrated by similar colours between neighbouring nodes). Step 3: Message passing is performed, both in the original and in the ECG graph, to update node representations.....	72
14	For roman-empire , we use a random GCN layer to obtain node embeddings based on the original graph G (left) or from the complementary graph G^{EGC} (right). The colours correspond to the ground-truth labels of the nodes.....	81
15	Correspondences between value iteration and graph convolution.....	86

16	From [9]: average rewards over time for CNAP (red) and PPO baseline (blue), in Swimmer (action dimension=2) and Halfcheetah (action dimension=6), using different sampling methods. In Swimmer, CNAP with sampling methods were compared with the original version by expanding all actions (green). In (a)(e), the actions were sampled using Gaussian distribution with mean= $N/2$ and std= $N/4$, where N was the number of action bins used to discretize the continuous action space. In (b)(f), two linear layers were used to learn the mean and std, respectively. In (c)(g), the Policy layer was reused in sampling actions to expand. In (d)(h), a separate linear layer was used to learn the optimal neighbor sampling distribution.	88
17	XLVIN model summary. Its modules are explained (and colour-coded) in Section 7.3.1.	93
18	Average clipped reward on Freeway, Alien, Enduro and H.E.R.O. over 1,000,000 transitions and ten seeds.	100
19	Top: A test maze (<i>left</i>) and the PCA projection of its TransE state embeddings (<i>right</i>), colour-coded by distance to goal (in green). Bottom: PCA projection of the XLVIN state embeddings after passing the first (<i>left</i>), second (<i>middle</i>), and ninth (<i>right</i>) level of the continual maze.	102
20	Left: V^* predictibility–Coefficient of determination from linearly regressing on the state embeddings obtained from the encoder (green) and from the executor (red). Right: Algorithmic bottleneck–Policy accuracy from introducing Gaussian noise in the scalar input fed into VI (red) and in the embeddings fed into the XLVIN executor (green).	103
21	Commutative diagram of symmetries in RL. State transitions due to an action a are back-to-front, transformations due to a symmetry \mathbf{g} are left-to-right, state encoding and decoding by the model is bottom-to-top.	110
22	Architecture of Equivariant MuZero, where h, g are encoders, τ is the transition model, ρ is the reward model, v is the value model and π is the policy predictor. Each colour represents an element of the C_4 group $\{\mathbf{I}, \mathbf{R}_{90^\circ}, \mathbf{R}_{180^\circ}, \mathbf{R}_{270^\circ}\}$ applied to the input (observation and action).	111
23	Results on procedurally-generated MiniPacman (top) and Chaser from ProcGen (bottom).	116

24	XLVIN model summary with compact dataflow. The individual modules are explained (and colour-coded) in Section 7.3.1, and the dataflow is outlined in Algorithm 3.....	151
25	The eight environments considered within our evaluation: continuous control environments (CartPole-v0, Acrobot-v1, MountainCar-v0, LunarLander-v2) and pixel-based environments (Atari Freeway, Alien, Enduro and H.E.R.O.).....	153
26	Synthetic graphs constructed for pre-training the GNN executor: random deterministic (20 states, 8 actions) (left) and CartPole (right).....	155
27	Success rate on 8×8 (left) and on 16×16 (right) held-out mazes obtained after passing each level of their respective train mazes. Cut-off curves imply failure to pass a difficulty level.....	156
28	Freeway frames (above) and reconstructions (below) using a VAE-style world model.....	157

List of acronyms and abbreviations

iid	Independent and Identically-Distributed
AI	Artificial Intelligence
ML	Machine Learning
DL	Deep Learning
NN	Neural Network
SGD	Stochastic Gradient Descent
MLP	Multi-layer Perceptron
ReLU	Rectified Linear Unit
GNN	Graph Neural Network
MPNN	Message Passing Neural Network
GCN	Graph Convolutional Network

GAT Graph Attention Network

RL Reinforcement Learning

VI Value Iteration

DP Dynamic Programming

Remerciements

This journey proved to me that the saying “it takes a village to raise a child” also applies to raising a Doctor. This thesis would have definitely not been possible without the people mentioned below, and many others besides them, to whom I will always be grateful.

There are no words to properly thank my parents, but I’ll try. Thank you for a support that was beyond unconditional. Thank you for my siblings, and thank you to my siblings, my constant joy. Thank you to my grandparents, who, through their love, reminded me many times of my values. Sau: nu sunt cuvinte să le mulțumesc părinților mei cum se cuvine, dar o să încerc. Aș zice mulțumesc pentru susținerea necondiționată, dar a fost mai mult de atât. Mulțumesc pentru frații mei și mulțumesc fraților mei, sursa mea constantă de bucurie. Mulțumesc lui mamaie, lui tataie și bunicilor mei, care, prin dragostea lor, mi-au amintit de multe ori de valorile care mă ghidează.

I’d like to thank Jian Tang, my PhD supervisor. First, for offering me a chance to do a research internship after the third year of my undergrad, which sparked the light that led to the rest of the journey. His close guidance during that period (including a mathematical proof on a whiteboard of why my architecture wouldn’t work, something I found incredible at the time for the field of machine learning!) and the early years of my PhD, together with the freedom he offered me afterwards, are something I will always appreciate.

In the chapter of academic role-models, I want to thank Pietro Liò. His support during my Masters and all the years afterwards was instrumental. Giving me the opportunity to co-supervise students, give talks and meet great people showed me kindness as a super-power in academia.

In the same chapter, many thanks to Doina Precup. The internship with her was an amazing, steep learning curve, that allowed me to wander from the drug discovery field I was familiar with to the more general problem of learning, encouraging me to think much deeper about my research. Her ability to listen and build on ideas ranging from the their earliest to their most mature phases inspires me.

As you can probably already tell, internships have greatly contributed to my education, possibly in almost an equal amount to my formal degrees. I owe a big thanks to George Papamakarios and Théophane Weber, for giving me the chance to explore one more time,

learning about symmetries and model-based RL. Also thank you to Marc Brockschmidt, for letting me exploit – doing a project on drug-discovery with his guidance at such an early time in my PhD and during COVID, showed me good standards of research and led me out of the local maxima I had reached. Also due to internships, I can thank my hosts at my three internships at Google – Carlos, Pratik, Doudou and Daniel, for knowing I can code my way to (almost) anything (given enough time and space). However, the most important part of these internships was the people I got to meet. Thank you to Miltos Allamanis, Charles Blundell, Ankit Anand, Kory Mathewson, Alvaro Sanchez-Gonzalez, Pol Moreno, Olivier Tieleman, Hado van Hasselt, Jessica Hamrick, Peter Battaglia, Sébastien Racanière, Ioana Bica, Anirudh Goyal and many others from DeepMind Montreal, DeepMind London and Microsoft Research Cambridge, for making me a better researcher. Each of them has had a lasting effect on my development, from research chats, providing mentorship, brainstorming to longer collaborations.

And it is the collaborations that have really been the crux of this PhD. I have been really lucky to work along side Mladen Nikolić, Ognjen Milinković and Pierre-Luc Bacon for XLVIN, Marc Lackenby for our work with expanders and for the follow-up on oversquashing, where I was impressed by Francesco di Giovanni, Konstantin Rusch, Michael Bronstein and Siddhartha Mishra, the amazing DeepMind teams for the OGB-LSC competition and the generalist algorithmic learner, as well as the impressive team working on our Nature paper, lead by Marinka Žitnik.

I am also grateful for the colleagues and friends I have gained along the way. People from Jian’s lab and Mila, Pietro’s lab and Cambridge, students I worked with and LoG organisers have provided motivation and support in more ways than they’ll know. Among many, thank you Sophie Xhonneux, Yu He, Zhaocheng Zhu, Meng Qu, Martin Klissarov, Dobrik Georgiev, Iulia Duță, Bianca Dumitrașcu and Lorenzo Giusti.

It must also be said that I got here because I’ve been blessed with amazing teachers and professors. For having an influence on me that started before this PhD and will last much after, I particularly need to thank Ștefan Smarandoiu and David Chisnall.

Also stretching much beyond these four years, I’d like to thank my friends. Ștefan, Mara, Tudor, Ștefi, Cîrceag, Deaconu, Anca and Alex, Mamal, Nithin, Dragoș, Radu and Sisi, thank you!

And, as he was often the foundation for everything I built over the duration of this PhD, it’s only because I want to finish this chapter by thanking him that he’s last. Thank you my fiancé-turned-husband-in-the-meantime, my rock, Petar. You make me a better human in every way, everyday. Hvala nije dovoljno.

Introduction

Deep learning focuses on learning functions that are high-dimensional in nature, making it vulnerable to the curse of dimensionality. However, real-world tasks are rarely adversarial: they usually come with geometric regularities stemming from the underlying low-dimensional processes and structure of the world. These regularities can then be used to make high-dimensional problems more tractable. Accordingly, restricting the space of functions of interest to deep learning to the functions that incorporate the geometric regularities of the real world can be beneficial to solving downstream tasks. For example, geometric regularities such as translation equivariance in CNNs [10] and time warping invariance in RNNs [11] for images and text respectively have been successfully exploited.

For the purpose of this thesis, we will be looking at two types of regularities that have been quickly increasing in use over the last years: permutation symmetries, in particular within graph neural networks (GNNs), in Chapters 2, 4, and 6¹, and rotation symmetries in Chapter 8. While there are many instantiations of GNN architectures, we will predominantly be focusing on the message passing paradigm [13–15], where node representations are updated based on messages received from their neighbours.

Often, especially in the case of graph representation learning, the graph given as input data is also used as the *computation graph* over which messages are passed. Formally, we can describe this as a framework which **Encodes** the input, **Processes** it and the finally **Decodes** the predicted solution to the problem under study, and it has proved sufficient in many cases [16]. In the coming chapters, we consider the cases where the natural input does not offer the exact computation graph over which the GNN layer should be applied – we will show how more potential can be found if we consider an extra step, which we will call **Modulating** the input before processing it, by directly addressing assumptions about the input graph. This Encode-Modulate-Process-Decode framework is portrayed in Figure 1.

More precisely, in the first core chapter we study the performance of GNNs on graphs where labels are assumed to be influenced by interactions between nodes that are far apart. *Long-range interactions*, especially when the input graph contains bottlenecks, may require

1. We note that Transformers, the models behind the many successes of Large Language Models, also belong to this class [12].

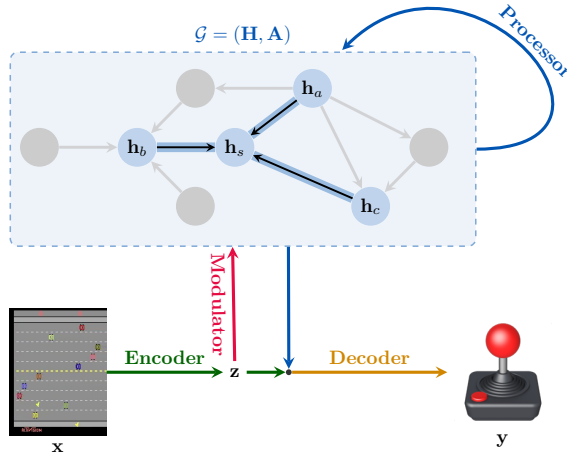


Figure 1 – High-level overview of the proposed framework. The encoder-decoder framework is a general machine learning paradigm, with encode-process-decode being commonly used as a way to process graph-structured inputs. Our proposal instead focuses on the graph \mathcal{G} underlying the task. For any task, we need to define the functions of 1) the modulator, whose purpose is to build the graph \mathcal{G} and 2) the processor, which derives the auxiliary measures to be used by the decoder.

GNN layers to summarise information from an exponentially increasing receptive field, leading to the problem known as over-squashing [4]. Due to the fact that most GNNs aggregate information in a radius increasing by one with every stacked message passing layer, the input graph does not adequately align with the required computation graph, if long-range dependencies are required. To tackle this, we propose using *expander graphs* to complement the input graph to form a computation graph without long-range dependencies. This is made possible by the fact that expander graphs are mathematical objects with low diameter by definition, allowing global diffusion of information using logarithmically many layers.

In the second core chapter, we tackle GNNs’ low performance on *heterophilic data*. In heterophilic graphs, a large proportion of nodes connected by edges do not share the same label, making it highly challenging to trivially extract the “salient” neighbours for downstream prediction tasks. However, unlike in the previous case, there are no known template graphs that can be composed with input graphs to improve the measure of interest (spectral gap in the case of bottlenecks, homophily in this case). Therefore, we use *weak classifiers* to determine which edges should be added to the input graph to obtain a computation graph with improved homophily.

For the third core chapter, we demonstrate how GNNs can find useful application beyond supervised learning tasks on graphs, by deploying them on general-purpose *reinforcement learning* (RL) problems. To enable this application, we leverage a strength of GNNs – their alignment with dynamic programming, which recommends them as a possible class of neural

networks to learn to imitate algorithms. This becomes particularly relevant in the case of RL tasks requiring planning—where the network needs to demonstrate capability to reason about the consequences of taking sequences of actions, before ever acting in the environment. Planning is an area of RL which is highly amenable to classical algorithms: for example, the value iteration algorithm provably plans optimally if the underlying RL environment is fixed and known. However, as in most environments of interest this is not the case, we use self-supervised transition models to construct relevant environment graphs to provide to a GNN, which then acts as an implicit planning mechanism. We can equip this GNN with planning capabilities by pre-training it to follow value iteration execution traces, over large quantities of synthetically generated environments—for which the dynamics are assumed known.

Lastly, in the fourth core chapter, we broaden the usage of geometric deep learning models in reinforcement learning. Specifically, while in the third chapter we leveraged GNNs—a permutation-equivariant model, which is among the most permissive geometries to optimise for—in RL tasks, here we focus on RL environments that exhibit additional symmetries, specifically, environments requiring *rotational equivariance*. Informally, an RL environment is rotationally equivariant if taking a particular action in the original environment is equivalent to taking a rotated action in the equally-rotated environment. To leverage these rotational symmetries in both the state and action space, we modulate the input by applying relevant elements of the rotation group to the input. Equivariant models constructed in this way can then be used as a backbone for state-of-the-art RL agents; in this case we focused on the popular *MuZero* agent. This leads to improving the empirical data efficiency of MuZero, as well as being able to provide theoretical guarantees that, as long as all the neural networks used by MuZero are equivariant, all of the computations and outputs of the MuZero algorithm will also be equivariant.

Before presenting the individual articles of the core chapters, we will present the necessary background knowledge in Chapter 1. This will span foundational material on diverse areas of deep learning: graph representation learning [17], geometric deep learning [18], neural algorithmic reasoning [19], reinforcement learning [20] and model-based planning [21]—all of which are required for easier comprehension of the core content in the main chapters of the thesis. Further, the Background chapter will also illuminate the need for modulating input structures, by presenting several case studies on GNNs on real-world biochemical tasks, which I have collaborated on before and during the period of my PhD. Even though such tasks are defined over *molecular graphs*—a “gold-standard” structure, known and given upfront—we illustrate that careful modulation of their computation graph can yield measurable improvements to downstream model performance. Perhaps, with hindsight, this comes as no surprise: chemical structures are often merely an approximation to the underlying *physics*, wherein *all* pairs of atoms can interact—not only ones linked by a chemical bond.

Once the foundations and motivations are ascertained, the core contributions of the thesis are presented in pairs of chapters. Namely, each of the core articles is accompanied by a “prologue” chapter, contextualising the article’s contributions in past and present related work, and describing the specific mechanism of computation graph modulation employed within the article. Chapters 2–3 cover *Expander Graph Propagation* [22, **EGP**], a method for ameliorating long-range dependencies in computational graphs by leveraging expander graphs as propagation templates. Chapters 4–5 cover *Evolving Computation Graphs* [23, **ECG**], an approach of reducing the heterophily in computational graphs by leveraging embeddings from weak classifiers. Chapters 6–7 cover *eXecuted Latent Value Iteration Networks* [24, **XLVIN**], a mechanism for deploying algorithmically-aligned GNNs in reinforcement learning tasks, using self-supervised transition models to modulate the computation graphs used by the GNN. Lastly, Chapters 8–9 cover *Equivariant MuZero* [25, **EqMuZero**], a framework for model-based planning that explicitly incorporates environment symmetries into the computational graph of the MuZero algorithm, for provable and empirical improvements to low-data efficiency.

Finally, Chapter 10 offers concluding remarks, and identifies fruitful avenues for further work, which I am aiming to pursue in the future.

Chapter 1

Background and Motivation

This chapter introduces the necessary background for tackling the problems presented in the rest of the thesis and understanding the solutions proposed, as well as the motivation behind them. We will start by describing graph representation learning, focusing on graph neural networks and a few of their most popular variants, which often constitute the baselines in my articles. This will be followed by a discussion of the expressive power of GNNs and the cases where they struggle, including under-reaching and over-smoothing, but also over-squashing, tackled in Chapter 2, and heterophilic data, the focus of Chapter 4, followed by a use case of GNNs’ advantages: learning to execute algorithms, a paradigm known as Neural Algorithmic Reasoning, the basis of Chapter 6. To better contextualise GNNs and to prepare the background necessary for the last chapter, Chapter 8, we introduce geometric deep learning, focusing on symmetry groups relevant to the articles (permutation and rotation groups), and the notion of equivariance. As the last two articles move from the area of supervised learning to reinforcement learning, we introduce RL and Markov Decision Processes, together with the relevant background of model-free and model-based agents. This chapter concludes with a few motivational tasks – examples of GNNs for real-world applications, giving practical examples of some of the theoretical insights described above.

1.1. Graph Representation Learning

In this section, we introduce the generic setup of learning representations on graphs. We denote graphs by $G = (V, E)$, where V is the set of nodes and E is the set of edges, and we denote by $(u, v) \in E$ an edge that connects nodes u and v . For the datasets considered in this thesis, we can assume that the input graphs are provided to the GNNs via two inputs: the *node feature matrix*, $\mathbf{X} \in \mathbb{R}^{|V| \times k}$ (such that $\mathbf{x}_u \in \mathbb{R}^k$ are the input features of node $u \in V$), and the *adjacency matrix*, $\mathbf{A} \in \{0, 1\}^{|V| \times |V|}$, such that a_{uv} indicates whether nodes u and v are connected by an edge. We further assume the graph is *undirected*; that is, $\mathbf{A} = \mathbf{A}^\top$. We

Table 1 – Summary of notation introduced in this section.

	Symbol	Definition
GNNs	G	the graph
	V	the set of nodes/vertices
	E	the set of edges
	\mathbf{A}	the adjacency matrix
	\mathbf{X}	the node feature matrix
	$\mathcal{N}_k(u)$	the k -hop neighbourhood of node u
RL	S	the state space
	A	the action space
	P	the transition function
	R	the reward function
	π	the policy function
	γ	the discount factor
	G_t	the return at time t
	τ	trajectory of the form $(s_0, a_0, r_1, s_1, a_1, r_2, \dots)$

also use $d_u = \sum_{v \in V} a_{uv}$ ($= \sum_{v \in V} a_{vu}$) to denote the degree of node u and $\mathcal{N}_k(u)$ to refer to the k -hop neighbourhood of node u .

1.1.1. Graph neural networks

We will describe GNNs using the Encode-Process-Decode pipeline [16].

Encode. The input node features \mathbf{X} are initially transformed using an encoder to obtain the inputs to the GNN layer. This is usually denoted as $\mathbf{h}_u^{(0)} = \mathbf{enc}(\mathbf{x}_u)$, with $\mathbf{h}_u^{(l)}$ being the representation of the node u after applying l GNN layers. The encoder, \mathbf{enc} , can be any kind of neural network, such as a multi-layer perceptron.

Process. The GNN layers following the encoder are often referred to as the “processor network”. Each of these layers performs a round of message passing [13]: propagating the information over the graph’s edges. Under this framework, we can divide computing node representations after layer l in two stages – computing the messages using a function M and *aggregating* them in a permutation invariant manner, using an aggregation function \oplus (such as sum, max or average):

$$\mathbf{m}_u^{(l)} = \bigoplus_{v \in \mathcal{N}_1(u)} M^{(l)}(\mathbf{h}_u^{(l)}, \mathbf{h}_v^{(l)}, \mathbf{e}_{uv}^{(l)}) \quad (1.1.1)$$

and using the computed messages to *update* the node representations using a function U :

$$\mathbf{h}_u^{(l+1)} = U^{(l)}(\mathbf{h}_u^{(l)}, \mathbf{m}_u^{(l)}) \quad (1.1.2)$$

Equivalently, taking $\psi = M$ and $\phi = U$, one step layer of a GNN can be summarised as follows, as popularised by [18]:

$$\mathbf{h}_u^{(l)} = \phi^{(l)} \left(\mathbf{h}_u^{(l-1)}, \bigoplus_{(u,v) \in E} \psi^{(l)}(\mathbf{h}_u^{(l-1)}, \mathbf{h}_v^{(l-1)}) \right) \quad (1.1.3)$$

where, by definition, we set $\mathbf{h}_u^{(0)} = \mathbf{x}_u$. Leveraging different (potentially learnable) functions for $\phi^{(l)} : \mathbb{R}^k \times \mathbb{R}^m \rightarrow \mathbb{R}^{k'}$, $\bigoplus : \text{bag}(\mathbb{R}^m) \rightarrow \mathbb{R}^m$ and $\psi^{(l)} : \mathbb{R}^k \times \mathbb{R}^k \rightarrow \mathbb{R}^m$ then recovers well-known GNN architectures.

Examples, where $\psi^{(l)}$ has a specialised form, include:

- GCN [14]: $\psi^{(l)}(\mathbf{x}_u, \mathbf{x}_v) = \beta_{uv} \omega^{(l)}(\mathbf{x}_v)$, with $\beta_{uv} \in \mathbb{R}$ being a constant based on \mathbf{A}
- Chebyshev Networks [26]: $\psi^{(l)}(\mathbf{x}_u, \mathbf{x}_v) = \beta_{uv} \omega^{(l)}(\mathbf{x}_v)$, with $\beta_{uv} \in \mathbb{R}$ being a constant based on the Chebyshev polynomials of the Laplacian of \mathbf{A}
- GAT [15]: $\psi^{(l)}(\mathbf{x}_u, \mathbf{x}_v) = \alpha^{(l)}(\mathbf{x}_u, \mathbf{x}_v) \omega^{(l)}(\mathbf{x}_v)$ with $\alpha^{(l)} : \mathbb{R}^k \times \mathbb{R}^k \rightarrow \mathbb{R}$ being a (softmax-normalised) self-attention mechanism.
- GatedGCN [27]: $\psi^{(l)}(\mathbf{x}_u, \mathbf{x}_v) = \eta^{(l)}(\mathbf{x}_u, \mathbf{x}_v) \odot \omega^{(l)}(\mathbf{x}_v)$, where $\eta^{(l)} : \mathbb{R}^k \times \mathbb{R}^k \rightarrow \mathbb{R}_+^{k'}$ is an edge gating mechanism, and \odot is the Hadamard product. Later iterations of GatedGCN [28, 29] also maintain the gating vectors as edge features, and incorporate an explicit residual connection [30] between them.
- GIN [1]: $\psi^{(l)}(\mathbf{x}_u, \mathbf{x}_v) = \mathbf{x}_v$. GINs have been deliberately designed as maximally-expressive GNNs with the simplest possible message function, delegating all computations to $\phi^{(l)}$. We will study GINs’ expressiveness in more detail in the next section.

Decode. Once the processor has been iterated for a desirable number of steps, T , we require appropriate decoder networks to obtain final model predictions. Assuming we only updated node features to $\mathbf{h}_u^{(T)}$, we can obtain node-, edge-, or graph-level predictions as follows:

$$\mathbf{y}_u = \text{dec}_{\text{node}}(\mathbf{h}_u^{(T)}) \quad (1.1.4)$$

$$\mathbf{y}_{uv} = \text{dec}_{\text{edge}}(\mathbf{h}_u^{(T)}, \mathbf{h}_v^{(T)}, \mathbf{e}_{uv}^{(T)}) \quad (1.1.5)$$

$$\mathbf{y}_G = \text{dec}_{\text{graph}}\left(\bigoplus_{u \in V} \mathbf{h}_u^{(T)}\right) \quad (1.1.6)$$

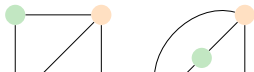
These will then be compared with the ground-truth labels, computing a loss function whose gradients are used to optimise the GNN parameters. The decoder networks, **dec**, can be any kind of neural network, such as a multi-layer perceptron. Section 1.5 discusses in more depth the form of the decoder, depending on the given task.

1.1.2. Expressive power of graph neural networks

Expressive power of graph neural networks has been increasingly studied in recent years, from various angles. For example, one can establish GNNs’ capacity for distinguishing non-isomorphic graphs, often relative to the Weisfeiler-Leman graph isomorphism test [31]. Alternately, one can quantify the extent to which various GNNs are vulnerable to

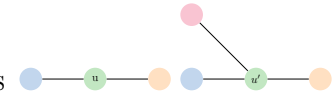
pathological effects, such as over-smoothing [32], under-reaching [33] and over-squashing [4]. Lastly, GNNs have often been studied under an algorithmic alignment lens [1], quantifying their level of correspondence to classical algorithms. We will briefly describe each of these approaches in the following sections.

Weisfeiler-Leman (WL) test. WL tests if two graphs can be mapped to each other, while preserving their structure – a property named graph isomorphism, for which the following



two graphs can provide an example: While there is currently no known solution for the graph isomorphism problem in polynomial time, the WL test is a computationally efficient way of testing it. WL provides a necessary but insufficient condition for isomorphism – if the outcome of the WL test is that the graphs are non-isomorphic, then that is the correct answer, while if the test succeeds, it could be that the pair is isomorphic, but not necessarily. The test perform iterative node coloring, starting by initialising each node u with the same color $c_u^{(0)} = C$, followed by repeatedly aggregating the labels of the node $c_u^{(l)}$ and of its neighbors' labels $c_v^{(l)}$ for $v \in \mathcal{N}_1(u)$ and injectively hashing this multiset of labels to obtain its new label $c_u^{(l+1)}$.

The WL test provides a way to characterize the power of GNNs – specifically, a GNN is equally powerful to the WL test if it can distinguish between the same pairs of graphs that the WL test can distinguish. In Xu et al. [1], it is shown that aggregators such as

mean and max cannot distinguish between simple graphs such as  and, consequently, that GNNs such as GCN and GraphSAGE are not as powerful as WL. Xu et al. then propose the Graph Isomorphism Network (GIN), which is equally expressive to the WL test, and provably also the most expressive GNN over featureless graphs:

$$\mathbf{h}_u^{(l)} = \phi^l \left((1 + \epsilon^{(l)})\mathbf{h}_u^{(l-1)} + \sum_{v \in \mathcal{N}_1(u)} \mathbf{h}_v^{(l-1)} \right) \quad (1.1.7)$$

where ϵ is a learnt or fixed scalar. The GIN is also the main baseline in Chapter 2.

Under-reaching. The number of GNN layers dictates what is the maximum distance across which nodes can communicate – for a GNN with k layers, node u will send and receive information to/from nodes that are at most k hops away. When the task depends on interaction that are across larger distances than the number of GNN layers, the GNN will suffer from *under-reaching* [33], and will be unable to complete the task.

Over-smoothing. While having too few layers is a problem as the GNN doesn't model the long range interactions the task depends on, increasing the number of layers can also be problematic for GNN learning. With more layers, due to repeated aggregation, the nodes can

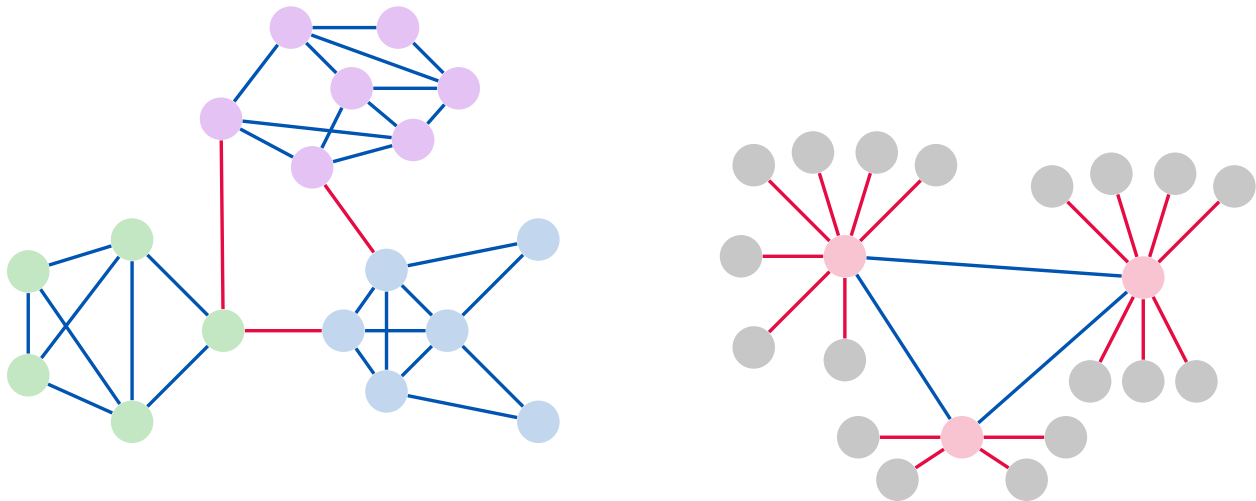
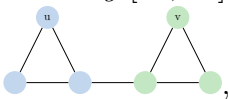
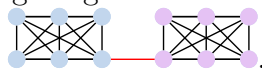


Figure 2 – Diagram of a *homophilic* graph (**left**) and a *heterophilic* graph (**right**). Node colours represent classes, and edges are coloured blue if they connect nodes of the same class, and red otherwise. Homophily (edges mostly blue) often arises in the presence of *community structures*; for example, when classifying documents in a citation network, documents with similar topics will tend to cite one another more frequently than documents with different topics. Heterophily (edges mostly red), instead, can arise if the graph is misaligned to the task, or if the network has adversarial actors; for example, in a fraud detection setting, the fraudulent actors are very rare, and will focus on creating many more connections with legitimate nodes than with other fraudsters, to avoid detection.

obtain indistinguishable representations leading to decreasing performance, a phenomenon called *over-smoothing* [32, 34]. For example, if we consider blue and green to be the labels of

the nodes in , mixing the features of a node and its neighbours can initially lead to all blue/green nodes being classified correctly, but once the number of GNN layers increases, nodes across clusters will exchange information, leading to nodes having the same representations regardless of which cluster they belong to. Over-smoothing is the main limiting factor towards building very deep GNNs.

Bottlenecks and Over-squashing. Another possible limitation of GNNs is their inability to learn long-range interactions when the graph has bottlenecks, as the one we portray in

Chapter 2: . In this case, the red edge—linking the two cliques together—will be under enormous pressure in order to allow nodes to communicate across the cliques. Specifically, on certain highly-bottlenecked graphs (such as trees), nodes may need to summarise exponentially increasing amounts of information into fixed-size vectors. This effect has been commonly denoted as *over-squashing* of information [4], and may prevent GNNs from achieving good performance even on their training data.

Heterophilic graphs. Lastly, GNNs have an implicit assumption that neighbouring nodes will tend to *reinforce* each other’s representations for downstream prediction tasks—which

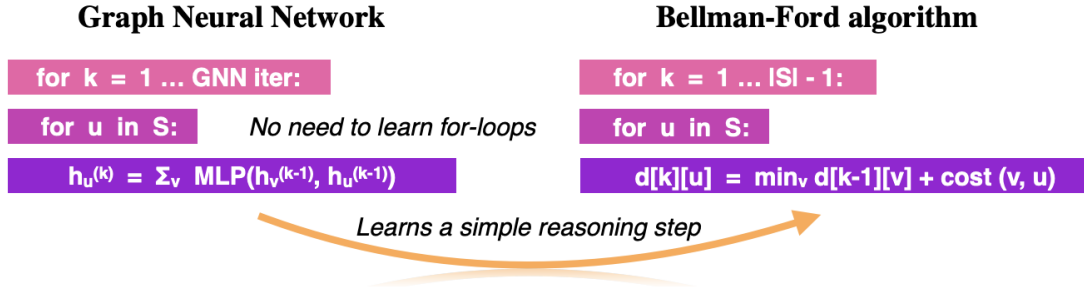


Figure 3 – Picture from Xu et al. [6] indicating the alignment between Graph Neural Networks and Bellman-Ford, a dynamic programming algorithm for finding shortest paths.

motivates repeated aggregation over neighbourhoods. In node classification tasks, this carries the implication that nodes tend to be connected by an edge only if they belong to the same class, and thus their representations should be close. GNNs, accordingly, struggle to perform well when this assumption does not hold; i.e., on heterophilic data, such as the graph in Figure 2. Certain architectural changes have been proposed to tackle this issue, such as separately modelling the self- and neighbourhood information [35]:

$$\mathbf{h}_u^{(l)} = \mathbf{W}_{\text{self}}^{(l)} \phi_1^{(l)}(\mathbf{h}_u^{(l-1)}) + \mathbf{W}_{\text{agg}}^{(l)} \phi_2^{(l)} \left(\bigoplus_{(u,v) \in E} \psi^{(l)}(\mathbf{h}_u^{(l-1)}, \mathbf{h}_v^{(l-1)}) \right) \quad (1.1.8)$$

where the self-connection allows the model to apply specific focus to the receiving node, and hence apply stronger forgetting to the neighbourhood messages if they prove distracting.

In Chapter 4, we will take a complementary approach – modifying the computation graph by leveraging weak classifiers, instead of modifying the GNN architecture, in order to reduce the heterophily in the input graph.

1.2. Neural Algorithmic Reasoning

Neural Algorithmic Reasoning (NAR) [19] is a paradigm that aims to combine neural networks with algorithms to take advantage of their complementary benefits. Neural networks work on raw, possibly noisy inputs and can be used across different tasks, but to achieve this they require large amounts of data, while remaining unreliable when extrapolating and lacking interpretability. On the other hand, algorithms are not robust to task variations and require inputs to be in a specific format, but once that condition is met, they can trivially generalise to inputs of different scales while maintaining guaranteed correctness and interpretability. Overall, the idea of NAR is to have neural networks learn to execute algorithmic computation, so that we can leverage the *robust* priors of algorithms if we know our target might depend on that algorithm’s operations.

Graph Neural Networks for Dynamic Programming. Dynamic programming (DP) is a computer programming technique that has been at the center of NAR, partially because Xu et al. [6] promptly discovered the **algorithmic alignment** between GNNs and DP, portrayed in Figure 3. More precisely, dynamic programming solves a problem x by *expanding* it into sub-problems $\eta(x)$, until we reach the base instances for which the solution can be trivially obtained and *scored* in terms of quality for the final solution of x . Lastly, the sub-problems are *recombined* to obtain the solution for x . Formally, this can be described as in Dudzik and Veličković [36]:

$$\text{dp}[x] \leftarrow \text{recombine}(\text{score}(\text{dp}[y], \text{dp}[x]) \text{ for } y \text{ in } \text{expand}(x)) \quad (1.2.1)$$

By associating sub-problems with nodes and the links between a problem and its directly expanded sub-problems with edges, we obtain a graph. In the Bellman-Ford example from Figure 3, which computes the shortest paths from a given source node s , the set of sub-problems is the set of nodes V , the expansion of a node u is given by its one-hop neighborhood $\mathcal{N}_1(u)$, with the base cases setting the initial distance assumptions: zero from source to source $d_s = 0$ and infinity otherwise $d_u = \infty$. The distances are updated by recombining the solutions of the sub-problems, with the distances between two vertices v and u being represented on the edge between them, $w_{v \rightarrow u}$, as follows:

$$d_u = \min(d_u, \min_{v \in \mathcal{N}_1(u)} d_v + w_{v \rightarrow u}) \quad (1.2.2)$$

With this formulation, it is easier to see that the GNN’s computation graph (algorithmically) aligns with the DP computation, resulting in the fact that the MLPs that are part of the GNN’s processor need to learn simple update equations (such as the addition between a neighbour node’s features, d_v , and the weight of the edge connecting it, $w_{v \rightarrow u}$). Alternatively, learning a DP algorithm with an MLP instead of the GNN, would require the model to learn the for-loop computation emphasised in Figure 3 that iterates over all sub-problems. Thus, GNNs are well suited to imitate DP algorithm execution, as their algorithmic alignment leads to data efficiency and generalisation abilities – an insight on which we base Chapter 6.

1.3. Geometric Deep Learning

While previously we focused on graph neural networks, Bronstein et al. [18] remarks how they can be unified with many other popular deep learning architectures (convolutional neural networks, recurrent neural networks, Transformers, GNNs and beyond) by viewing deep learning from the perspective of symmetries in data. This approach is known as geometric deep learning, and it has been one of the most successful directions in deep learning research in the past decade. To better understand this and prepare for using geometric deep learning advances in RL in Chapter 8, we will first introduce fundamental aspects of group theory.

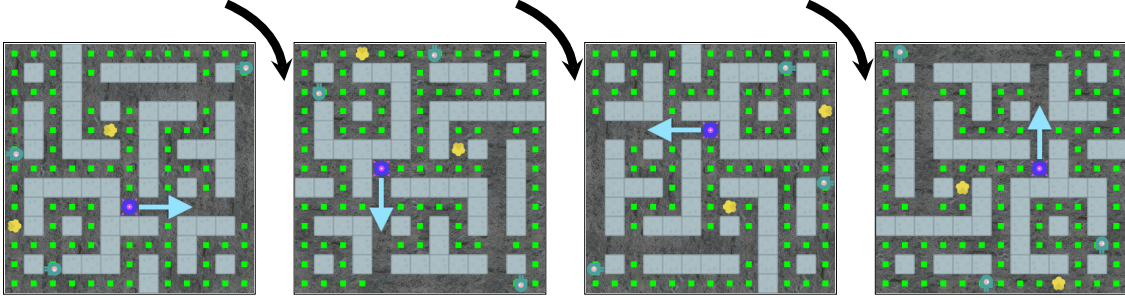


Figure 4 – A frame from the Chaser environment in the ProcGen suite [7] and its 90° , 180° and 270° rotations.

1.3.1. Groups and Representations

A *group* (\mathfrak{G}, \circ) is a set \mathfrak{G} equipped with a *composition* operation $\circ : \mathfrak{G} \times \mathfrak{G} \rightarrow \mathfrak{G}$ (written concisely as $\mathfrak{g} \circ \mathfrak{h} = \mathfrak{gh}$), satisfying the following axioms:

- (*associativity*) $(\mathfrak{gh})\mathfrak{l} = \mathfrak{g}(\mathfrak{hl})$ for all $\mathfrak{g}, \mathfrak{h}, \mathfrak{l} \in \mathfrak{G}$;
- (*identity*) there exists a unique $\mathfrak{e} \in \mathfrak{G}$ satisfying $\mathfrak{eg} = \mathfrak{ge} = \mathfrak{g}$ for all $\mathfrak{g} \in \mathfrak{G}$;
- (*inverse*) for every $\mathfrak{g} \in \mathfrak{G}$ there exists a unique $\mathfrak{g}^{-1} \in \mathfrak{G}$ such that $\mathfrak{gg}^{-1} = \mathfrak{g}^{-1}\mathfrak{g} = \mathfrak{e}$.

Groups are a natural way to describe *symmetries*: object transformations that leave the underlying object unchanged. Chapter 8, which is one of the core contributions of the thesis, studies geometric deep learning for RL. In this chapter, we empirically evaluate the agent on the group of 90° rotations, which has four elements. Figure 4 displays a frame from the Chaser [7] environment which has been acted upon with each element of the group, by rotating the observation and the action.

Groups can be reasoned about in the context of linear algebra by using their *real representations*: functions $\rho_{\mathcal{V}} : \mathfrak{G} \rightarrow \text{GL}(\mathcal{V})$ that give, for every group element $\mathfrak{g} \in \mathfrak{G}$, a real, invertible matrix demonstrating how this element *acts* on a vector space \mathcal{V} . $\text{GL}(\mathcal{V})$ is the *general linear group* over \mathcal{V} —the group of all invertible matrices operating over vectors in \mathcal{V} .

For example, for the rotation group $\mathfrak{G} = \text{SO}(n)$, and $\mathcal{V} = \mathbb{R}^n$, the representation $\rho_{\mathcal{V}}$ could provide an appropriate $n \times n$ rotation matrix for each rotation \mathfrak{g} . As another example, for the n -element permutation group $\mathfrak{G} = S_n$ and $\mathcal{V} = \mathbb{R}^m$, with $m \geq n$, the representation $\rho_{\mathcal{V}}$ could provide an $m \times m$ permutation matrix, where a specific collection of n elements are permuted according to the permutation \mathfrak{g} , with the other $m - n$ elements fixed in place.

1.3.2. Equivariance and Invariance

As symmetries are assumed to not change the essence of the data they act on, we would like to construct neural networks that adequately represent such symmetry-transformed inputs. Assume we have a neural network $f : \mathcal{X} \rightarrow \mathcal{Y}$, mapping between vector spaces \mathcal{X} and \mathcal{Y} , and that we would like this network to respect the symmetries within a group \mathfrak{G} . Then

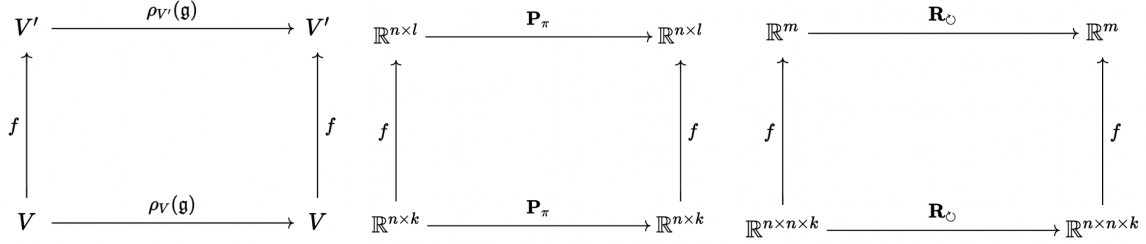


Figure 5 – Commutative diagram displaying \mathfrak{G} -equivariance of function f , as described in Equation 1.3.1 – general case to the left, then for permutation and rotation respectively.

we can impose the following condition, for all group elements $\mathfrak{g} \in \mathfrak{G}$ and inputs $\mathbf{x} \in \mathcal{X}$:

$$f(\rho_{\mathcal{X}}(\mathfrak{g})\mathbf{x}) = \rho_{\mathcal{Y}}(\mathfrak{g})f(\mathbf{x}). \quad (1.3.1)$$

This condition is known as \mathfrak{G} -equivariance—for any group element, it does not matter whether we act with it on the input or on the output of the function f —the end result is the same. A special case of this, \mathfrak{G} -invariance, arises when the output representation is trivial ($\rho_{\mathcal{Y}}(\mathfrak{g}) = \mathbf{I}$):

$$f(\rho_{\mathcal{X}}(\mathfrak{g})\mathbf{x}) = f(\mathbf{x}). \quad (1.3.2)$$

In geometric deep learning, equivariance to reflections, rotations, translations and permutations has been of particular interest [18]. For example, if \mathcal{X} in Figure 5-left represents node features of a graph and $\rho_{\mathcal{X}}(\mathfrak{g})$ gives the permutation matrix \mathbf{P} corresponding to \mathfrak{g} , then we can reconstruct the second commutative diagram, popular instantiations of it being graph neural networks, deep sets [37] and (graph) transformers [38, 39] similarly, with

$$f(\mathbf{P}\mathbf{X}) = \mathbf{P}f(\mathbf{X}) \quad (1.3.3)$$

The right figure refers to rotations, such as in the continuous group $\text{SO}(2)$, or the discrete group of 90° rotations, C_4 , which will be used in Chapter 8:

$$f(\mathbf{R}_\circ\mathbf{X}) = \mathbf{R}_\circ f(\mathbf{X}) \quad (1.3.4)$$

Note that, especially if the function f does not preserve the dimensionality of the space, the corresponding matrices \mathbf{P} or \mathbf{R}_\circ may be *different* on the left-hand side and right-hand side of these equations.

As an example given in Figure 4-right, if the function f maps image data in $\mathbb{R}^{n \times n \times k}$ (with k -dimensional features in each pixel of the $n \times n$ grid) to fixed-size vector representations in \mathbb{R}^m , then rotations \mathbf{R}_\circ of the input space of f are represented by pixel-mapping matrices ($\mathbb{R}^{n^2 \times n^2}$), and the rotations \mathbf{R}_\circ of the output space may be represented by rotation matrices over \mathbf{R}^m ($\mathbb{R}^{m \times m}$).

1.4. Reinforcement Learning

A reinforcement learning environment is usually stated as a discounted *Markov decision process* (MDP). 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*, $R : \mathcal{S} \times \mathcal{A} \rightarrow \mathbb{R}$ is a reward function, $P : \mathcal{S} \times \mathcal{A} \rightarrow \text{Dist}(\mathcal{S})$ is a *transition function* such that $P(s'|s,a)$ 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* $\pi : \mathcal{S} \times \mathcal{A} \rightarrow [0,1]$ such that $\pi(a|s)$ is a conditional probability distribution over actions given a state. The *return* is defined as:

$$G_t = \sum_{k=0}^{\infty} \gamma^k R(a_{t+k}, s_{t+k}) \quad (1.4.1)$$

Value functions represent the expected return induced by a policy in an MDP when conditioned on a state:

$$V^\pi(s,a) = \mathbb{E}_\pi[G_t | s_t = s] \quad (1.4.2)$$

or state-action pair:

$$Q^\pi(s,a) = \mathbb{E}_\pi[G_t | s_t = s, a_t = a] \quad (1.4.3)$$

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, to find a π^* it suffices to find the unique optimal value function V^* as the fixed-point of the Bellman optimality operator. The optimal value function V^* is such a fixed-point and satisfies the *Bellman optimality equations* [40]:

$$V^*(s) = \max_{a \in \mathcal{A}} \left(R(s,a) + \gamma \sum_{s' \in \mathcal{S}} P(s'|s,a) V^*(s') \right) \quad (1.4.4)$$

Value iteration (VI) is a successive approximation method for finding the optimal value function of a discount MDP as the fixed-point of the so-called Bellman optimality operator [41]. VI randomly initialises a value function $V_0(s)$, and then iteratively updates it as follows:

$$V_{t+1}(s) = \max_{a \in \mathcal{A}} \left(R(s,a) + \gamma \sum_{s' \in \mathcal{S}} P(s'|s,a) V_t(s') \right) . \quad (1.4.5)$$

Value iteration is guaranteed to converge and it is a fundamental principle behind value-based methods.

Value-based methods. The main idea behind value-based methods, such as Q-learning, is that if we have access to a function giving what the return would be from a state, such as $V^*(s)$, or from taking a particular action in a given state—often denoted as $Q^*(s,a)$ —then

the optimal policy can be obtained by considering the actions that maximise the rewards:

$$\pi^*(a|s) = \begin{cases} 1, & \text{if } a = \arg \max_a Q^*(s,a) \\ 0, & \text{otherwise} \end{cases} \quad (1.4.6)$$

However, in most environments the optimal value function cannot be computed, as we don't have access to the reward function, R , or the transition function, P . In this case, we can estimate Q , for example, by minimising the temporal difference (TD):

$$\delta = Q(s,a) - (R(s,a) + \gamma \max_a Q(s',a)) \quad (1.4.7)$$

While value-based methods have achieved impressive results in deep reinforcement learning [42], they require learning *value functions*, which can be considered as only an intermediate step on the path to the true objective – learning good *policies*. Policy-based methods, presented next, optimise the policy objective directly instead. There exist non-parameteric methods that optimise the policy, such as policy iteration, and policy-gradient methods that use neural networks as function approximators for the policy. In the next part we will focus on the latter, as this constitutes the backbone of our RL agents in Chapters 6 and 8.

Policy-gradient methods. Policy-gradient methods optimise parameterised policies with respect to the expected return using gradient ascent. That is, for a policy π_θ parametrised by θ , policy gradient would update its parameters iteratively as follows:

$$\theta \leftarrow \theta + \eta \nabla_\theta \mathbb{E}_{\pi_\theta}(G_t) \quad (1.4.8)$$

where $\mathbb{E}_{\pi_\theta}(G_t)$ is the expected return when acting using policy π_θ , and η is a learning rate hyperparameter. Such an update has the underlying aim of increasing the probabilities of the actions resulting in high cumulative rewards. One such fundamental algorithm is REINFORCE [43], computing gradients as follows:

$$\nabla_\theta \mathbb{E}_{\tau|\pi_\theta}(R(\tau)) = \mathbb{E}_{\tau|\pi_\theta} \left(R(\tau) \sum_{t=0}^{k_\tau} \nabla_\theta \log \pi_\theta(s_t, a_t) \right) \quad (1.4.9)$$

where π_θ is the neural network, with parameters θ , approximating the desired policy. τ is a (state-action-reward) trajectory ($\tau = (s_0, a_0, r_0, s_1, a_1, \dots, s_{k_\tau}, a_{k_\tau}, r_{k_\tau})$), and $R(\tau)$ is the overall return of this trajectory, computed for example as the G_t return value specified before. At each step of REINFORCE, we sample a trajectory τ from our current policy π_θ , then evaluate this trajectory's return $R(\tau)$, and use the update rule to compute a gradient for θ to follow.

Note that REINFORCE has a clear issue with *credit assignment*: it attaches the overall trajectory return $R(\tau)$ to *all* the actions taken within τ , with no easy way to distinguish good actions from bad ones within it. To tackle problems such as credit assignment, exploration and the bias-variance trade-off, modified versions of REINFORCE were proposed such as REINFORCE with baseline, Actor-Critic, Advantage Actor Critic (A2C), Asynchronous

Advantage Actor Critic (A3C) [44], Trust-Region Policy Optimization (TRPO) [45] and Proximal Policy Optimization (PPO) [46].

In particular, REINFORCE uses the score of observed trajectories to increase or decrease the probabilities of *every action* in that sequence. While this is unbiased and relying on true, rather than estimated returns, its high variance can be problematic – returns of trajectories starting from the same state can vary significantly due to the stochasticity in the policy or the environment. Increasing the batch size could reduce the variance in aggregate, but at the cost of sample efficiency.

An alternative is presented by the class of **actor-critic** methods, where the actor (policy that controls the agent π_θ) is augmented by a critic q_w , a neural network parametrized by w , which estimates how good the action taken is. In particular, the policy parameters are updated by scaling the actions’ probabilities using these action value estimates, instead of the observed cumulative returns (with learning rate α):

$$\Delta\theta = \alpha \nabla_\theta (\log \pi_\theta(s_t, a_t)) q_w(s_t, a_t) \tag{1.4.10}$$

At the same time, we seek to improve the action value estimates by updating the critic’s parameters using the TD error (with learning rate β):

$$\Delta w = \beta (R(s_t, a_t) + \gamma q_w(s_{t+1}, a_{t+1}) - q_w(s_t, a_t)) \nabla_w q_w(s_t, a_t) \tag{1.4.11}$$

The parameters of the actor and the critic are updated separately.

While this actor-critic formulation moves from scaling every action probability equally in a trajectory based on the overall return to considering individual action values, it could still suffer from learning instability due to disproportionate gradients if, for example, all actions in a state have a high value. A2C instead scales action probabilities using an advantage function, which calculates how much better an action is compared to the average value in that state:

$$A(s, a) = Q(s, a) - V(s) \tag{1.4.12}$$

Even after such fixes, it was noticed that large steps in policy updates harm the stability of RL training. TRPO and PPO address this limitation by proposing objectives that control how far a new policy is allowed to deviate from the previous iteration of the policy. Thus, they propose updating the parameters conservatively and avoiding having too much variability in training (large updates) or too slow learning (small updates).

We will focus on Proximal Policy Optimization, a central building block in Chapter 6. PPO builds on the previously mentioned proposals and is often the baseline of choice for model-free agents, including in large language models that use reinforcement learning to learn from human feedback [47]. PPO obtains updated policy parameters by:

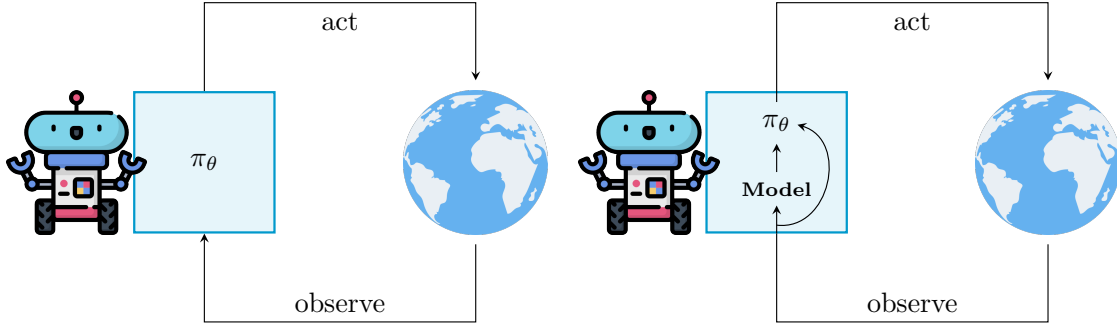


Figure 6 – Left portrays model-free agents, while right shows a general model-based agent.

$$\theta_{k+1} = \arg \max_{\theta} \mathbb{E}_{s,a \sim \pi_{\theta_k}} [\mathcal{L}(s, a, \theta_k, \theta)] \quad (1.4.13)$$

where the loss is given by:

$$\mathcal{L}(s, a, \theta_k, \theta) = \min \left(\frac{\pi_{\theta}(a|s)}{\pi_{\theta_k}(a|s)} A(s, a), \text{clip} \left(\frac{\pi_{\theta}(a|s)}{\pi_{\theta_k}(a|s)}, 1 - \epsilon, 1 + \epsilon \right) A(s, a) \right) \quad (1.4.14)$$

where ϵ defines the clip range, which controls how far the new policy can be from the old policy π_{θ_k} , only updating the policy parameters if the ratio of the policies is in the clipping range, or if the advantages bring it closer to the range. Note that, as specified in Equation 1.4.13, the trajectories used to optimise the PPO loss are sampled from the previous stable policy π_{θ_k} .

1.4.1. Model-based RL

Model-free agents as those described in the previous section are based on *reactive* policies, acting by adapting to observed rewards. In many cases, these require large quantities of data and are slow to adapt, as they cannot anticipate novel situations easily.

Model-based agents (Figure 6-right) aim to address these limitations by learning models of the world and using them to simulate the effects of actions before taking them. This leads to gains in data efficiency, as having a good model implies fewer interaction in the environment are needed to learn to act, quick adaptation to previously unexplored situations and better safety through modeling of consequences resulting from acting, as well as having the theoretical upside: perfect models are guaranteed to lead to planning perfect policies. Some of the most successful examples of model-based RL have been applied to game-playing and natural sciences [48–50, 8].

In terms of what the agent learns, how it learns and how it's used, there are many types of model-based RL agents. Here we will only mention briefly a few variants, and focus on MuZero [8] for the rest of the section, as it constitutes the backbone of Chapter 8.

Firstly, depending on what it learns, the model¹ can be:

- forward, for example predicting state transitions: $s_t, a_t \rightarrow s_{t+1}$ or what the reward is $s_t, a_t \rightarrow r_t$. This is the most common type of model, also used in MuZero.
- backward/reverse, for example modelling what could have been the previous state and action that was taken to reach the current state $s_{t+1} \rightarrow s_t, a_t$
- inverse, such as predicting which action takes the agent from one state to another $s_t, s_{t+1} \rightarrow a_t$

Regarding the manner of learning, some common examples include modelling one step dynamics, multi-step outputs or matching the distributions between observed trajectories and trajectories rolled out by the model. For example, when having a model that learns state transitions, minimising the error between predicted and observed next states can be seen as a supervised learning problem, or self-supervised objectives can be used, contrasting observed triplets s_t, a_t, s_{t+1} with triplets with randomly sampled next state s_t, a_t, s' , as we use in Chapter 6. For multi-step model learning, the input can be s_t together with a sequence of actions $a_t, a_{t+1}, a_{t+2}, \dots, a_{t+k}$, and the model is trained to predict next states $s_{t+1}, s_{t+2}, \dots, s_{t+k+1}$.

Once the model is available, it is common to use it in one of two ways: for planning, or for data augmentation. For example, model predictive control (MPC) samples multiple action sequences and selects the one with the highest evaluation (e.g. cumulative reward), whose first action will be used in the environment. This type of planning is done after each time step, thus belonging to the class of decision-time planners. Monte Carlo tree search (MCTS) is also in this category, but instead of randomly sampling action sequences, MCTS-based agents incrementally extend a search tree, where each node represents a state, with an associated value. The tree is used to determine which actions to choose such that the agent is more likely to transition to a state with a higher estimated value. Alternatively, the model can be used to simulate experiences, which can be gathered in a dataset and be used for value approximation or policy learning, in the style of Dyna [51].

There are many other methods of interest for model-based RL [52, 53]. We will next focus on MuZero [8], a highly-performant agent that established the state of the art on the full Atari benchmark and achieved superhuman results on chess, Go and Shogi, that is the backbone of Chapter 8.

MuZero. MuZero is a model-based agent that learns a multi-step model of forward dynamics leveraged for decision-time planning using MCTS. MuZero can be explained in three stages: planning, acting and training, portrayed in Figure 7, and depends on three neural networks:

- representation network, h , which embeds inputs into a high-dimensional space;

1. It is worth noting that, in all of the cases listed below, the learnt mappings are implicitly written as deterministic functions. However, in many cases, there could be multiple possible outcomes of state transitions, or backward states—as such, in practice, models may be *probabilistic* and *generative*.

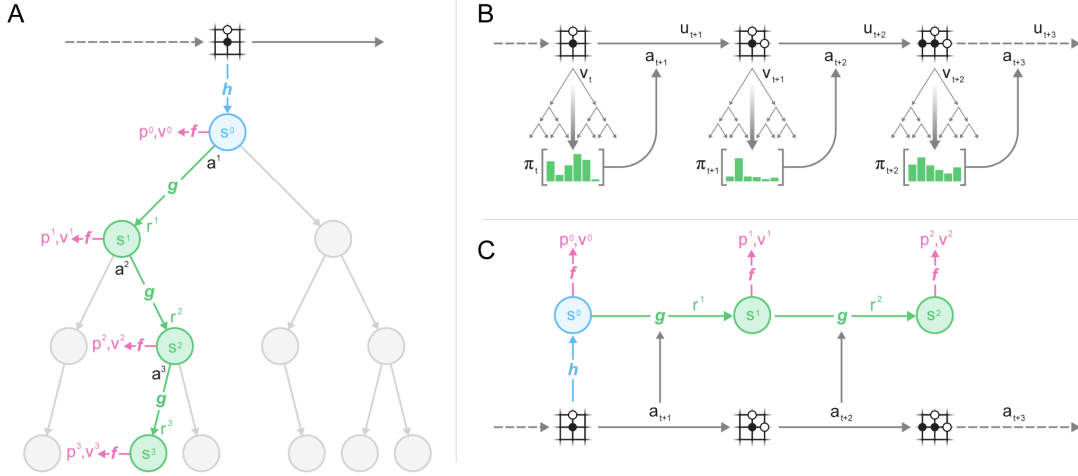


Figure 7 – Picture from [8] portraying how MuZero plans (A), acts (B) and trains (C).

- dynamics network, g , which predicts next state embeddings and rewards, conditioned on actions, and
- prediction network, f , which computes policy and value estimates from state embeddings.

In the first stage of planning, action selection is done using the stored statistics in each internal node of the MCTS tree using the *upper confidence bound* (UCB) [54], followed by expansion—using f and g —and the backup—when the statistics along the trajectory are updated. The output of MCTS is a policy π which is used in the second stage, acting, to select the actions to run in the environment. These trajectories are stored in a replay buffer and are used to train f, g and h by minimising the differences between what the model predicts for the reward r_t^k , value v_t^k and policy p_t^k functions and the data experiences stored:

$$l_t(\theta) = \sum_{k=0}^K l^r(u_{t+k}, r_t^k), l^v(z_{t+k}, v_t^k), + l^p(\pi_{t+k}, p_t^k) \quad (1.4.15)$$

with u_t being the true, observed rewards at time t , π_t , as mentioned, the policy probabilities computed by MCTS and used to select real actions at time t and $z_{t+k} = u_{t+1} + \gamma u_{t+2} + \dots + \gamma^{n-1} u_{t+n} + \gamma^n v_{t+n}$ is the value target. v_t is the estimated value of the state at time t , as computed by the MCTS algorithm.

1.5. Graph Neural Networks for Real-world applications

Graphs are ubiquitous data structures underlying virtually all tasks of interest, including natural inputs (molecules [55]), entity relations (transportation networks [56], chip placements [57]), or concept linking in **reasoning** processes (algorithms [6] and other theoretical

constructs [58, 59]). These properties are inherent in several of the most useful GNN benchmarks, such as the Open Graph Benchmark [60, OGB] and Benchmarking-GNNs [28].

We will next present possible tasks on graphs – first theoretically, then exemplified with applications.

1.5.1. Graph task types

For classification tasks, we consider C to represent the set of possible classes and \mathcal{L} is the cross-entropy loss function. For regression tasks, \mathcal{L} is the mean-squared error. In all tasks, the aim is to learn the function f , also known as the model, that optimizes the loss function. We assume that we model our task using a GNN with T layers.

Node classification. For node classification tasks, f takes as input a node with input features \mathbf{x}_u , which has an associated label $y_u \in C$. The aim is to minimise $\mathbb{E}[\mathcal{L}(y_u, \hat{y}_u)]$, where \hat{y}_u is the prediction of $f(\mathbf{X})_u = \hat{y}_u$.

For example, in the antibody-antigen tasks [61, 62] presented in Section 1.5.2.2, nodes are classified as belonging to the binding region or not, while Chapter 4 focuses on classifying nodes in the correct class despite being connected to nodes from different classes.

Graph classification. Graph classification refers to predicting one label for each graph sample in the dataset – for an input graph G and the label of the graph $y_G \in C$, we aim to learn f that minimises $\mathbb{E}[\mathcal{L}(y_G, \hat{y}_G)]$. In order to obtain a graph-level prediction, a graph aggregator can be used, such as summing all node features, followed by an MLP predictor ρ :

$$\hat{y}_G = \rho \left(\sum_{u \in V} \mathbf{h}_u^{(T)} \right) \quad (1.5.1)$$

An alternative is using a master node, often initialised as a zero-vector $\mathbf{h}_G^{(0)} = 0$ and updated after each message passing layer:

$$\mathbf{h}_G^{(t+1)} = \rho_{t+1} \left(\sum_{u \in V} \mathbf{h}_u^{(t+1)}, \sum_{(u,v) \in \mathcal{E}} \mathbf{h}_{uv}^{(t+1)}, \mathbf{h}_G^{(t)} \right) \quad (1.5.2)$$

Graph regression is exemplified in the first part of Section 1.5.2.2 through the property prediction of molecules, and graph classification is the main topic for Chapter 2, where long-range interactions between nodes are more easily leveraged before getting a graph-level prediction.

Edge prediction. The task of edge prediction, or link prediction, refers to predicting if an edge exists between a pair of nodes or predicting the type of this edge. More precisely, for

nodes u and v , the model predicts:

$$\hat{y}_{uv} = \rho(\mathbf{h}_u^{(T)}, \mathbf{h}_v^{(T)}) \tag{1.5.3}$$

For example, [63] predicts if two drugs are linked positively or negatively, while [64], mentioned in Section 1.5.2.2, classifies the relation between two drugs out of 964 classes.

In knowledge graph literature, edges are often *typed*, and referred to as *relations* (u, e_{uv}, v) , linking two nodes (which can also be referred to as *entities*). These nodes may be referred to as the *head* and *tail* of the relation. One popular loss, TransE [65], deployed in Chapter 6 to learn a transition model, contrasts true triplets (u, e_{uv}, v) with triplets using randomly sampled tails $(u, e_{uv'}, v')$ (or, equivalently, randomly sampled heads):

$$\mathcal{L}_{\text{TransE}}((u, e_{uv}, v), v') = \max(0, \gamma_m + d(\mathbf{h}_u + \mathbf{e}_{uv}, \mathbf{h}_v) - d(\mathbf{h}_u + \mathbf{e}_{uv}, \mathbf{h}_{v'})) \tag{1.5.4}$$

where γ_m is a margin hyper-parameter that implies that the distance between \mathbf{h}_u translated by \mathbf{e}_{uv} to \mathbf{h}_v should be at least γ_m smaller than the distance from $\mathbf{h}_u + \mathbf{e}_{uv}$ to $\mathbf{h}_{v'}$.

1.5.2. Motivating modulators: Applied examples

In most tasks mentioned so far, the input graph and the computation graph (i.e., the graph used for deciding which messages to pass in the GNN equation) were considered identical. However, in many cases of interest, this is not sufficient, and *modulating* the computation graph can provide signal for a more efficient **algorithm**, or is more aligned with the **application**, as in the case of supporting long-range molecular interactions.

Exploring the derivation of improved computation graphs for GNNs is hence a critical problem—one that arguably lies at the heart of several of the contributions of this thesis. Within the framework proposed by Figure 1, it corresponds to the **Modulator** component.

Graph neural networks and geometric deep learning have been extensively used in scientific tasks, and are poised to further accelerate scientific discovery [66]. I will present here several motivating real-world examples from biology and chemistry, on which I have worked concurrently or prior to the core works presented in this thesis, that show the utility of modulating the computation graph.

1.5.2.1. Large-scale learning on graphs.

Effectively and efficiently deploying graph neural networks (GNNs) at scale remains one of the most challenging aspects of graph representation learning. Many powerful solutions have only ever been validated on comparatively small datasets, often with counter-intuitive outcomes—a barrier which has been broken by the Open Graph Benchmark Large-Scale Challenge (OGB-LSC). As an equal first-author contributor within a team of researchers from DeepMind, I have entered the OGB-LSC with two large-scale GNNs: a deep transductive

node classifier powered by bootstrapping, and a very deep (up to 50-layer) inductive graph regressor regularised by denoising objectives. Our models achieved an award-level (top-3) performance on both the MAG240M and PCQM4M benchmarks within the LSC. In doing so, we demonstrate evidence of scalable self-supervised graph representation learning, and utility of very deep GNNs—both very important open issues. We will focus on PCQM4M here, as our findings on the MAG240M academic graph dataset are in many regards analogous.

For PCQM4M, the task is *graph regression*: estimate the HOMO-LUMO gap of each molecule. This is an important quantum-chemical property for which ground-truth labels are obtained through expensive DFT (density functional theory) calculations, possibly taking several hours per molecule. It is believed that machine learning models, such as GNNs over the molecular graph, may obtain useful approximations to the DFT at only a fraction of the computational cost, if provided with sufficient training data [13].

While several factors contributed to the superior performance of our model [67], in the end, by far, the most impactful method for our GNN regressor on PCQM4M has been Noisy Nodes [68], and our results largely echo the findings therein. The main observation of Noisy Nodes is that very deep GNNs can be strongly regularised by appropriate denoising objectives. Noisy Nodes perturbs the input node or edge features in a pre-specified way, then requires the decoder to reconstruct the un-perturbed information from the GNN’s latent representations.

Similarly, for this task, where very specific constraints apply for valid inputs—given by valences, for example—we can modulate the molecular graph by randomly corrupting the nodes or the edges, to obtain an improved algorithm that optimizes for the property prediction task, as well as for the Noisy Nodes objective. More specifically, we randomly replace atom or bond types, or, when appropriate, we modify the 3D atom coordinates, aiming to then reconstruct the original node/edge properties. Consequently, we note that, while most prior works on GNN modelling seldom use more than eight steps of message passing [69], our method allows us to observe monotonic improvements of deeper GNNs on this task, all the way to 32 layers when the validation performance plateaus, allowing for longer-range reasoning and better final performance. We attribute this benefit to the random corruptions to the computational graph’s edge types, making the model less prone to overfitting to message passing over molecular graphs.

1.5.2.2. Molecular interactions.

The tasks we consider next have one common structural bias, portrayed in Figure 8: the ground-truth label we need to predict is based on pairwise interactions (drug-drug, drug-protein or amino acids in antibody-antigen). In both cases, we will model these pairwise interactions by exploiting *bipartite computational graphs*, allowing for explicitly

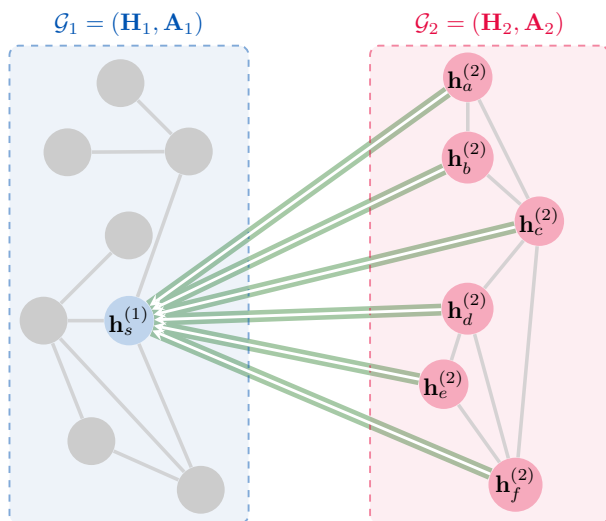


Figure 8 – The high-level overview of molecular interaction tasks – we form a bipartite graph with the two entities or types of entities of interest, allowing for all pairwise interactions: atoms of drug pairs, drug-protein and amino acids in antibody-antigen complex.

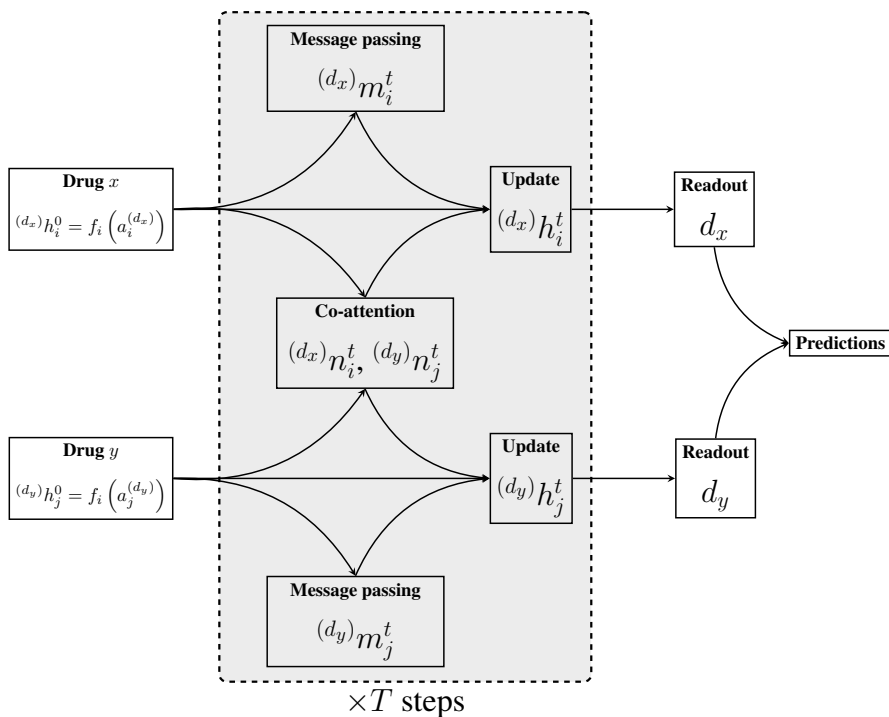


Figure 9 – A high-level overview of our drug-drug side-effect prediction model. The next-level features of atom i of drug x , $(d_x)h_i^t$, are derived by combining its *input features*, $(d_x)h_i^{t-1}$, its *inner message*, $(d_x)m_i^t$, computed using message passing, and its *outer message*, $(d_x)n_i^t$, computed using co-attention over the second drug, d_y .

modelling the cross-influence between the relevant pairs.

Drug-drug interactions. Diseases are often caused by complex biological processes which cannot be treated by individual drugs and thus introduce the need for concurrent use of multiple medications. Similarly, drug combinations are needed when patients suffer from multiple medical conditions.

I’ll present my work on drug combinations, found in the form of a link prediction task. The aim is to predict which side-effect will occur for a given co-administration of two drugs by classifying the edge (previously referred to as relation) type between drug pairs (which we called entities).

To achieve the state-of-the-art performance on this task, we heavily leverage bipartite computational graphs. As the individual molecular structures of the drugs are known, we design our GNNs that classify drug pairs to periodically exchange information between the two drugs in a pair, by performing message passing over the *full bipartite graph*, connecting every atom in one drug to every atom in the other. Our ablation studies clearly indicate that the model’s performance significantly degrades in the absence of these drug-drug exchange layers.

This proposed approach aims to follow domain intuition of what guides the treatment outcome, forming a graph that includes all possible underlying interactions. Specifically, one reason for side-effects when administering two drugs together is that one drug contains a functional group that reacts with a functional group from the other drug. By forming a bipartite graph between the atom representations of the two chemical graphs, the model can predict side-effects by jointly reasoning over functional groups from the two molecules. Message passing provides a robust mechanism for encoding within-drug representations of atoms. For learning an appropriate *joint* drug-drug representation, we allow atoms to interact *across* drug boundaries via a *co-attentional mechanism*, as in Figure 9. This approach [64] is empirically better than models considering drugs individually, sharing information late in the computation graph and even (complementary) approaches that consider additional data sources, such as interaction with proteins.

Antibody-antigen interactions. Antibodies are proteins in the immune system which bind to antigens to detect and neutralise them. The binding sites in an antibody-antigen interaction are known as the paratope and epitope, respectively, and the prediction of these regions is key to vaccine and synthetic antibody development. More traditionally, this problem was attempted by analyzing the antibody and the antigen in isolation, lacking complementary information that leads to structural changes.

Similarly to the drug-drug interaction example, we will form a bipartite computational graph using the antibody and the antigen amino acids. Predicting the binding sites of antibodies and antigens will thus be a *node classification* task: an amino acid participates in the binding if it connects to any amino acids from the other subset. We present two methods of modeling the interaction:

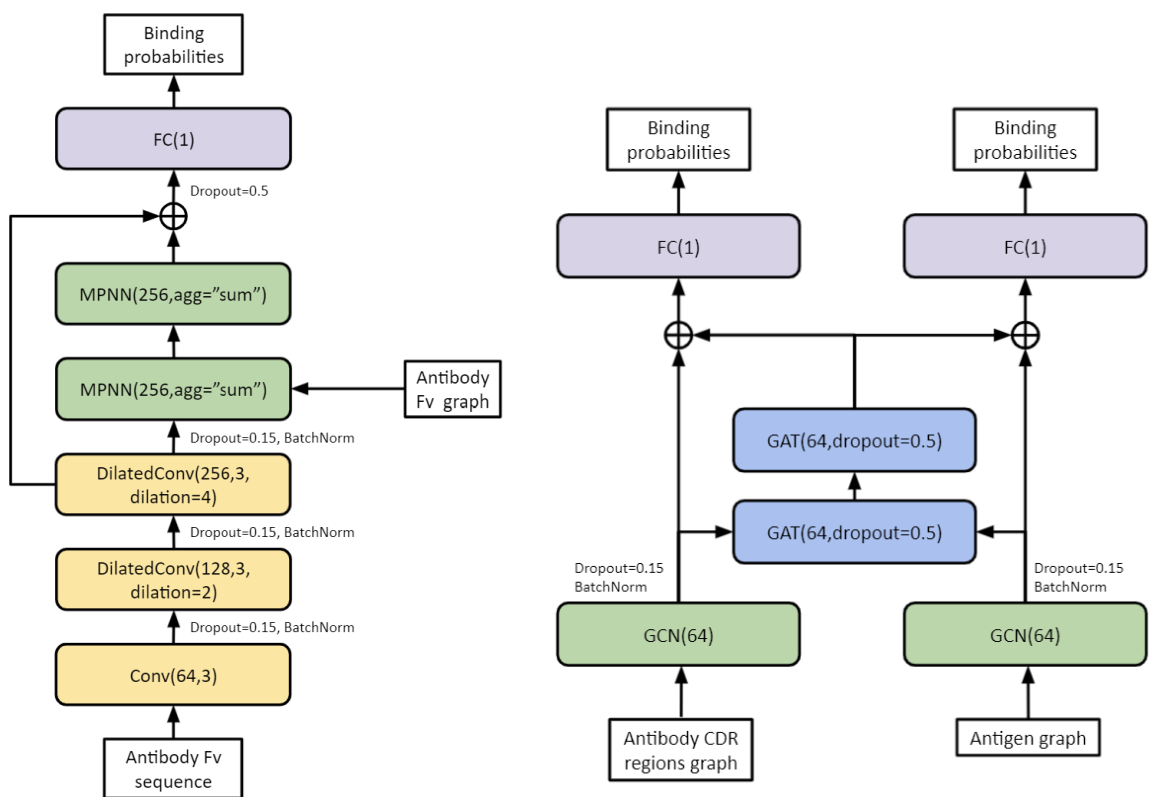


Figure 10 – The Para-EPMP architecture (**left**) and the Epi-EPMP multitask architecture (**right**). The output feature dimension for each layer is the first term in the bracket. For the convolutional layers, the second term is the kernel size.

- In [61], contrary to prior art, we argue that paratope and epitope predictors require asymmetric treatment as paratopes are highly sequential and can be predicted well in isolation, while epitopes are structural in nature and are inherently conditioned by the paratope. We propose distinct neural message passing architectures that are geared towards the specific aspects of paratope and epitope prediction, respectively. To combine information from the two representations, we allow the residue representations to interact across the antibody-antigen boundary. For this purpose, we used graph attention networks [15] over the fully connected bipartite graph between the antigen residues and the antibody residues (i.e. each antigen residue is attending over every antibody residue, and vice-versa) [70]. Once attention is performed, the network then predicts both the paratope and epitope residues at once. This paratope prediction is solely for aiding the epitope prediction as, without such a multi-task objective, antibody residues would remain unlabelled, and hence the GAT would have to learn in an unsupervised manner which antibody residues are the most important in the attention layers. Instead, by leveraging multitasking, the network is given explicit cues as to which parts of the antibody are actually relevant to the binding site.

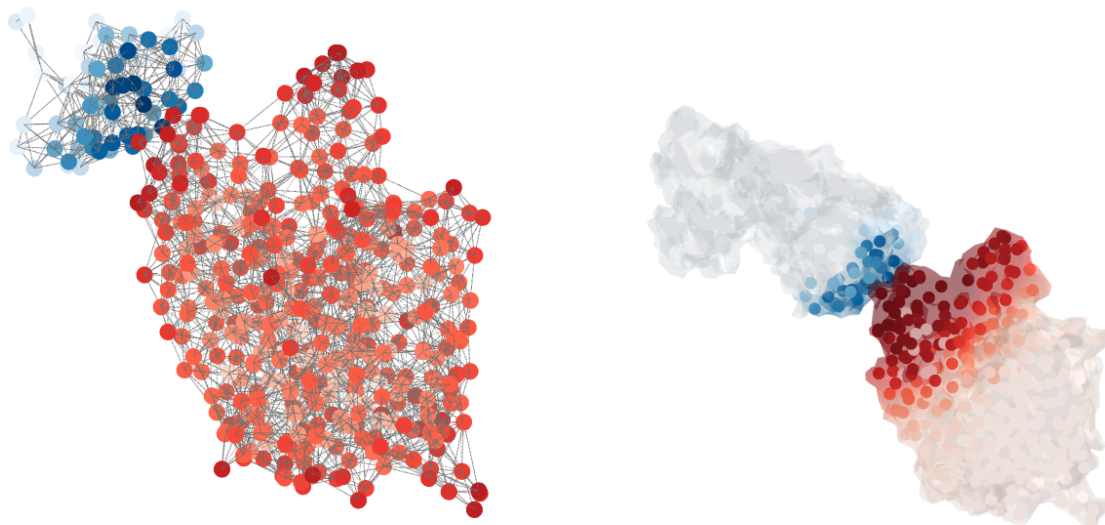


Figure 11 – Qualitative example on the 4jr9 pdb. We plot on the residuals the binding probability as increasing intensity colors: blue for the antibody and red for the antigen. The left figure shows the results of the $E(n)$ -EPMP on the residual graph, while the right sides displays the predictions of the surface-based method.

Furthermore, it allows for knowledge transfer between the two tasks, regularising the epitope representations – similarly to Subsection 1.5.2.1, modulating the graph lends itself to an additional objective for the optimization algorithm.

- Lastly, we extend the architectures from the previous tasks by integrating insights from geometric deep learning based on insights about the data, such as invariance to translations, rotations, and reflections and the importance of the outer surface of the molecules. To this end, in [62] we integrate equivariant GNNs and surface-based methods, allowing us to consider point-cloud and mesh representations. As shown in Figure 11, the latter leads to better localized predictions.

1.6. Summary

In the previous three examples, I demonstrated how even *simple* modulations of the computational graph—such as random corruptions and full bipartite interactions—can make a significant difference to the downstream performance of GNNs in relevant biological and chemical tasks. These benefits proved robust across various levels of training data abundance, as well as input feature diversity.

Armed with this intuition, we are ready to dive into the core contributions of this thesis. The computation graph modulations induced in the following chapters will, in contrast, be *nontrivial*:

- In EGP [22], we demonstrate the utility of *sparse expander graphs* as a general-purpose computational template. Sparse expander graphs are exotic and elusive mathematical objects: they have extremely rich theoretical underpinnings, and only few ways of constructing them are known to mathematics.
- In ECG [23], we demonstrate the utility of improving homophily in the computational graph, via learned *weak classifiers*—both pointwise and structural. This procedure is conceptually elegant, and easy to motivate, but nontrivial to stabilise, as it relies on (at least) a two-phase method, and the success of the second phase critically depends on the power of the classifiers trained in the first phase.
- In XLVIN [24], we demonstrate how computational graphs can be dynamically inferred on-the-fly by leveraging a *learned transition model*, coupled with *breadth-first search*. Such a setup is often necessary when deploying GNN processors in reinforcement learning, as the agent usually does not know *a priori* what the environment transition dynamics are: it must infer these through interacting with the environment. The learned transition model hence indicates the agent’s best current guess of these transition dynamics, from which relevant local computational graphs can be obtained via sampling and search.
- In Equivariant MuZero [25], we demonstrate how to improve the low-data performance and robustness of state-of-the-art reinforcement learning agents by taking into account *geometric constraints*. Specifically, by exploiting the symmetries assumed in the RL environment, we are able to define and modulate specialised computational graphs between various *views* of that environment—for example, all possible 2D rotations of it. The computational graph is defined in a way that respects the assumed symmetries of the environment, and leads to significant empirical gains in data efficiency, along with provable equivariance properties of the underlying planning algorithm—in the case of MuZero, this is Monte Carlo tree search (MCTS).

Chapter 2

Graph Neural Networks for Long-Range Interaction: Prologue to the first article

2.1. Article Details

Expander Graph Propagation (EGP). Andreea Deac, Marc Lackenby, Petar Veličković. The paper was published at Learning on Graphs 2022 and received the Best Paper Award at NeurIPS 2022 Frontiers in Graph Learning workshop.

Personal contribution: Marc initially suggested that expander graphs could have highly favourable properties for addressing the over-squashing problem, by definition. Based on this suggestion, I have proposed, shaped, implemented and thoroughly evaluated a viable machine learning method—EGP—based on this suggestion, and advice from Petar. Marc and Petar also contributed the foundational theory behind EGP, with Marc proving the Theorem about the impossibility of building infinite positively-curved sparse computational graphs without bottlenecks. All authors wrote and improved the manuscript.

2.2. Context

Graph neural networks have proved successful in many applications, drug discovery [55], transportation networks [56] and mathematical advances [58] being just a few of them. At the same time, some of their limitations were brought to light, many of them regarding their scalability. One class of problems comes from the way the propagation is designed: GNNs aggregate information from 1-hop neighbourhoods by using one GNN layer. To aggregate information from k -hop neighbourhoods, k layers need to be applied. Problems arise when not enough layers are used (under-reaching [33]), when the signals are being too homogenised (over-smoothing [32]) or when a layer is required to summarise signals from an exponentially large receptive field (over-squashing [4]). The latter, particularly important for tasks where long-range interaction is required or bottlenecks are present, is the topic of EGP. In this

work, we use expander graphs, a mathematical object known to have good propagation properties—such as low diameter, high Cheeger constant, and favourable mixing time—as a template to send information across the graph. Thus, all nodes will be connected to each other in at most $\log(N)$ steps, where N is the number of nodes in the graph, ensuring that nodes that are far apart in the input graph can now easily communicate.

Thanks to their qualities, the expander graphs were also used concurrently in Shirzad et al. [71], presenting an approach that leverages them to improve the scalability and accuracy of graph Transformers.

2.3. Modulator

In EGP [22], we demonstrate the utility of *sparse expander graphs* as a general-purpose computational template. Sparse expander graphs are exotic and elusive mathematical objects: they have extremely rich theoretical underpinnings, and only few ways of constructing them are known to mathematics. In this case, the modulator samples an appropriate expander graph, and substitutes it as the computation graph for certain GNN layers.

2.4. Contributions and Research Impact

EGP presents using expander graphs as global templates for propagation in GNNs. In order to do this, we first propose an efficient way to construct a family of expander graphs. Using them we are able to show empirical improvements on a theoretical task called `TreeNeighboursMatch` [4]—which was previously introduced to rigorously study over-squashing—as well as on practical real-world tasks from the Open Graph Benchmark [60]. Moreover, in EGP we extend on the findings of Topping et al. [72], where, under some conditions, negatively curved edges were linked to over-squashing. Specifically, we observe that expander graphs can never be sufficiently negatively curved to trigger those conditions and, even more, that negatively curved edges are required to have sparse communications without bottlenecks. This highlights negatively-curved edges as important area of future theoretical study for the over-squashing effect.

Following this work, expanders graphs were identified as a potential solution for sparsifying attention in graph transformers [73], Black et al. [74] described the connection between expansion properties and effective resistance and Giovanni et al. [75] showed how the over-squashing effect limits the expressive power of GNNs. The latter, which I was also involved in as a co-author, was developed as a direct consequence of the conclusions of EGP, and its framework using *commute times* provides strong theoretical evidence for the downstream utility of expander graphs as a computational template for GNNs.

Chapter 3

Expander Graph Propagation

Deploying graph neural networks (GNNs) on whole-graph classification or regression tasks is known to be challenging: it often requires computing node features that are mindful of both local interactions in their neighbourhood and the global context of the graph structure. GNN architectures that navigate this space need to avoid pathological behaviours, such as bottlenecks and oversquashing, while ideally having linear time and space complexity requirements. In this work, we propose an elegant approach based on propagating information over *expander graphs*. We leverage an efficient method for constructing expander graphs of a given size, and use this insight to propose the EGP model. We show that EGP is able to address all of the above concerns, while requiring minimal effort to set up, and provide evidence of its empirical utility on relevant graph classification datasets and baselines in the Open Graph Benchmark. Importantly, using expander graphs as a template for message passing necessarily gives rise to negative curvature. While this appears to be counterintuitive in light of recent related work on oversquashing, we theoretically demonstrate that negatively curved edges are likely to be **required** to obtain scalable message passing without bottlenecks. To the best of our knowledge, this is a previously unstudied result in the context of graph representation learning, and we believe our analysis paves the way to a novel class of scalable methods to counter oversquashing in GNNs.

3.1. Introduction

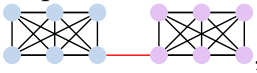
Graph neural networks (GNNs) are a flexible class of models for learning representations over graph-structured data [76]. Their versatility [14, 15, 13] and generality [18, 17] has made them a very attractive approach, leading to considerable application in areas as diverse as virtual drug screening [55], traffic prediction [56], combinatorial chip design [57] and pure mathematics [59, 58].

Most GNNs rely on repeatedly propagating information between neighbouring nodes in the graph. This is commonly expressed in the *message passing* [13] paradigm: nodes

send vector-based *messages* to each other along the edges of the graph, and nodes update their representations by *aggregating* all the messages sent to them, in a permutation-invariant manner. Under many industrially-relevant tasks, this paradigm is very potent, often allowing for highly scalable model variants [77–79].

However, in many areas of scientific interest, purely local interactions are likely insufficient. Among the principal graph tasks, *graph classification* is perhaps most ripe with such situations: to meaningfully attach a label to a graph, in many cases it is insufficient to treat graphs as “bags of nodes”. For example, when classifying a molecule for its potency as a candidate drug [55], the label is driven by complex substructure interactions in the molecule [80], rather than a naïve sum of atom-level effects.

Accordingly, GNNs deployed in this regime need to update node features in a manner that is mindful of the *global* properties of the graph. It quickly became apparent that it is often inadequate to merely stack more message passing layers over the input graph. In fact, for many graph classification tasks, such approaches may be weaker than discarding the graph structure altogether [81, 82]. Now, it is well-understood that stacking many local layers leaves GNNs vulnerable to pathological behaviours such as oversquashing [4]. Intuitively, oversquashing occurs when nodes need to store quantities of information that are *exponentially* increasing with model depth [4, Section 5]. Such nodes often arise in the vicinity of *bottlenecks* in a graph—small collections of edges which are responsible for carrying representations between large groups of nodes. One typical example of such a bottleneck

can be found in a *barbell graph* , where the red edge is under significant representational pressure to transport information between the two communities.

Within this space, we are interested in proposing a method that satisfies *four* desirable criteria: **(C1)** it is capable of propagating information *globally* in the graph; **(C2)** it is *resistant* to the oversquashing effect and does not introduce bottlenecks; **(C3)** its time and space complexity remain *subquadratic* (tighter than $O(|V|^2)$ for sparse graphs); and **(C4)** it requires *no dedicated preprocessing* of the input. Satisfying all four of these criteria simultaneously is challenging, and we will survey many of the popular approaches in the next section—demonstrating ways in which they fail to meet some of them.

In this paper, we identify *expander graphs* as very attractive objects in this regard. Specifically, they offer a family of graph structures that are fundamentally *sparse* ($|E| = O(|V|)$), while having *low diameter*: thus, any two nodes in an expander graph may reach each other in a short number of hops, eliminating bottlenecks and oversquashing (see Figure 12). Further, we will demonstrate an efficient way to construct a family of expander graphs (leveraging known theoretical results on the *special linear group*, $SL(2, \mathbb{Z}_n)$). Once an expander graph of appropriate size is constructed, we can perform a certain number of GNN *propagation*

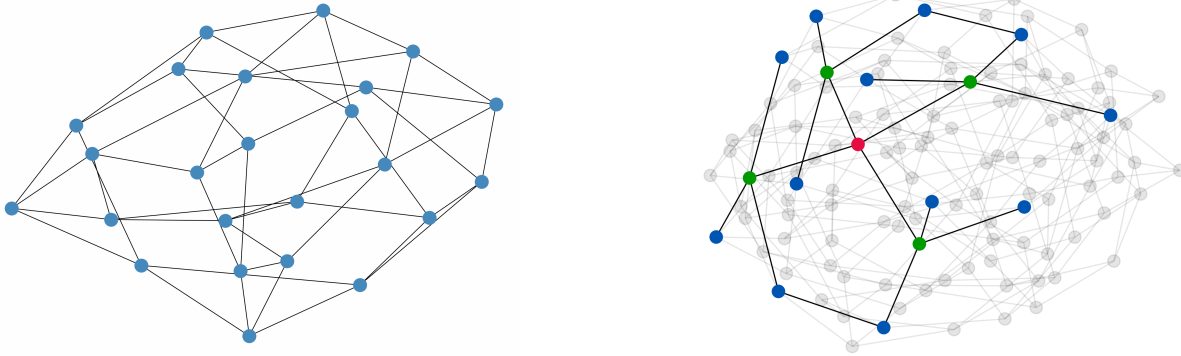


Figure 12 – Left: The Cayley graph of $SL(2, \mathbb{Z}_3)$, constructed using our method. It has $|V| = 24$ nodes and it is 4-regular (implying $|E| = 2|V|$), hence it is sparse. Despite its sparsity, it is highly interconnected: any node is reachable from any other node by no more than 4 hops. Hence, it can serve as a strong “template” for globally propagating node features with a GNN. **Right:** The Cayley graph of $SL(2, \mathbb{Z}_5)$, constructed in an analogous way (with $|V| = 120$ nodes). A 2-hop neighbourhood of one node (in red) is highlighted, demonstrating its tree-like local structure.

steps over its structure to globally distribute the nodes’ features. Accordingly, we name our method *expander graph propagation* (**EGP**).

A key contribution of our work extends the implications of prior art on oversquashing via curvature analysis [72]. According to [72], negatively curved edges are causing the oversquashing effect—yet, counterintuitively, the edges of the expander graphs we construct will *always be negatively curved!* We prove, however, that our expanders can never be sufficiently negatively curved to trigger the conditions necessary for the results in [72] to be applicable, and show that the existence of negatively curved edges might in fact be **required** in order to have sparse communication without bottlenecks.

3.2. Related work

We begin with a survey of the many prior approaches to handling global context in graph representation learning, evaluating them carefully against our four desirable criteria (**C1–C4**; cf. Table 2). This list is by no means exhaustive, but should be indicative of the most important directions.

Stacking more layers. As already highlighted, one way to achieve global information propagation is to have a deeper GNN. In this case, we are capable of satisfying (**C1**) and (**C4**)—no dedicated preprocessing is needed. However, depending on the graph’s diameter, we may need up to $O(|V|)$ layers to cover the graph, leading to quadratic complexity (violating (**C3**)) and introducing a vulnerability to bottlenecks (**C2**), as theoretically and empirically demonstrated in [4].

Table 2 – A summary of principal approaches to handling global context in graph representation learning (Section 3.2). “(✓)” indicates that a criterion *may* be satisfied, depending on the method’s tradeoffs. Our proposal, the expander graph propagation (EGP) method, satisfies all four criteria.

Approach	(C1) (global prop.)	(C2) (no bottlenecks)	(C3) (subquadratic)	(C4) (no dedicated preproc.)
GNNs	✗	✗	✓	✓
Sufficiently deep GNNs	✓	✗	✗	✓
Master node [83, 13]	✓	✗	✓	✓
Fully connected [4, 84–88]	✓	✓	✗	✓
Feature aug. [89–94]	✓	(✓)	(✓)	✗
Graph rewiring [95, 72, 96]	✓	✓	✓	✗
Hierarchical MP [97–102]	✓	✓	(✓)	✗
EGP (ours)	✓	✓	✓	✓

Master nodes. An attractive approach to introducing global context is to introduce a *master node* to the graph, and connect it to all of the graph’s nodes. This can be done either explicitly [13] or implicitly, by storing a “global” vector [83]. It trivially reduces the graph’s diameter to 2, introduces $O(1)$ new nodes and $O(|V|)$ new edges, and requires no dedicated preprocessing, hence it satisfies **(C1, C3, C4)**. However, these benefits come at the expense of introducing a bottleneck in the master node: it has a very challenging task (especially when graphs get larger) to continually incorporate information over a very large neighbourhood in a useful way. Hence it fails to satisfy **(C2)**.

Fully connected graphs. The converse approach is to make *every* node a master node: in this case, we make all pairs of nodes connected by an edge—this was initially proposed as a powerful method to alleviate oversquashing by [4]. This strategy proved highly popular in the recent surge of Graph Transformers [85, 86, 88], and is common for GNNs used in physical simulation [84] or reasoning [87] tasks. The graph’s diameter is reduced to 1, no bottlenecks remain, and the approach does not require any dedicated preprocessing. Hence **(C1, C2, C4)** are trivially satisfied. The main downside of this approach is the introduction of $O(|V|^2)$ edges, which means **(C3)** can never be satisfied—and this approach will hence be prohibitive even for modestly-sized graphs.

Feature augmentation. An alternative approach is to provide additional features to the GNN which directly identify the structural role each node plays in the graph [89]. If done properly (i.e., if the computed features are relevant to the target), this can drastically improve expressive power. Hence, in theory, it is possible to satisfy **(C1)** while not violating **(C2, C3)**. However, computing appropriate features requires either specific domain knowledge, or appropriate pre-training [90–94], in order to obtain such embeddings. Hence all of these gains come at the expense of failing to satisfy **(C4)**.

Graph rewiring. Another promising line of research involves modifying the edges of the original graph to alleviate bottlenecks. Popular examples of this approach involve using diffusion [95]—which diffuse additional edges through the application of kernels such as the personalised PageRank, and stochastic discrete Ricci flows [72]—which surgically modify a small quantity of edges to alleviate the oversquashing effect on the nodes with negative Ricci curvature. Recent concurrent work [96] also uses constructions inspired by expander graphs to randomly locally rewire a given input graph. If realised carefully, such approaches will not deviate too far from the original graph, while provably alleviating oversquashing; hence it is possible to satisfy **(C1, C2, C3)**. However, this comes at a cost of having to examine the input graph structure, with methods that do not necessarily scale easily with the number of nodes. As such, dedicated preprocessing is needed, failing to satisfy **(C4)**.

Hierarchical message passing. Lastly, going beyond modifying the edges, it is also possible to introduce additional *nodes* in the graph—each of them responsible for a particular *substructure* in the graph¹. If done carefully, it has the potential to drastically reduce the graph’s diameter while not introducing bottlenecked nodes (hence, allowing us to satisfy **(C1, C2)**). However, in prior work, a cost has to be paid for this, usually in the need for dedicated preprocessing. Prior proposals for hierarchical GNNs that remain scalable require a dedicated pre-processing step [97–99], sometimes coupled with domain knowledge [99]—thus failing to satisfy **(C4)**. In addition, such methods may require adding prohibitively large numbers of substructures [100, 101] or expensive pre-computation, e.g. computing the graph Laplacian eigenvectors [102]. This might make even **(C3)** hard to satisfy.

We remark that our work is not the first to study expander graph-related topics in the context of GNNs. Specifically, the ExpanderGNN [103] leverages expander graphs over neural network weights to sparsify the update step in GNNs. This is a direct application of Deep Expander Networks [104], which studied such constructs over CNNs. With respect to our contributions, neither of these cases discuss expanders in the context of the computational graph for a GNN, nor attempt to propagate messages over such a structure. Further, neither satisfy all four of our desired criteria **(C1–C4)**.

3.3. Theoretical background

We now dedicate our attention to the key theoretical results over expander graphs, which will allow EGP to have favourable properties and be efficiently precomputable.

1. Master nodes are a special case: a single node is responsible for a “substructure” spanning the entire graph.

Definition 3.3.1. For a finite connected graph $G = (V(G), E(G))$, we consider functions $f: V(G) \rightarrow \mathbb{R}$. The *Laplacian* $Lf: V(G) \rightarrow \mathbb{R}$ of such a function is defined to be

$$Lf(v) = \deg(v)f(v) - \sum_{vw \in E(G)} f(w),$$

where $\deg(v)$ is the degree of the vertex v .

The mapping $L: \mathbb{R}^{V(G)} \rightarrow \mathbb{R}^{V(G)}$ sending a function f to its Laplacian Lf is a linear transformation. It is not hard to show [105] that L is symmetric with respect to the standard basis for $\mathbb{R}^{V(G)}$ and positive semi-definite and hence has non-negative real eigenvalues

$$0 = \lambda_0(G) < \lambda_1(G) \leq \lambda_2(G) \leq \dots$$

The smallest eigenvalue is 0 and its associated eigenspace consists of the constant functions (assuming G is connected). The smallest positive eigenvalue, $\lambda_1(G)$, is central to the definition of expander graphs, as the next definition shows.

Definition 3.3.2. An infinite collection $\{G_i\}$ of finite connected graphs is an *expander family* if there is a constant $c > 0$ such that for all G_i in the collection, $\lambda_1(G_i) \geq c$.

Expander families [106–108] have many remarkable and useful properties, particularly when there is a uniform upper bound on the degree of the vertices of G_i .

Definition 3.3.3. Let G be a finite graph. For $A \subset V(G)$, its *boundary* ∂A is the collection of edges with one endpoint in A and one endpoint not in A . The *Cheeger constant* $h(G)$ is defined to be

$$h(G) = \min \left\{ \frac{|\partial A|}{|A|} : A \subset V(G), 0 < |A| \leq |V(G)|/2 \right\}.$$

Thus, having a small Cheeger constant is equivalent to the graph having a ‘bottleneck’, in the sense that there is a collection of edges ∂A that, when removed, disconnects the vertices into two sets (A and its complement, $V(G) \setminus A$), with the property that the sizes of A and its complement are significantly larger than the size of ∂A .

Expander families can be reinterpreted using Cheeger constants, as follows (see, e.g., [109–112]):

Theorem 3.3.4. *Let $\{G_i\}$ be an infinite collection of finite connected graphs with a uniform upper bound on their vertex degrees. Then the following are equivalent:*

- (1) $\{G_i\}$ is an expander family;
- (2) there is a constant $\epsilon > 0$ such that for all graphs in the collection, $h(G_i) \geq \epsilon$.

Hence, expander graphs have higher Cheeger constants and will hence experience less severe problems arising due to bottleneck edges. The following result is one of the many useful properties of expander families, and it concerns their *diameter*. It was proved by Mohar [113, Theorem 2.3]. See also [110].

Theorem 3.3.5. *The diameter $\text{diam}(G)$ of a graph G satisfies*

$$\text{diam}(G) \leq 2 \left\lceil \frac{\Delta(G) + \lambda_1(G)}{4\lambda_1(G)} \log(|V(G)| - 1) \right\rceil,$$

where $\Delta(G)$ is the maximal degree of any vertex of G . Hence, if $\{G_i\}$ is an expander family of finite graphs with a uniform upper bound on their vertex degrees, then there is a constant $k > 0$ such that for all graphs in the family,

$$\text{diam}(G_i) \leq k \log V(G_i).$$

Therefore, if we want to globally propagate information over an expander graph which has $|V|$ nodes, we only need $O(\log |V|)$ propagation steps to do so—yielding subquadratic complexity.

We showed that expanders will experience less severe problems arising due to bottleneck edges, with favourable propagation qualities. What is missing is an efficient method of constructing an expander of (roughly) $|V|$ nodes. To demonstrate such a method, we leverage known results from group theory.

Definition 3.3.6. A group (Γ, \circ) is a set Γ equipped with a *composition* operation $\circ : \Gamma \times \Gamma \rightarrow \Gamma$ (written concisely by omitting \circ , i.e. $g \circ h = gh$, for $g, h \in \Gamma$), satisfying the following axioms:

- (*Associativity*) $(gh)l = g(hl)$, for $g, h, l \in \Gamma$.
- (*Identity*) There exists a unique $e \in \Gamma$ satisfying $eg = ge = g$ for all $g \in \Gamma$.
- (*Inverse*) For every $g \in \Gamma$ there exists a unique $g^{-1} \in \Gamma$ such that $gg^{-1} = g^{-1}g = e$.

A group is hence a natural construct for reasoning about transformations that leave an object invariant (unchanged). Further, we define a relevant notion of a group’s generating set:

Definition 3.3.7. Let Γ be a group. A subset $S \subseteq \Gamma$ is a *generating set* for Γ if it can be used to “generate” all of Γ via composition. Concretely, any element $g \in \Gamma$ can be expressed by composing elements in the generating set, or their inverses; that is, we can express $g = s_1^{\pm 1} s_2^{\pm 1} s_3^{\pm 1} \dots s_{n-1}^{\pm 1} s_n^{\pm 1}$ for $s_i \in S$.

Now we are ready to define a Cayley graph of a group w.r.t. its generating set.

Definition 3.3.8. Let Γ be a group with a finite generating set S . Then the associated *Cayley graph* $\text{Cay}(\Gamma; S)$ has vertex set Γ and it has an edge $g \rightarrow gs$ for each $g \in \Gamma$ and each $s \in S$. We say that s is the *label* on this edge. This is a potentially non-simple graph, as it may have edges with both endpoints on the same vertex and it may have multiple edges between a pair of vertices. In particular, when s has order 2, then we view the edge $g \rightarrow gs$ and the edge $gs \rightarrow gs^2 = g$ as distinct edges.

Note that the degree of each vertex of a Cayley graph $\text{Cay}(\Gamma; S)$ is $2|S|$. This is because each vertex g is joined by edges to gs and gs^{-1} for each $s \in S$. Thus, we shall be particularly

interested in the case where there is a uniform upper bound on $|S|$. The specific group we use for EGP is as follows.

For each positive integer n , the *special linear group* $\text{SL}(2, \mathbb{Z}_n)$ denotes the group of 2×2 matrices with entries that are integers modulo n and with determinant 1. One of its generating sets is:

$$S_n = \left\{ \begin{pmatrix} 1 & 1 \\ 0 & 1 \end{pmatrix}, \begin{pmatrix} 1 & 0 \\ 1 & 1 \end{pmatrix} \right\}.$$

Central to our constructions is the following important result.

Theorem 3.3.9. *The family of Cayley graph $\text{Cay}(\text{SL}(2, \mathbb{Z}_n); S_n)$ forms an expander family.*

The proof uses a result of Selberg [114] who showed that the smallest positive eigenvalue of the Laplacian of certain hyperbolic surfaces is at least $3/16$. One can use this to produce a lower bound on the first eigenvalue of the Laplacian on $\text{Cay}(\text{SL}(2, \mathbb{Z}_n); S_n)$. Full proofs are given in [108, 107].

Lastly, it is useful to state a known result: the number of nodes of $\text{Cay}(\text{SL}(2, \mathbb{Z}_n); S_n)$ is:

$$|V(\text{Cay}(\text{SL}(2, \mathbb{Z}_n); S_n))| = n^3 \prod_{\text{prime } p|n} \left(1 - \frac{1}{p^2}\right), \quad (3.3.10)$$

hence, it is of the order of $O(n^3)$. We now study the local properties of Cayley graphs in detail.

3.4. Local structure of the Cayley graphs, and the utility of negative curvature

Recent work [72] has suggested that the local structure of the graph G underlying a GNN may play an important role in the way that information propagates around G . In particular, various notions of ‘Ricci curvature’ such as Forman curvature [115], Ollivier curvature [116, 117] and balanced Forman curvature [72] have been examined. These are all local quantities, in the sense that they depend on the structure of the graph within a small neighbourhood of each edge. In this section, we will therefore examine the local structure of the Cayley graphs $G_n = \text{Cay}(\text{SL}(2, \mathbb{Z}_n); S_n)$.

The various notions of curvature given above are defined for each e of the graph G . Since, as defined by [72], the balanced Forman curvature of an edge depends only on local structures (i.e. triangles and squares) around that edge, they can be determined by only observing the immediate 2-hop surrounding of that edge. Formally, for an edge e of a graph G , let $N_2(e)$ be the induced subgraph with vertices that are at most two hops away from at least one endpoint of e . Then the curvature of e only depends on the isomorphism type of $N_2(e)$. More specifically, if e and e' are edges in possibly distinct graphs, and there is a

graph isomorphism between $N_2(e)$ and $N_2(e')$ that sends e to e' , then this guarantees that the curvatures of e and e' are equal.

This situation arises prominently in the Cayley graphs that we are considering, as follows.

Proposition 3.4.1. *Let s be one of*

$$\begin{pmatrix} 1 & 1 \\ 0 & 1 \end{pmatrix}, \quad \begin{pmatrix} 1 & 0 \\ 1 & 1 \end{pmatrix}.$$

Let $n, n' > 18$ and let e and e' be s -labelled edges in G_n and $G_{n'}$. Then there is a graph isomorphism between $N_2(e)$ and $N_2(e')$ taking e to e' .

We prove Proposition 3.4.1 in Appendix A.1. This immediately allows us to characterise the balanced Forman curvature and Ollivier curvature for all of the Cayley graphs we generate:

Proposition 3.4.2. *The balanced Forman curvatures $\text{Ric}(n)$, and the Ollivier curvatures $\kappa(n)$ of all edges of Cayley graphs G_n are given by:*

$$\text{Ric}(n) = \begin{cases} 0 & \text{if } n = 2 \\ -1/4 & \text{if } n = 3 \\ -1/2 & \text{if } n = 4 \\ -1 & \text{if } n \geq 5, \end{cases} \quad \kappa(n) = \begin{cases} 0 & \text{if } n = 2 \\ -1/8 & \text{if } n = 3 \\ -1/4 & \text{if } n = 4 \\ -3/8 & \text{if } n = 5 \\ -1/2 & \text{if } n \geq 6. \end{cases}$$

PROOF. Proposition 3.4.1 implies that the balanced Forman and Ollivier curvatures are all equal for $n > 18$. Their values for $2 \leq n \leq 19$ can all be empirically computed, and are given as above. \square

Prior work [72] suggests it is preferable for GNNs to operate on graphs with positive Ricci curvature, whereas our graphs G_n ($n > 2$) all have negative Ricci curvature. However, we contend that negative Ricci curvature is not in itself an impediment to efficient propagation around a GNN. Indeed, it was shown in [72, Theorem 4] that poor propagation arises when the balanced Forman curvature is close to -2 , specifically if it is at most $-2 + \delta$ for some $\delta > 0$. Here, δ is required to satisfy certain inequalities. But, with certainty, $\delta = 1$ can *never* be satisfied in the hypotheses of [72, Theorem 4].

Furthermore, positive Ricci curvature may have *downsides* when used for GNNs. One significant downside can be derived using the main result of [118], which says that the three properties of expansion, sparsity and non-negative Ollivier curvature are incompatible, in the following sense.

Theorem 3.4.3. *For any $\delta > 0$ and $\Delta > 0$, there are only finitely many graphs with maximum vertex degree Δ , Cheeger constant at least δ and non-negative Ollivier curvature.*

We prove Theorem 3.4.3 in Appendix A.2. Furthermore, quoting directly from [118]:

“The high-level message is that on large sparse graphs, non-negative curvature (in an even weak sense) induces extremely poor spectral expansion. This stands in stark contrast with the traditional idea – quantified by a broad variety of functional inequalities over the past decade – that non-negative curvature is associated with good mixing behavior.”

In our view, it is highly desirable that the graphs used for GNNs have high Cheeger constants, in the sense of globally lacking bottlenecks. Having bounded vertex degree is certainly useful too, since it implies that the graphs will be sparse, and the nodes will not have to handle ever-increasing neighbourhoods for message passing as graphs grow larger in size.

However, by proving Theorem 3.4.3, we showed non-negative Ollivier curvature is *incompatible* with these properties for sufficiently large graphs. Specifically, given the *finite* supply of non-negatively curved sparse graphs, we can define N' as the largest number of nodes of such graphs. Then, for all graphs G where $|V(G)| > N'$, we will be *unable* to produce a computational graph for a GNN which is non-negatively curved everywhere. It remains an interesting challenge to provide an bound on N' (as a function of δ and Δ). It is possible that a careful analysis of [118] may provide this.

Further, while the expander graphs we generate are negatively-curved at -1 everywhere, and we will empirically show this helps alleviate oversquashing, we also believe that it is worthy of further investigation to theoretically examine whether performance of GNNs decreases significantly when the curvature is less than -1 .

The negative curvature of each edge in G_n implies that they are locally ‘tree-like’. In Appendix A.3, we make this statement precise by showing that G_n is ‘tree-like’ up to scale $c \log(n)$ about each node, for $c \simeq (1/2)(\log((1 + \sqrt{5})/2))^{-1}$ (see Figure 12 (Right) for a schematic view).

This tree-like structure might seem, at first, to be counter-productive for good propagation across the graphs G_n . Indeed, GNNs based on trees have been shown to have provably poor performance [4]. The reason for this seems to be two-fold. On the one hand, trees have small Cheeger constant. Indeed, any tree G on n vertices has a Cheeger constant $1/\lfloor n/2 \rfloor$, since we may find an edge that, when removed, decomposes the graph into subgraphs with $\lfloor n/2 \rfloor$ and $\lceil n/2 \rceil$ vertices. As discussed in Section 3.3 and in [72], when a graph has small Cheeger constant, its performance when used as a template for a GNN is likely to become poor. Secondly, GNNs based on trees are susceptible to oversquashing. For a k -regular infinite tree, there are $k(k-1)^{r-1}$ vertices at distance r from a given vertex. Hence, if information is to be propagated at least distance r from a given vertex, then seemingly an exponential amount of information is required to be stored.

However, neither of these issues are problematic for a GNN based on the Cayley graph G_n . By Theorem 3.3.9, their Cheeger constants are bounded away from 0. Secondly, although they are tree-like locally, this is only true up to scale $O(\log n)$. In fact, the r -neighbourhood

of any vertex is the whole graph G_n as soon as $r > C \log n$, for some constant C , by Theorem 3.3.5. Being tree-like up to distance $O(\log n)$ does not lead to a requirement to store too much information as the message propagates. This is because $k(k-1)^{r-1}$ is polynomial in n when $r \leq O(\log n)$. Beyond this scale, there exist many additional connections, which lead to many possible paths joining any pair of vertices. The perspective of information transfer also gives rise to another perspective in which expanders fare very favourably: the *mixing time* of their corresponding Markov chain (see Appendix A.4 for details).

3.5. Expander graph propagation

Let an input to a graph neural network be a node feature matrix $\mathbf{X} \in \mathbb{R}^{|V| \times d}$, and an adjacency matrix $\mathbf{A} \in \mathbb{R}^{|V| \times |V|}$. This setup is such that the feature vector of node u , $\mathbf{x}_u \in \mathbb{R}^d$, can be recovered by taking an appropriate row from \mathbf{X} . Note that the adjacency information can also be fed in an edge-list manner, which is desirable from a scalability perspective. Further, each edge in the graph may be endowed with additional features rather than a single real scalar. None of the above modifications would change the essence of our findings; we use a matrix formalism here purely for simplicity.

There exist many ways in which the computed Cayley graph $\text{Cay}(\text{SL}(2, \mathbb{Z}_n); S_n)$ can be leveraged for message propagation, and exploring these variations could be very useful for future work. Here, we opt for a simple construction: interleave running a standard GNN over the given input structure, followed by running another GNN layer over the relevant Cayley graph. If we let $\mathbf{A}^{\text{Cay}(n)}$ be an adjacency matrix derived from $\text{Cay}(\text{SL}(2, \mathbb{Z}_n); S_n)$, this implies:

$$\mathbf{H} = \text{GNN}(\text{GNN}(\mathbf{X}, \mathbf{A}), \mathbf{A}^{\text{Cay}(n)}) \quad (3.5.1)$$

Here, GNN refers to any preferred GNN layer, such as the graph isomorphism network [1, GIN]:

$$\mathbf{h}_u = \phi \left((1 + \epsilon) \mathbf{x}_u + \sum_{v \in \mathcal{N}_u} \mathbf{x}_v \right) \quad (3.5.2)$$

where \mathcal{N}_u is the neighbourhood of node u , i.e. in our setup, the set of all nodes v such that $a_{vu} \neq 0$. $\epsilon \in \mathbb{R}$ is a learnable scalar, and $\phi: \mathbb{R}^d \rightarrow \mathbb{R}^d$ is a two-layer MLP.

This procedure is iterated for a certain number of steps, after which the computed node embeddings in \mathbf{H} can be used for any downstream task of interest—such as node classification, link prediction or graph classification. Note that, unlike [4], who apply their custom layer only at the *tail* of the architecture, we apply the expander graph immediately after each layer over the input graph. We find that if the input graph given by \mathbf{A} contains bottlenecks, applying the GNN over $\mathbf{A}^{\text{Cay}(n)}$ only at the end may result in oversquashing occurring before any expander graph propagation can take place.

The setup so far assumed the number of nodes in our input graph to line up with the Cayley graph, that is, $\mathbf{A}^{\text{Cay}(n)} \in \mathbb{R}^{|V| \times |V|}$. However, there is no guarantee that we can find an appropriate n such that $\text{Cay}(\text{SL}(2, \mathbb{Z}_n); S_n)$ would have $|V|$ nodes. What we can do in practice, as an approximation, is choose the smallest n such that the number of nodes of $\text{Cay}(\text{SL}(2, \mathbb{Z}_n); S_n)$ is $\geq |V|$, then consider $\mathbf{A}_{1:|V|, 1:|V|}^{\text{Cay}(n)}$ —i.e. only the subgraph containing the first $|V|$ nodes in the Cayley graph.

There is a slight misalignment to our theory in this slicing choice—if the $|V|$ vertices in this subgraph are chosen completely arbitrarily, we risk disconnecting the graph. However, in all our experiments we construct the Cayley graph in a breadth-first manner, starting from the identity element as “node zero”. Hence, the node at index i is always guaranteed to be reachable from the nodes at lower indices ($j < i$), and the graph cannot be disconnected under this construction. More interesting strategies for this step can also be considered in the future. Note that, much like the fully connected graph used by [4], we interpret the Cayley graph mainly as a *template* for global information propagation, in order to relieve bottlenecks in a scalable way. Our interpretation, hence, assumes that the efficient diffusion of information over the whole graph is of benefit to the learning task we perform. When this is not the case, it might be worthwhile to construct expanders that somehow align with the input graph, but no such expander constructions are currently known, to the best of our knowledge. There is also a possible effect of *stochasticity* due to arbitrarily having to align the Cayley graph’s nodes to the input graph—which would not appear when using master nodes or fully-connected graphs—though our preliminary experiments did not observe any such negative effects.

Algorithm 1 summarises the steps of our proposed EGP model. As direct corollaries of results we proved or demonstrated, we note that EGP satisfies all four of our desirable criteria: **(C1)** by Theorem 3.3.5 (so long as logarithmically many layers are applied), **(C2)** by Theorem 3.3.4 (high Cheeger constant implies no bottlenecks), **(C3)** by the fact our Cayley graphs are 4-regular and hence sparse, and **(C4)** by the fact we can generate a Cayley graph of appropriate size without detailed analysis of the input—we may precompute a “bank” of Cayley graphs of various sizes to use in an ad-hoc manner.

3.6. Empirical evaluation

Our work provides mainly a theoretical contribution: demonstrating a simple, theoretically-grounded approach to relieving bottlenecks and oversquashing in GNNs without requiring quadratic complexity or dedicated preprocessing. Further, we prove several additional results which deepen our understanding of curvature-based analysis of GNNs, showing how our expanders can be favourable in spite of their negatively-curved edges. We now provide results that empirically supplement our claim.

Algorithm 1: Expander graph propagation (EGP) forward pass

Inputs : Node features $\mathbf{X} \in \mathbb{R}^{|V| \times d}$, Adjacency matrix $\mathbf{A} \in \mathbb{R}^{|V| \times |V|}$
Output: Node embeddings \mathbf{H}

// Choose the smallest Cayley graph from our family that has number of nodes equal to, or greater than, $|V|$
 $n \leftarrow \operatorname{argmin}_{m \in \mathbb{N}} |V(\operatorname{Cay}(\operatorname{SL}(2, \mathbb{Z}_m); S_m))| \geq |V|$; // We can use Equation 3.3.10 to determine n

$G^{\operatorname{Cay}(n)} \leftarrow \operatorname{Cay}(\operatorname{SL}(2, \mathbb{Z}_n); S_n)$

$\mathbf{A}_{uv}^{\operatorname{Cay}(n)} \leftarrow \begin{cases} 1 & (u, v) \in E(G^{\operatorname{Cay}(n)}) \\ 0 & \text{otherwise} \end{cases}$; // Populate adjacency matrix of the Cayley graph

$\mathbf{H}^{(0)} \leftarrow \mathbf{X}$; // Initialise GNN inputs

for $t \in \{1, \dots, T\}$ **do**

if $t \bmod 2 = 0$ **then**

$\mathbf{H}^{(t)} \leftarrow \operatorname{GNN}^{(t)}(\mathbf{H}^{(t-1)}, \mathbf{A})$; // GNN layer over input graph; e.g. Equation 3.5.2

else

$\mathbf{H}^{(t)} \leftarrow \operatorname{GNN}^{\operatorname{Cay}(t)}(\mathbf{H}^{(t-1)}, \mathbf{A}_{1:|V|, 1:|V|}^{\operatorname{Cay}(n)})$; // GNN layer over Cayley graph; e.g. Eq. 3.5.2

return $\mathbf{H}^{(T)}$; // Return final embeddings for downstream use

Tree-NeighborsMatch We start by comparing our models on the **Tree-NeighborsMatch** task (for more details, see Alon and Yahav [4, Section 4.1]). **Tree-NeighborsMatch** is a synthetic benchmark explicitly designed to test a GNN’s ability to counter oversquashing, and therefore it allows us to empirically verify that EGP is capable of alleviating oversquashing. We augment the original GIN implementation from the authors [4] with EGP layers, and find that it is capable of solving the task at **depth=5 at 100% accuracy**, demonstrating alleviated oversquashing. In comparison, baseline GIN can only achieve 29% on this same task, and the best-performing GNN without EGP—i.e. propagating over the input tree only—cannot exceed 60% accuracy.

OGB Datasets For real-world evaluation, we leverage the established Open Graph Benchmark collection of tasks [2, OGB]. Specifically, we provide results on all of its graph classification datasets: `ogbg-molhiv`, `ogbg-molpcba`, `ogbg-ppa` and `ogbg-code2`. The first two are among the largest molecule property prediction datasets in the MoleculeNet benchmark [119]. The third dataset is concerned with classifying species into their taxa, from their protein-protein association networks [120, 121] given as input. The fourth dataset is a

Table 3 – Comparative evaluation performance on the four datasets studied. Our baseline model is a GIN [1], using exactly the same implementation as in [2]. See Appendix A.5 for ablations.

Model	ogbg-molhiv	ogbg-molpcba	ogbg-ppa	ogbg-code2
GIN	0.7558 ± 0.0140	0.2266 ± 0.0028	0.6892 ± 0.0100	0.1495 ± 0.0023
GIN + EGP	0.7934 ± 0.0035	0.2329 ± 0.0019	0.7027 ± 0.0159	0.1497 ± 0.0015

code summarisation task: it requires predicting the tokens in the name of a Python method, given the abstract syntax tree (AST) of its implementation.

We provide a summary of important dataset statistics in Appendix A.5; please see [2] for detailed information. These datasets are designed to span a wide variety of domains (virtual drug screening, molecular activity prediction, protein-protein interactions, code summarisation) and sizes (from small molecules to very large syntax trees—the largest graph in *ogbg-code2* has 36,123 nodes).

Models In all four datasets, we want to *directly* evaluate the empirical gain of introducing an EGP layer and completely rule out any effects from parameter count, or similar architectural decisions.

To enable this, we take inspiration from the experimental setup of [4]. Our baseline model is the GIN [1], with hyperparameters as given by [2]. We use the *official* publicly available model implementation from the OGB authors [2], and modify all *even* layers of the architecture to operate over the appropriately-sampled Cayley graph.

Note that our construction leaves both the parameter count and latent dimension of the model *unchanged*, hence any benefits coming from optimising those have been diminished.

Results The results of our evaluation are presented in Table 3. It can be observed that, in all four cases, propagating information over the Cayley graph yields improvements in mean performance—these improvements are most apparent on *ogbg-molhiv*, but also present in *ogbg-molpcba* and *ogbg-ppa*. We believe that these results provide encouraging empirical evidence that propagating information over Cayley graphs is an elegant idea for alleviating bottlenecks. We provide additional results on OGB, comparing EGP to various other oversquashing-counteracting methods, in Appendix A.5.

3.7. Conclusion

In this paper, we have presented expander graph propagation (EGP), a novel and elegant approach to alleviating bottlenecks in graph representation learning, which provably supports global communication while not requiring quadratic complexity or dedicated preprocessing of the input.

To this end, we offered a detailed theoretical overview of Cayley graphs of special linear groups, $\text{Cay}(\text{SL}(2, \mathbb{Z}_n); S_n)$. We cite proofs that these graphs have highly favourable properties for information propagation in graph neural networks: they are sparse and 4-regular, they have logarithmic diameter, and they can be efficiently precomputed by a simple procedure that does not rely on the input structure. We show that, in spite of having negatively curved edges, our findings do not violate any prior results on understanding oversquashing via curvature. Even under a simple intervention—interleaving EGP layers inbetween standard GNN layers—we have been able to recover significant performance returns without changing the parameter count or latent space dimensionality.

We hope that our work serves as a foundation for further work on deploying Cayley graphs—or other expander families—within the context of GNNs.

Chapter 4

Graph Neural Networks for Heterophilic data: Prologue to the second article

4.1. Article Details

Evolving Computation Graphs (ECG). Andreea Deac, Jian Tang. The paper is under submission at NeurIPS 2023 and was accepted at ICML 2023 TAGML workshop.

Personal contribution: I proposed the idea, implemented and ran the experiments on this paper based on Jian’s supervision of the project. All authors contributed to writing the manuscript.

4.2. Context

Most graph neural networks are built on the assumption that the input graphs are homophilic, which means that nodes are connected by an edge if they belong to the same class. In fact, when graphs are highly homophilic, substantial progress can be made without most of the complexities induced by GNNs—if we assume that most of a node’s neighbours share a label with it, then taking simple averages of neighbours can be a sufficiently strong regulariser—a feat demonstrated by the simplified graph convolution [79, SGC]. Consequentially, GNNs were noticed to perform significantly worse on heterophilic data [122].

As many graphs of interest in the world are heterophilic (such as social networks, where, e.g., re-tweeting does not necessarily mean agreement, fraud networks, e.g., fraudsters connecting with victims, or protein-protein networks, e.g., proteins with different functions being associated in the same pathways), more works focused on designing specialised GNNs that perform well on heterophilous datasets. However, Platonov et al. [3] showed that the previous benchmark used to test heterophilic architectures suffered from limitations such as node duplicates, class imbalance and overlap in the data sources. Therefore, they proposed a benchmark that addresses these limitations, on which it was noticed that the specialised GNNs

are outperformed by classic, general-purpose GNN architectures. Therefore, we wanted to find a solution that improves GNNs’ performance without specialising the architecture, such as modifying the computation graphs towards increased homophily. This was also based on the previous success we obtained with Expander Graph Propagation [22], presented in the previous chapter, where we modify the computation graphs using templates that are known to improve long-range connectivity.

However, in the case of ECG, such template graphs are not immediately available. In order to know how to improve homophily on *test nodes*, it is important to have a strong guess of the test node’s label—but, as classifying the test nodes well is arguably the main objective of a learned GNN model, this produces a *chicken-and-egg problem*: the data necessary for obtaining a perfectly-homophilic graph at test time requires first solving the downstream task we care about! Therefore, in ECG, we propose a customised solution, where we first use *weak classifiers* to discover nodes that are similar by predicted label or local structure, and then deriving new computation graphs by leveraging embedding similarity in the weak classifier’s latent space.

4.3. Modulator

In ECG [23], we demonstrate the utility of improving homophily in the computational graph, via learned *weak classifiers*—both point-wise and structural. This procedure is conceptually elegant, and easy to motivate, but nontrivial to stabilise, as it relies on (at least) a two-phase method, and the success of the second phase critically depends on the power of the classifiers trained in the first phase. In this case, the modulator consists of a pre-trained weak classifier, from which edges are sampled using the similarity of node embeddings obtained from this classifier. As a common example, the ECG modulator may sample edges according to the k nearest neighbours of each node in the latent space of the weak classifier.

4.4. Contributions and Research Impact

“Evolving Computation Graphs” (ECG) [23] proposed a novel method to improve GNNs’ performance on heterophilic graphs, yielding improvements in 95% of the cases tested. As this work is recent, there have not been any follow-up papers building on it yet, but one of the things I am most excited about would be extending the idea of evolved computation graphs beyond ameliorating heterophily. In general, we may consider a generic *reward model*, providing a way to score the “desirability” of a particular graph (in the particular case of ECG, such a reward model would compute the graph’s homophily ratio), and then trying to optimise the model’s reward as guidance for edge selection.

Chapter 5

Evolving Computation Graphs

Graph neural networks (GNNs) have demonstrated success in modeling relational data, especially for data that exhibits homophily: when a connection between nodes tends to imply that they belong to the same class. However, while this assumption is true in many relevant situations, there are important real-world scenarios that violate this assumption, and this has spurred research into improving GNNs for these cases. In this work, we propose Evolving Computation Graphs (ECGs), a novel method for enhancing GNNs on heterophilic datasets. Our approach builds on prior theoretical insights linking node degree, high homophily, and inter vs intra-class embedding similarity by rewiring the GNNs’ computation graph towards adding edges that connect nodes that are likely to be in the same class. We utilise weaker classifiers to identify these edges, ultimately improving GNN performance on non-homophilic data as a result. We evaluate ECGs on a diverse set of recently-proposed heterophilous datasets and demonstrate improvements over the relevant baselines. ECG presents a simple, intuitive and elegant approach for improving GNN performance on heterophilic datasets without requiring prior domain knowledge.

5.1. Introduction

Neural networks applied to graph-structured data have demonstrated success across various domains, including practical applications like drug discovery [55], transportation networks [56], chip design [57] and theoretical advancements [58, 59]. Numerous architectures fall under the category of graph neural networks [18], with one of the most versatile ones being Message Passing Neural Networks [13]. The fundamental concept behind these networks is that nodes communicate with their neighbouring nodes through messages in each layer. These messages, received from neighbours, are then aggregated in a permutation-invariant manner to contribute to a new node representation.

It has been observed that the performance of graph neural networks may rely on the underlying assumption of *homophily*, which suggests that nodes are connected by edges if

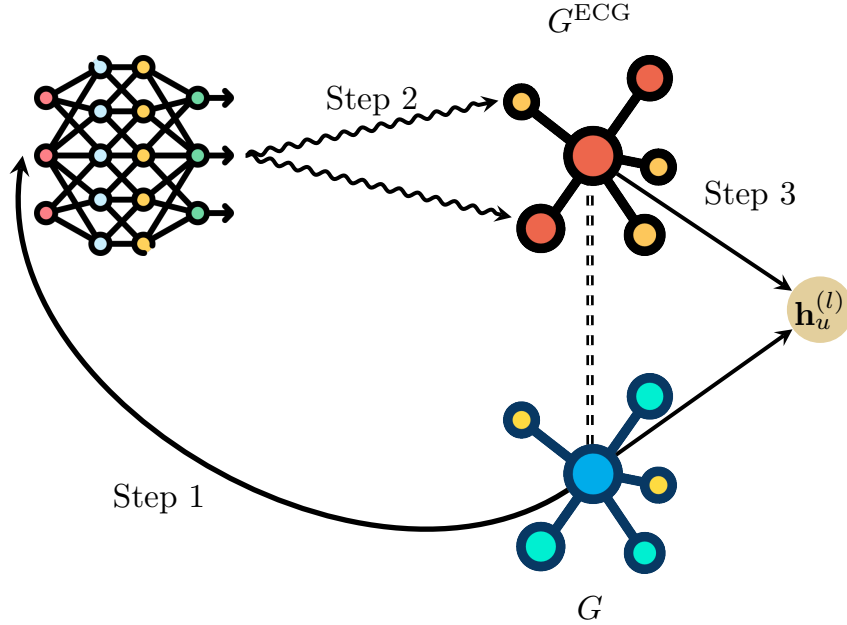


Figure 13 – A simplified illustration of Evolving Computation Graphs. **Step 1:** nodes in a graph, G , are embedded using a pre-trained weak classifier. **Step 2:** Based on these embeddings, a nearest-neighbour graph, G^{ECG} , is generated. This graph is likely to have improved propagation and homophily properties (illustrated by similar colours between neighbouring nodes). **Step 3:** Message passing is performed, both in the original and in the ECG graph, to update node representations.

they are similar based on their attributes or belonging to the same class, as commonly seen in social or citation networks. However, this assumption often fails to accurately describe real-world data when the graph contains *heterophilic* edges, connecting dissimilar nodes. This observation holds particular significance since graph neural networks tend to exhibit significantly poorer performance on heterophilic graphs compared to datasets known to be homophilic. Several studies [35, 123–125] have highlighted this issue, using a mixture of strongly homophilous graphs—such as Cora, Citeseer and Pubmed [126]—as well a standard suite of six heterophilic datasets—Squirrel, Chameleon, Cornell, Texas, Wisconsin and Actor [127, 128]—first curated jointly by Pei et al. [129].

In the context of this standard suite of heterophilic graphs, it has been observed that general graph neural network architectures tend to underperform unless there is high label informativeness [122, 130]. In prior work, this issue was tackled primarily by proposing modifications to the GNN architecture. These modifications include changes to the aggregation function, such as separating self- and neighbour embeddings [35], mixing low- and high-frequency signals [131, 132], and predicting and utilising the compatibility matrix [133]. Other approaches involve using the Jacobi basis in spectral GNNs [124] or learning cellular sheaves for neural sheaf diffusion [134] to improve performance.

However, it was recently remarked [3] that this standard heterophilous suite has significant drawbacks, such as data originating from only three sources, two of the datasets having significant numbers of repeated nodes and improper evaluation regarding class imbalance. To address these shortcomings, a more recent benchmark suite has been introduced by Platonov et al. [3], incorporating improvements on all of the above issues. Interestingly, once such corrections are accounted for, standard GNN models such as graph convolutional networks [14, GCN], GraphSAGE [135, SAGE], graph attention networks [15, GAT], and Graph Transformers [136, GT] have demonstrated superior performance compared to specialized architectures tailored specifically for heterophily—in spite of the heterophilic properties of the datasets. The notable exception is *-sep* [35] which has consistently improved GAT and GT by modelling self and neighbouring nodes separately.

In light of this surprising discovery, we suggest that there should be alternate routes to making the most of heterophilic datasets. Rather than attempting to modify these standard GNNs, we propose modifying their *computation graph*: effectively, enforcing messages to be sent across additional pairs of nodes. These node pairs are chosen according to a particular measure of *similarity*. If the similarity metric is favourably chosen, such a computation graph will improve the overall homophily statistics, thereby creating more favourable conditions for GNNs to perform well.

We further propose that the *modification* of the computation graph should be separate from its *utilisation*¹. That is, we proceed in two phases: the first phase learns the representations that allow us to construct new computation graphs, and the second phase utilises those representations to construct new computation graphs, to be utilised by a GNN in each layer. This design choice makes our method elegant, performant and easy to evaluate: the two-phase nature means we are not susceptible to bilevel optimisation (as in [138, 139]), the graphs we use need to be precomputed exactly once rather than updated on-the-fly in every layer (as in [140]), and because the same computation graph is used across all GNN layers, we can more rigidly evaluate how useful this graph is, all other things kept equal.

Hence, the essence of our method is *Evolving Computation Graphs (ECG)*, which uses weak classifiers to generate node embeddings. These embeddings are then used to define a similarity metric between nodes (such as cosine similarity). We then select edges in a *k*-nearest neighbour fashion: we connect each node to *k* nodes most similar to it, according to the metric. The edges selected in this manner form a complementary graph, which we propose using in parallel with the input graph to update each node’s representation. For this purpose, we use standard, off-the-shelf, GNNs. Our method is illustrated in Figure 13.

1. We note that [129, 137] also rest on a similar proposal, however their updated computation graph is fully derived as a function of the input graph structure (ignoring node features and labels), and thus it is unavoidably *vulnerable* to any biases or inconsistencies in the input graph—and real-world graph inputs are rarely flawless.

The nature of the weak classifier employed in ECG is flexible and, for the purpose of this paper, we used two representative options. The first option is a point-wise MLP classifier, attempting to cluster together nodes based on the given training labels, without any graph-based biases. For the second option, we attempt the converse: utilising the given graph structure and node features, but not relying on the training labels. This is a suitable setting for a self-supervised graph representation learning method, such as BGRL [93], which is designed to cluster together nodes with similar local neighbourhoods—both in terms of subgraphs and features—through a bootstrapping objective [141].

To evaluate the effectiveness of ECG, we conduct experiments on the benchmark suite proposed by Platonov et al. [3]. Our results demonstrate that ECG models outperform their GNN baselines in 19 out of 20 head-to-head comparisons. The most significant improvements can be noticed for GCNs—which are best suited to benefit from improved homophily—where improvements reach up to 10% in absolute terms. Further, the best performing ECG models outperform a diverse set of representative heterophily-specialised GNNs.

5.2. Background

In this section, we introduce the generic setup of learning representations on graphs, along with all of the key components that comprise our ECG method.

Graph representation learning setup. We denote graphs by $G = (V, E)$, where V is the set of nodes and E is the set of edges, and we denote by $e_{uv} \in E$ the edge that connects nodes u and v . For the datasets considered here, we can assume that the input graphs are provided to the GNNs via two inputs: the *node feature matrix*, $\mathbf{X} \in \mathbb{R}^{|V| \times k}$ (such that $\mathbf{x}_u \in \mathbb{R}^k$ are the input features of node $u \in V$), and the *adjacency matrix*, $\mathbf{A} \in \{0, 1\}^{|V| \times |V|}$, such that a_{uv} indicates whether nodes u and v are connected by an edge. We further assume the graph is *undirected*; that is, $\mathbf{A} = \mathbf{A}^\top$. We also use $d_u = \sum_{v \in V} a_{uv}$ ($= \sum_{v \in V} a_{vu}$) to denote the degree of node u .

We focus on node classification tasks with C representing the set of possible classes, where for node with input features \mathbf{x}_u , there is a label $y_u \in C$. Thus we aim to learn a function f that minimises $\mathbb{E}[\mathcal{L}(y_u, \hat{y}_u)]$, where \hat{y}_u is the prediction of $f(\mathbf{x}_u) = \hat{y}_u$, and \mathcal{L} is the cross-entropy loss function.

Graph neural networks. The one-step layer of a GNN can be summarised as follows [18]:

$$\mathbf{h}_u^{(l)} = \phi^{(l)} \left(\mathbf{h}_u^{(l-1)}, \bigoplus_{(u,v) \in E} \psi^{(l)}(\mathbf{h}_u^{(l-1)}, \mathbf{h}_v^{(l-1)}) \right) \quad (5.2.1)$$

where, by definition, we set $\mathbf{h}_u^{(0)} = \mathbf{x}_u$. Leveraging different (potentially learnable) functions for $\phi^{(l)} : \mathbb{R}^k \times \mathbb{R}^m \rightarrow \mathbb{R}^k$, $\bigoplus : \text{bag}(\mathbb{R}^m) \rightarrow \mathbb{R}^m$ and $\psi^{(l)} : \mathbb{R}^k \times \mathbb{R}^k \rightarrow \mathbb{R}^m$ then recovers well-known GNN architectures. Examples include GCN [14]: $\psi^{(l)}(\mathbf{x}_u, \mathbf{x}_v) = \beta_{uv} \omega^{(l)}(\mathbf{x}_v)$,

with $\beta_{uv} \in \mathbb{R}$ being a constant based on \mathbf{A} , GAT [15]: $\psi^{(l)}(\mathbf{x}_u, \mathbf{x}_v) = \alpha^{(l)}(\mathbf{x}_u, \mathbf{x}_v)\omega^{(l)}(\mathbf{x}_v)$ with $\alpha^{(l)} : \mathbb{R}^k \times \mathbb{R}^k \rightarrow \mathbb{R}$ being a (softmax-normalised) self-attention mechanism, and *-sep* [35]: $\phi^{(l)} = \mathbf{W}_{\text{self}}^{(l)}\phi_1^{(l)}(\mathbf{h}_u^{(l-1)}) + \mathbf{W}_{\text{agg}}^{(l)}\phi_2^{(l)}\left(\bigoplus_{(u,v) \in E} \psi^{(l)}(\mathbf{h}_u^{(l-1)}, \mathbf{h}_v^{(l-1)})\right)$, where we explicitly decompose $\phi^{(l)}$ into two parts, with one of them ($\phi_1^{(l)}$) depending on the receiver node only. Homophily. has been repeatedly mentioned as an important measure of the graph, especially when it comes to GNN performance. Intuitively, it corresponds to an assumption that neighbouring nodes tend to share labels: $a_{uv} = 1 \implies y_u = y_v$, which is often the case for many industrially-relevant real world graphs (such as social networks). Intuitively, a graph with high homophily will make it easier to exploit neighbourhood structure to derive more accurate node labels.

However, in spite of the importance of quantifying homophily in a graph, there is no universally-agreed-upon metric for this. One very popular metric, used by several studies, is *edge homophily* [142], which measures the proportion of homophilic edges:

$$\text{h-edge} = \frac{|(u,v) \in E : y_u = y_v|}{|E|} \quad (5.2.2)$$

while [130] also introduces *adjusted homophily* to account for number of classes and their distributions:

$$\text{h-adj} = \frac{\text{h-edge} - \sum_{k=1}^C D_k^2 / (2|E|)^2}{1 - \sum_{k=1}^C D_k^2 / (2|E|)^2} \quad (5.2.3)$$

where $D_k = \sum_{u: y_u=k} d_u$, the sum of degrees for the nodes belonging to class k .

Additionally, the *label informativeness* (LI) measure proposed in [130] measures how much information about a node’s label is gained by observing its neighbour’s label, on average. It is defined as

$$\text{LI} = I(y_\xi, y_\eta) / H(y_\xi) \quad (5.2.4)$$

where $(\xi, \eta) \in E$ is a uniformly-sampled edge, H is the Shannon entropy and I is mutual information.

Weak classifier. In order to derive novel computation graphs which are likely to result in higher test performance, we likely require “novel” homophilic connections to emerge—rather than amplifying the homophily already present in \mathbf{A} . Therefore, for the purposes of building a useful computation graph, our ECG method aims to first learn representations of nodes governed by a model which does *not* have access to inputs (\mathbf{X}), graph structure (\mathbf{A}) and training labels (\mathbf{y}_{tr}) simultaneously. We hence call such a model a “weak classifier”, as it is not exposed to the same kind of inductive biases as a supervised GNN would (and hence it must obtain useful models which do not rely on these biases).

MLPs. Arguably the simplest way to make a weak classifier, as above, is to withhold access to the graph structure (\mathbf{A}), and force the model to classify the nodes in pure isolation from one another. This is effectively a standard multi-layer perceptron (MLP) applied

pointwise. Another way of understanding this model is setting $\mathbf{A} = \mathbf{I}_{|V|}$, or equivalently, $E = \{(u, u) \mid u \in V\}$, in Equation 5.2.1, which is sometimes referred to as the Deep Sets model [37]. We train this model by using cross-entropy against the training nodes’ labels (\mathbf{y}_{tr}) and, once trained, use the final layer activations, $\mathbf{h}_u^{(L)}$ —for a model with L layers—as our MLP embeddings.

BGRL. While using the embeddings from an MLP can offer a solid way to improve homophily metrics, their confidence will degrade for nodes where the model is less accurate outside of the training set—which are arguably the nodes we would like to improve predictions on the most. Accordingly, as a converse approach to obtaining a weak classifier, we may also withhold access to the training labels (\mathbf{y}_{tr}). Now the model is forced to arrange the node representations in a way that will be mindful of the input features and graph structure, but without knowing the task specifics upfront, and hence not vulnerable to overfitting on the training nodes. Such a weak classifier naturally lends itself to self-supervised learning on graphs.

Bootstrapped graph latents [93, BGRL] is a state-of-the-art self-supervised graph representation learning method based on BYOL [141]. BGRL learns two GNN encoders with identical architecture; an *online* encoder, \mathcal{E}_θ , and a *target* encoder, \mathcal{E}_ϕ . BGRL also contains a *predictor* network p_θ . We offer a “bird’s eye” view of how BGRL is trained, and defer to [93] for implementation details.

At each step of training, BGRL proceeds as follows. First, two data augmentations (e.g. random node and edge dropout) are applied to the input graph, obtaining augmented graphs $(\mathbf{X}_1, \mathbf{A}_1)$ and $(\mathbf{X}_2, \mathbf{A}_2)$. Then, the two encoders are applied to these augmentations, recovering a pair of latent node embeddings: $\mathbf{H}_1 = \mathcal{E}_\theta(\mathbf{X}_1, \mathbf{A}_1)$, $\mathbf{H}_2 = \mathcal{E}_\phi(\mathbf{X}_2, \mathbf{A}_2)$. The first embedding is additionally passed through the predictor network: $\mathbf{Z}_1 = p_\theta(\mathbf{H}_1)$. At this point, BGRL attempts to preserve the cosine similarity between all the corresponding nodes in \mathbf{Z}_1 and \mathbf{H}_2 , via the following loss function:

$$\mathcal{L}_{\text{BGRL}} = -\frac{\mathbf{Z}_1 \mathbf{H}_2^\top}{\|\mathbf{Z}_1\| \|\mathbf{H}_2\|} \quad (5.2.5)$$

Lastly, the parameters of the online encoder \mathcal{E}_θ and predictor p_θ are updated via stochastic gradient descent on $\mathcal{L}_{\text{BGRL}}$, and the parameters of the target encoder \mathcal{E}_ϕ are updated as the exponential moving average of the online encoder’s parameters.

Once the training procedure concludes, typically only the online network \mathcal{E}_θ is retained, and hence the embeddings $\mathbf{H} = \mathcal{E}_\theta(\mathbf{X}, \mathbf{A})$ are the BGRL embeddings of the input graph given by node features \mathbf{X} and adjacency matrix \mathbf{A} .

Owing to its bootstrapped objective, BGRL does not require the generation of negative samples, and is hence computationally efficient compared to contrastive learning approaches. Further, it is very successful at large scales; it was shown by Addanki et al. [67] that BGRL’s

benefits persist on industrially relevant graphs of hundreds of millions of nodes, leading to one of the top-3 winning entries at the OGB-LSC competition [60]. This is why we employ it as a representative self-supervised embedding method for our ECG framework.

5.3. Evolving Computation Graphs

Armed with the concepts above, we are now ready to describe the steps of the ECG methodology. Please refer to Algorithm 2 for a pseudocode summary.

Step 1: Embedding extraction. Firstly, we assume that an appropriate weak classifier has already been trained (as discussed in previous sections), and is capable of producing node embeddings. We start by invoking this classifier to obtain ECG embeddings $\mathbf{H}_{\text{ECG}} = \gamma(\mathbf{X}, \mathbf{A})$. We study two simple but potent variants of γ , as per the previous section:

MLP: In this case, we utilise a simple deep MLP²; that is, $\gamma(\mathbf{X}, \mathbf{A}) = \sigma(\sigma(\mathbf{X}\mathbf{W}_1)\mathbf{W}_2)$, where \mathbf{W}_i are the weights of the MLP, and σ is the GELU activation function [143].

BGRL: In this case, we set $\gamma = \mathcal{E}_\theta$, the online encoder of BGRL. For our experiments, we utilise a publicly available off-the-shelf implementation of BGRL provided by the Deep Graph Library³ [144], which uses a two-layer GCN [14] as the base encoder.

The parameters of γ are kept frozen throughout, and are not to be further trained on.

Step 2: Graph construction. Having obtained \mathbf{H}_{ECG} , we can now use it to compute a similarity metric between the nodes, such as cosine similarity, as follows:

$$\mathbf{S} = \mathbf{H}_{\text{ECG}}\mathbf{H}_{\text{ECG}}^\top \quad \hat{s}_{uv} = \frac{s_{uv}}{\|\mathbf{h}_{\text{ECG}_u}\| \|\mathbf{h}_{\text{ECG}_v}\|} \quad (5.3.1)$$

Based on this similarity metric, for each node $u \in V$ we select its neighbourhood $\mathcal{N}_u^{\text{ECG}}$ to be its k nearest neighbours in \mathbf{S} (where k is a tunable hyperparameter):

$$\mathcal{N}_u^{\text{ECG}} = \text{top-}k_{v \in V} \hat{s}_{uv} \quad (5.3.2)$$

Equivalently, we construct a new computation graph, $G^{\text{ECG}} = (V, E^{\text{ECG}})$, such that its edges are $E^{\text{ECG}} = \{(u, v) \mid u \in V \wedge v \in \mathcal{N}_u\}$. These edges are effectively determined by the weak classifier.

Step 3: Parallel message passing. Finally, once the ECG graph, G^{ECG} , is available, we can run our GNN of choice over it. To retain the topological benefits contained in the input graph structure, we opt to run two GNN layers in parallel—one over the input graph (as in

2. Note that this MLP only computes high-dimensional embeddings of each node; while training γ , an additional logistic regression layer is attached to this architecture.

3. <https://github.com/dmlc/dgl/tree/master/examples/pytorch/bgml>

Equation 5.2.1), and one over the ECG graph, as follows:

$$\mathbf{h}_{\text{INP}_u}^{(l)} = \phi_{\text{INP}}^{(l)} \left(\mathbf{h}_u^{(l-1)}, \bigoplus_{(u,v) \in E} \psi_{\text{INP}}^{(l)}(\mathbf{h}_u^{(l-1)}, \mathbf{h}_v^{(l-1)}) \right) \quad (5.3.3)$$

$$\mathbf{h}_{\text{ECG}_u}^{(l)} = \phi_{\text{ECG}}^{(l)} \left(\mathbf{h}_u^{(l-1)}, \bigoplus_{(u,v) \in E^{\text{ECG}}} \psi_{\text{ECG}}^{(l)}(\mathbf{h}_u^{(l-1)}, \mathbf{h}_v^{(l-1)}) \right) \quad (5.3.4)$$

Then the representation after l layers is obtained by jointly transforming these two representations:

$$\mathbf{h}_u^{(l)} = \mathbf{W}^{(l)} \mathbf{h}_{\text{INP}_u}^{(l)} + \mathbf{U}^{(l)} \mathbf{h}_{\text{ECG}_u}^{(l)} \quad (5.3.5)$$

where $\mathbf{W}^{(l)}$ and $\mathbf{U}^{(l)}$ are learnable parameters.

Equations 5.3.3–5.3.5 can then be repeatedly iterated, much like is the case for any standard GNN layer. As it is possible that G^{ECG} will contain noisy edges which do not contribute to useful propagation of messages, we additionally apply DropEdge [145] when propagating over the ECG graph, with probability $p_{de} = 0.5$.

5.4. Experiments

We evaluate the performance of ECG on five heterophilic datasets, recently-proposed by Platonov et al. [3]: `roman-empire`, `amazon-ratings`, `minesweeper`, `tolokers` and `questions`. All five datasets are node classification tasks, testing for varying levels of homophily in the input (`roman-empire` has the highest label informativeness), different connectivity profiles (`tolokers` is the most dense, `questions` has the lowest values of clustering coefficients) and providing both real-world datasets (`amazon-ratings`), as well as synthetic examples (`minesweeper`).

We ran ECG as an extension on standard GNN models, choosing the “-sep” variant [35] for GAT and GT as it was noted to improve their performance consistently on these tasks [3]. Thus, our baselines are GCN, GraphSAGE, GAT-sep and GT-sep, which we extend by modifying their computation graph as presented in Section 5.3. For each ECG model, we ran three variants, depending on which weak classifier was used to select the complementary edges, E^{ECG} : the MLP, the BGRL, or a concatenation of the output of the two.

For each of these architectures, the hyper-parameters to sweep are the number of neighbours sampled in the ECG graph, k (selected from $\{3, 10, 20\}$), the edge dropout rate used on it (selected from $\{0., 0.5\}$), the hidden dimension of the graph neural networks, where the one ran on the original graph G always matches the one ran on G^{ECG} (selected from $\{256, 512\}$), as well as the standard choice of number of layers (selected from $\{2, 3, 4, 5\}$). In the Appendix, we present additional information on the experiments, together with the hyper-parameters corresponding to the best validation results.

Algorithm 2: Evolving Computation Graph for Graph Neural Networks: ECG-GNN

Input: Graph $G = (V, E)$; Node Feature Matrix \mathbf{X} ; Adjacency Matrix \mathbf{A} .
Hyper-parameters: Value of k ; Drop edge probability p_{de} ; Number of layers L ;
Output: Predicted labels $\hat{\mathbf{y}}$

```

begin
  /* Step 1: Extract embeddings */
   $\mathbf{H}_{\text{ECG}} \leftarrow \gamma(\mathbf{X}, \mathbf{A})$  /* Embeddings stored in matrix */
  /* Step 2: Construct ECG graph */
   $\mathbf{S} \leftarrow \mathbf{H}_{\text{ECG}} \mathbf{H}_{\text{ECG}}^\top$ 
  for  $u \in V$  do
    for  $v \in V$  do
       $\hat{s}_{uv} \leftarrow s_{uv} / (\|\mathbf{h}_{\text{ECG}_u}\| \|\mathbf{h}_{\text{ECG}_v}\|)$  /* Compute pair-wise cosine similarities */
     $\mathcal{N}_u^{\text{ECG}} \leftarrow \text{top-}k_{v \in V \hat{s}_{uv}}$  /* Compute  $k$  nearest neighbours of  $u$  */
   $E^{\text{ECG}} \leftarrow \{(u, v) \mid u \in V \wedge v \in \mathcal{N}_u\}$  /* Construct the ECG edges */
  /* Step 3: Running ECG-GNN with parallel processing of  $G$  and  $G^{\text{ECG}}$  */
  for  $u \in V$  do
     $\mathbf{h}_u^0 \leftarrow \mathbf{x}_u$  /* Setting initial node features */
  for  $l \leftarrow 1$  to  $L$  do
    /* Message passing propagation with the two parallel processors on  $G$  and  $G^{\text{ECG}}$  respectively */
     $E_{(l)}^{\text{ECG}} \leftarrow \text{DropEdge}(E^{\text{ECG}}, p_{de})$  /* Randomly drop edges in the ECG graph */
    for  $u \in V$  do
       $\mathbf{h}_{\text{INP}_u}^{(l)} \leftarrow \phi_{\text{INP}}^{(l)} \left( \mathbf{h}_u^{(l-1)}, \bigoplus_{(u,v) \in E} \psi_{\text{INP}}^{(l)} \left( \mathbf{h}_u^{(l-1)}, \mathbf{h}_v^{(l-1)} \right) \right)$  /* GNN on  $G$  */
       $\mathbf{h}_{\text{ECG}_u}^{(l)} \leftarrow \phi_{\text{ECG}}^{(l)} \left( \mathbf{h}_u^{(l-1)}, \bigoplus_{(u,v) \in E_{(l)}^{\text{ECG}}} \psi_{\text{ECG}}^{(l)} \left( \mathbf{h}_u^{(l-1)}, \mathbf{h}_v^{(l-1)} \right) \right)$  /* GNN on  $G^{\text{ECG}}$  */
       $\mathbf{h}_u^{(l)} \leftarrow \mathbf{W}^{(l)} \mathbf{h}_{\text{INP}_u}^{(l)} + \mathbf{U}^{(l)} \mathbf{h}_{\text{ECG}_u}^{(l)}$  /* Updating the node representation */
    /* Predict node labels */
    for  $u \in V$  do
       $p_u \leftarrow \text{softmax}(\mathbf{W}^{(c)} \mathbf{h}_u^{(L)})$   $\hat{y}_u \leftarrow \arg \max_{c \in C} p_c$  /* Predicted class label */
  return  $\hat{\mathbf{y}}$ 

```

In Table 4, we show the test performance corresponding to the highest validation score among all embedding possibilities for each of the five datasets and for each of the four baselines. Altogether, there are 20 dataset-model combinations that ECG is tested on. We find that on 19 out of these 20 combinations (marked with arrow up in the table), using ECG improves the performance of the corresponding GNN architecture, the only exception being GraphSAGE on `amazon-ratings`.

Moreover, we observe the highest gains in performance are achieved by ECG-GCN, ranging from 1.17% to 10.84% (absolute values) in a manner that is correlated with the homophily of the dataset. This confirms the hypothesis that, due to the aggregation function it employs, GCN is also the architecture most prone to performance changes based on the homophily of the graph.

Table 4 – ECG performance on datasets proposed in [3]. We report accuracy for roman-empire and amazon-ratings and ROC AUC for minesweeper, tolokera, and questions.

Model	roman-empire	amazon-ratings	minesweeper	tolokers	questions
MLP	65.88 \pm 0.38	45.90 \pm 0.52	50.89 \pm 1.39	72.95 \pm 1.06	70.34 \pm 0.76
GCN [14]	73.69 \pm 0.74	48.70 \pm 0.63	89.75 \pm 0.52	83.64 \pm 0.67	76.09 \pm 1.27
ECG-GCN	84.53 \pm 0.26 (\uparrow)	51.12 \pm 0.38 (\uparrow)	92.63 \pm 0.10 (\uparrow)	84.81 \pm 0.25 (\uparrow)	77.50 \pm 0.35 (\uparrow)
SAGE [135]	85.74 \pm 0.67	53.63 \pm 0.39	93.51 \pm 0.57	82.43 \pm 0.44	76.44 \pm 0.62
ECG-SAGE	87.88 \pm 0.25 (\uparrow)	53.45 \pm 0.27 (\downarrow)	94.11 \pm 0.07 (\uparrow)	82.61 \pm 0.29 (\uparrow)	77.23 \pm 0.36 (\uparrow)
GAT-sep [15]	88.75 \pm 0.41	52.70 \pm 0.62	93.91 \pm 0.35	83.78 \pm 0.43	76.79 \pm 0.71
ECG-GAT-sep	89.62 \pm 0.18 (\uparrow)	53.65 \pm 0.39 (\uparrow)	94.52 \pm 0.20 (\uparrow)	84.23 \pm 0.25 (\uparrow)	77.38 \pm 0.18 (\uparrow)
GT-sep [136]	87.32 \pm 0.39	52.18 \pm 0.80	92.29 \pm 0.47	82.52 \pm 0.92	78.05 \pm 0.93
ECG-GT-sep	89.56 \pm 0.16 (\uparrow)	53.25 \pm 0.39 (\uparrow)	93.62 \pm 0.27 (\uparrow)	84.00 \pm 0.24 (\uparrow)	78.12 \pm 0.32 (\uparrow)
H ₂ GCN	60.11 \pm 0.52	36.47 \pm 0.23	89.71 \pm 0.31	73.35 \pm 1.01	63.59 \pm 1.46
CPGNN	63.96 \pm 0.62	39.79 \pm 0.77	52.03 \pm 5.46	73.36 \pm 1.01	65.96 \pm 1.95
GPR-GNN	64.85 \pm 0.27	44.88 \pm 0.34	86.24 \pm 0.61	72.94 \pm 0.97	55.48 \pm 0.91
FSGNN	79.92 \pm 0.56	52.74 \pm 0.83	90.08 \pm 0.70	82.76 \pm 0.61	78.86 \pm 0.92
GloGNN	59.63 \pm 0.69	36.89 \pm 0.14	51.08 \pm 1.23	73.39 \pm 1.17	65.74 \pm 1.19
FAGCN	65.22 \pm 0.56	44.12 \pm 0.30	88.17 \pm 0.73	77.75 \pm 1.05	77.24 \pm 1.26
GBK-GNN	74.57 \pm 0.47	45.98 \pm 0.71	90.85 \pm 0.58	81.01 \pm 0.67	74.47 \pm 0.86
JacobiConv	71.14 \pm 0.42	43.55 \pm 0.48	89.66 \pm 0.40	68.66 \pm 0.65	73.88 \pm 1.16

Additionally, we note that BGRL-based embeddings are consistently preferred for the roman-empire and questions graphs. These datasets have the lowest average degree and average local clustering – this emphasises the need for different methods of obtaining complementary graphs, balancing connectivity and homophily aspects, as remarked in [146].

5.4.1. Qualitative studies

In Table 5, we analyse the properties of the complementary graphs G^{ECG} with $k = 3$ nearest neighbours. We note that this represents the graph used by ECG-GCN, which preferred lower values of k , while the optimal values of k for SAGE, GAT-sep and GT-sep were on the higher end, varying between 3, 10 and 20 depending on the dataset.

We observe that MLP-ECG confirms our hypothesis: taking the edges corresponding to the pairs of nodes marked as most similar by the ResNet results in a graph G^{ECG} with high homophily, especially compared to the original input graph. It is important to note that all of our MLP-ECG graphs were obtained with a relatively shallow ResNet, which, as it can be seen in Table 4 lacks in performance compared to the graph-based methods. However, our method’s success in conjunction with GNNs shows that even a weak classifier can be used

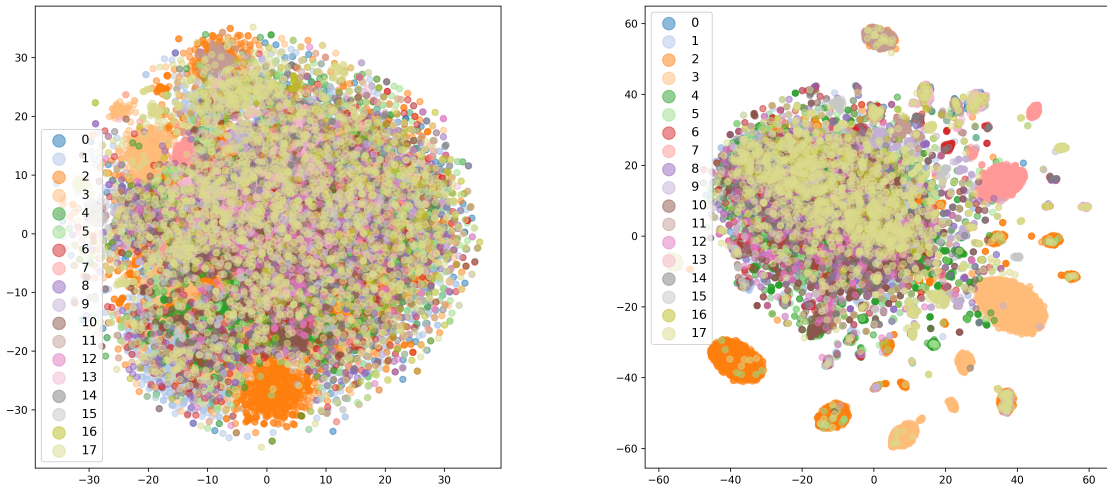


Figure 14 – For `roman-empire`, we use a random GCN layer to obtain node embeddings based on the original graph G (left) or from the complementary graph G^{ECG} (right). The colours correspond to the ground-truth labels of the nodes.

to generate homophilic graphs that can improve performance when used to complement the information provided by the given input data.

In Figure 14, we also verify how predictive of the node classes the graph topology is when obtained from the original data compared to when we build a complementary graph G^{ECG} . More precisely, we first build the graph G^{ECG} as presented in Step 1 of Algorithm 2, using pre-trained MLP embeddings. Then we use a randomly initialised GCN to compute node embeddings on the input graph G , as well as on G^{ECG} . We visualise these two sets of node embeddings using t-SNE [147] by projecting to a 2D space, attributing the colour of each point based on the node’s ground truth label. We can observe that using the G^{ECG} topology leads to more distinguishable clusters corresponding to the classes even without any training, thus supporting the enhancements in performance when building ECG-GNN.

5.5. Related work

Many specialised architectures have been proposed to tackle GNNs limitations in modeling heterophilic data. H_2GCN [35] proposes separation of ego and neighbour embeddings, using higher-order neighbourhoods and combining representations from intermediate layers. CPGNN [123] learns a compatibility matrix to explicitly integrate information about label correlations and uses it to modulate messages. Similarly, GGNN [146] modifies GCN through degree corrections and signed messages, based on an insight linking heterophily and over-smoothing. FAGNN [131] uses a self-gating mechanism to adaptively integrate low-frequency

Table 5 – Statistics of the original heterophilous graphs and of the evolutionary computation graph obtained from MLP and BGRL.

	roman-empire	amazon-ratings	minesweeper	tolokers	questions
edges	32,927	93,050	39,402	519,000	153,540
edge homophily	0.05	0.38	0.68	0.59	0.84
adjusted homophily	-0.05	0.14	0.01	0.09	0.02
LI	0.11	0.04	0.00	0.01	0.00
ECG($k = 3$) edges	67,986	73,476	30,000	35,274	146,763
MLP-ECG edge homophily	0.73	0.66	0.79	0.79	0.97
MLP-ECG adjusted homophily	0.7	0.53	0.33	0.4	0.41
MLP-ECG LI	0.65	0.33	0.16	0.19	0.28
BGRL-ECG edge homophily	0.16	0.3	0.68	0.6	0.93
BGRL-ECG adjusted homophily	0.06	0.02	0.12	0.08	0.01
BGRL-ECG LI	0.1	0.03	0.05	0.03	0.03

signals, high-frequency signals and raw features and ACM-GNN [132] extends it to enable adaptive channel mixing node-wise. GPRGNN [148] learns Generalized PageRank weights that adjust to node label patterns. FSGNN [149] proposes Feature Selection GNN which separates node feature aggregation from the depth of the GNN through multiplication the node features with different powers and transformations of the adjacency matrix and uses a softmax to select the relevant features. GloGNN [150] leverages global nodes to aggregate information, while GBK-GNN [151] uses bi-kernel feature transformation.

Most relevant to ECG-GNN could be considered GeomGCN [129] and the work of Suresh et al. [137]. The former uses network embedding methods to find neighbouring nodes in the corresponding latent space, to be then aggregated with the neighbours in the original graph, over which a two-level aggregation is then performed. Similarly, [137] modifies the computation graph by computing similarity of degree sequences for different numbers of hops. However, in both cases, the input node features and labels are not used, making it prone to inaccuracies in the graph structure.

However, it was recently pointed [3] that the standard datasets on which these models were tested, such as Squirrel, Chameleon, Cornell, Texas and Wisconsin, had considerable drawbacks: high number of duplicated nodes, highly imbalanced classes and lack of diversity in setups considered. In fact, when evaluated on their newly proposed heterophilic suite, it was noted that most specialised architectures are outperformed by their standard counterparts such as GCN, SAGE, GAT and GT, with only the separation of ego and neighbour embeddings from [35] maintaining an advantage.

5.6. Limitations and further work

We observe that our complementary graph is constructed based on two sources of homophily: a pretrained ResNet model that relies on provided training labels, and a pretrained self-supervised graph module, BGRL, which depends solely on the graph structure and node features without any labels. In cases where neither of these two approaches generates a graph with a satisfactory level of homophily and advantageous connectivity, the complementary graph may struggle to enhance the overall model performance, as evidenced by the relatively smaller gains observed in the `amazon-ratings` dataset.

In such cases, it may be beneficial to explore additional sources for obtaining embeddings. Furthermore, there is potential for improvements in leveraging the complementary information, such as separately using three distinct graphs: the original graph, the MLP-ECG graph, and the BGRL-ECG graph. This approach might prove to be more effective in incorporating the different types of information, rather than relying solely on the projection of concatenated embeddings.

Finally, it is worth noting that while this method enhances the performance of standard graph neural networks, it can also be applied to specialised architectures specifically designed to improve performance on heterophilic data. These two approaches are independent and can in principle be combined to further boost a model’s capabilities. By integrating the complementary graph construction into specialized architectures, we could leverage the benefits of both techniques and potentially achieve even better results when dealing with heterogeneous data. As our paper focuses on the effects of modulating the graph structure using weak classifiers, we apply only commonly-used GNN layers, and leave explorations of this kind to future work.

5.7. Conclusions

We present Evolving Computation Graphs for graph neural networks, ECG-GNN. This is a two-phase method focused on improving the performance of standard GNNs on heterophilous data. Firstly, it builds an evolved computation graph, formed from the original input and a complementary set of edges determined by a weak classifier. Then, the two components of this computation graph are modelled in parallel by two GNNs processors, and projected to the same embeddings space after each propagation layer. This simple and elegant extension of existing graph neural networks proves to be very effective – for four models considered on five diverse heterophilic datasets, the ECG-GNN enhances the performance in 95% of the cases.

Chapter 6

Graph Neural Networks for Reinforcement Learning: Prologue to the third article

6.1. Article Details

Neural Algorithmic Reasoners are Implicit Planners. Andreea Deac, Petar Veličković, Ognjen Milinković, Pierre-Luc Bacon, Jian Tang, Mladen Nikolic. The paper was accepted at NeurIPS 2021 as a Spotlight talk.

Personal contribution: This work started with a self-proposed project [152] I worked on for the course taught by Pierre-Luc, on “Reinforcement Learning and Control”. This project was in the area of Neural Algorithmic Reasoning [19] and it aimed to teach a graph neural network to execute value iteration. Based on its success, we followed up by building an agent that plans using this learnt GNN executor, over a graph built using a transition model. On the implementation side, I worked on the neural executor and the RL loop and Ognjen built the transition model, experimented with pre-training it, as well as providing the visualisations of the agent’s learned representations. Petar, Pierre-Luc, Mladen and Jian provided insights into which environments to use and what ablation studies to perform, for which I was responsible. All authors contributed to writing the paper and revising the article in response to reviewer comments.

6.2. Context

Classical algorithms are fundamental units of computation because they are guaranteed to be correct, consist of modules that can be combined, strongly generalise to unseen inputs and have interpretable operations that address highly relevant downstream problems. On the other hand, neural networks work with raw inputs, can deal with inaccuracies in the data and can be reused in different tasks. The Neural Algorithmic Reasoning (NAR) paradigm [19] proposed having neural networks that learn to execute algorithms, thus obtaining the

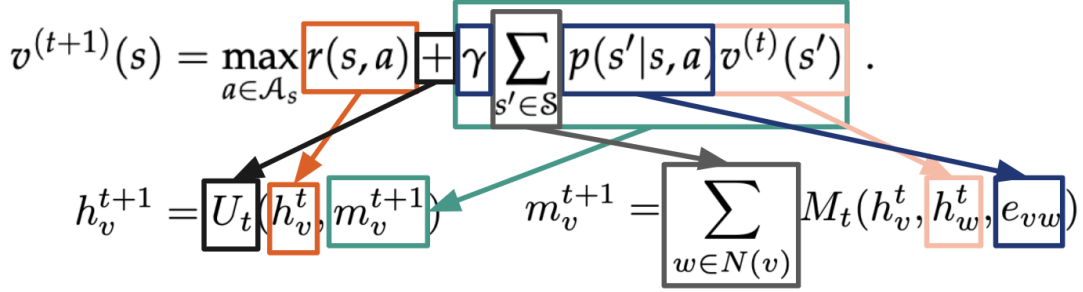


Figure 15 – Correspondences between value iteration and graph convolution

Table 6 – Our models are trained to execute the value iteration algorithm by predicting the values of each MDP state. We show the mean-squared error of the predicted values against ground-truth. Taking the action with the highest expected predicted value using the Bellman equation, we can also predict policies. We then compute the policy accuracy with respect to the ground-truth optimal policy. We test different GNN architectures, noting the MPNN is generally robust to aggregator type, while the less aligned version using attention under-performs in terms of policy accuracy.

Model	MSE			Accuracy		
	$ \mathcal{S} = 20$ $ \mathcal{A} = 5$	$ \mathcal{S} = 50$ $ \mathcal{A} = 10$	$ \mathcal{S} = 100$ $ \mathcal{A} = 20$	$ \mathcal{S} = 20$ $ \mathcal{A} = 5$	$ \mathcal{S} = 50$ $ \mathcal{A} = 10$	$ \mathcal{S} = 100$ $ \mathcal{A} = 20$
MPNN-Sum	0.457	2.175	5.154	97.75	99.3	99.32
MPNN-Mean	0.455	2.199	5.199	98.125	99.3	99.32
MPNN-Max	0.454	2.157	5.119	98.	99.25	99.22
MPNN-2-Sum	0.454	2.159	5.123	98.37	99.4	99.37
Attn-Sum	0.757	1.725	3.765	89.75	90.55	89.69

best of both worlds. In this context, and based on the assumption that there exist algorithms that could perform optimal planning, such as value iteration [40], I have worked on a neural network that learnt to execute value iteration [152] with the idea of leveraging new possibilities for reinforcement learning agents. More precisely, thanks to the alignment between value iteration and graph convolution portrayed in Figure 15, in [152], I designed a graph neural network that can predict value iteration outputs to a high degree of accuracy, as shown in Table 6.

Being able to plan using value iteration could greatly improve a reinforcement learning agent’s abilities. However, for most environments of interest, the inputs to the VI algorithm would have to be approximate, and, as mentioned, classical algorithms operate with the assumption that the inputs are perfect. This assumption is taken in order to be able to study these kinds of hard problems in a purely abstractified setting, free from the noisy environments of the real world. However, this means that algorithms have no associated guarantees of working on incorrect or noisy inputs. On the other hand, neural networks are

known to work with raw inputs. Therefore, in the article that is the focus of the next chapter, “Neural Algorithmic Reasoners are Implicit Planners”, we proposed the eXecuted Latent Value Iteration Network (XLVIN), an agent which uses the neural algorithmic reasoner pre-trained on value iteration as an implicit planner, and combined with neural networks that approximate the inputs to the executor in latent space.

Moreover, as deploying neural algorithmic reasoners proved valuable, I also continued to collaborate on works that developed neural algorithmic reasoners. In the work of Xhonneux et al. [153], for the first time we trained a neural network to learn multiple graph algorithms and gained insights into what is needed to achieve good performance on algorithm multi-tasking. Further, I also took part in designing the Triplet-GMPNN architecture [154], that scaled this idea to all algorithms in the CLRS-30 benchmark [155], learning 30 algorithms with one single multi-task graph neural network.

6.3. Modulator

In XLVIN [24], we demonstrate how computational graphs can be dynamically inferred on-the-fly by leveraging a *learned transition model*, coupled with *breadth-first search*. Such a setup is often necessary when deploying GNN processors in reinforcement learning, as the agent usually does not know *a priori* what the environment transition dynamics are: it must infer these through interacting with the environment. The learned transition model hence indicates the agent’s best current guess of these transition dynamics, from which relevant local computational graphs can be obtained via sampling and search. Accordingly, the modulator of XLVIN consists of a learned transition model, which can be used to sample successor states, and these are then linked to their predecessors with edges in the modulated computation graph.

6.4. Contributions and Research Impact

“Neural Algorithmic Reasoners are Implicit Planners” was one of the first deployments of the NAR paradigm. It was successful in potentiating planning, showing good performance with significantly less data compared to purely model-free baselines or baselines which predicted the VI inputs and used them to run the algorithm itself, on diverse classic-control and Atari environments. Moreover, the ablation studies showed the neural executor is more robust to noise compared to a classic algorithm executor, emphasising the importance of avoiding the *algorithmic bottleneck*.

The algorithmic bottleneck, discovered for the first time in this paper, relates to the need to predict low-dimensional—often *scalar*—values in order to be able to execute an algorithm directly on them. For example, in optimal-trajectory-finding algorithms such as Value Iteration, it is necessary to provide scalar transition probabilities and reward values in every

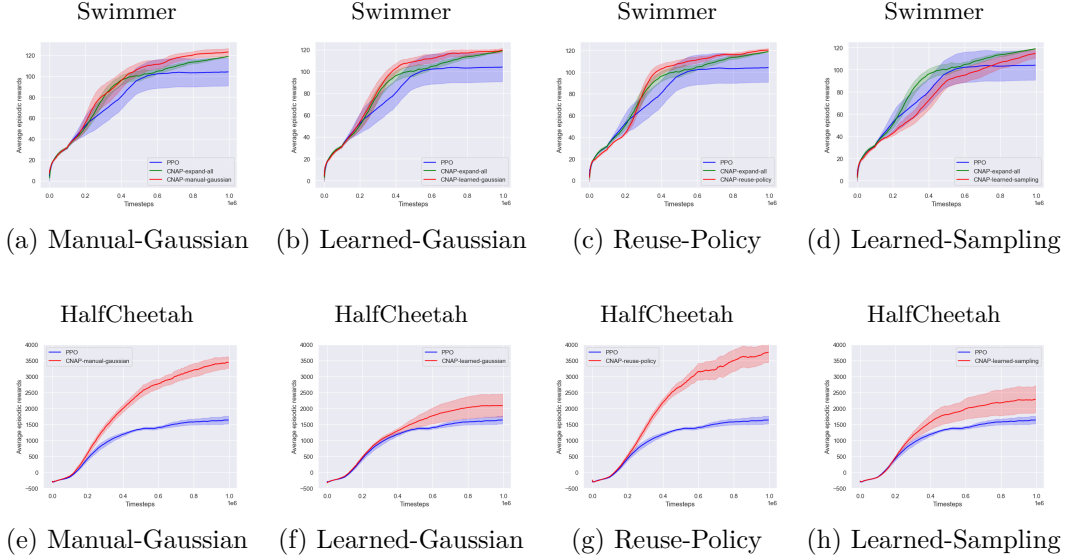


Figure 16 – From [9]: average rewards over time for CNAP (red) and PPO baseline (blue), in Swimmer (action dimension=2) and Halfcheetah (action dimension=6), using different sampling methods. In Swimmer, CNAP with sampling methods were compared with the original version by expanding all actions (green). In (a)(e), the actions were sampled using Gaussian distribution with mean= $N/2$ and std= $N/4$, where N was the number of action bins used to discretize the continuous action space. In (b)(f), two linear layers were used to learn the mean and std, respectively. In (c)(g), the Policy layer was reused in sampling actions to expand. In (d)(h), a separate linear layer was used to learn the optimal neighbor sampling distribution.

node and edge of the MDP graph. However, if the neural network has not observed enough data to reliably compute such inputs, the algorithm may further amplify this problem due to being executed on an incorrect input, with no way to correct for any mistakes downstream. Our executor, being a high-dimensional GNN, avoids this issue altogether, making it easier to deploy downstream in spite of any inaccuracies it may have when executing the algorithm.

In a similar vein to our work, Numeroso et al. [156] use NAR on the max-flow min-cut theorem for brain vessel classification, and Veličković et al. [157] introduce physical laws into the pre-trained executor.

After the XLVIN project, we wanted to see if the framework would also be successful in more extended cases, for example when scaling to large or continuous action spaces, when the algorithm and its application are not fully aligned and the built local MDP is incomplete. Therefore, we worked on “Continuous Neural Algorithmic Planners” [9, CNAP] where we sample actions to build the graph fed as input to the neural executor, matching or improving model-free baselines on MuJoCo environments [158] such as in Figure 16.

Chapter 7

Neural Algorithmic Reasoners are Implicit Planners

Implicit planning has emerged as an elegant technique for combining learned models of the world with end-to-end model-free reinforcement learning. We study the class of implicit planners inspired by *value iteration*, an algorithm that is guaranteed to yield perfect policies in fully-specified tabular environments. We find that prior approaches either assume that the environment is provided in such a tabular form—which is highly restrictive—or infer “local neighbourhoods” of states to run value iteration over—for which we discover an algorithmic bottleneck effect. This effect is caused by explicitly running the planning algorithm based on scalar predictions in every state, which can be harmful to data efficiency if such scalars are improperly predicted. We propose eXecuted Latent Value Iteration Networks (XLVINS), which alleviate the above limitations. Our method performs all planning computations in a high-dimensional *latent space*, breaking the algorithmic bottleneck. It maintains alignment with value iteration by carefully leveraging neural graph-algorithmic reasoning and contrastive self-supervised learning. Across eight low-data settings—including classical control, navigation and Atari—XLVINS provide significant improvements to data efficiency against value iteration-based implicit planners, as well as relevant model-free baselines. Lastly, we empirically verify that XLVINS can closely align with value iteration.

7.1. 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* [159–161, 48, 162–164]. Such approaches propose inductive biases in the policy function to enable planning to emerge, while training the policy

in a model-free manner. Accordingly, implicit planners combine the effectiveness of large-scale neural network training with the data efficiency promises of planning, making them a very attractive research direction.

Many popular implicit planners attempt to align with the computations of the value iteration (VI) algorithm within a policy network [159, 160, 162, 165, 166].

As VI is a differentiable algorithm, guaranteed to find the *optimal* policy, it can combine with gradient-based optimisation and provides useful theoretical guarantees. We also recognise the potential of VI-inspired deep RL, hence it is our primary topic of study here. However, applying VI assumes that the underlying RL environment (a) is *tabular*, and that its (b) *transition* and (c) *reward* distributions are both **fully known** and provided upfront. Such assumptions are unfortunately unrealistic for most environments of importance to RL research. Very often the dynamics of the environment will not be even partially known, and the state space may either be continuous (e.g. for control tasks) or very high-dimensional (e.g. for pixel-space observation in Atari), making a tabular representation hard to realise from a storage complexity perspective.

Accordingly, VI-based implicit planners often offer representation learning based solutions for alleviating some of the above limitations. Impactful early work [159, 162, 166] showed that, in tabular settings with known transition dynamics, the reward distribution and VI computations can be approximated by a (graph) convolutional network. While highly insightful, this line of work still does not allow for RL in generic non-tabular environments with unobserved dynamics. Conversely, approaches such as ATreeC [165] and VPN [160] lift the remaining two requirements, by using a latent transition model to construct a “local environment” around the current state. They then use learned models to predict scalar rewards and values in every node of this environment, applying VI-style algorithms directly.

While such approaches apparently allow for seamless VI-based implicit planning, we discover that the prediction of scalar signals represents an *algorithmic bottleneck*: if the neural network has observed insufficient data to properly estimate these scalars, the predictions of the VI algorithm will be equally suboptimal. This is limiting in low-data regimes, and can be seen as unfavourable, particularly given that one of the main premises of implicit planning is improved data efficiency.

In this paper, we propose the **eXecuted Latent Value Iteration Network** (XLVIN), an implicit planning policy network which embodies the computation of VI while addressing *all* of the limitations mentioned previously. We retain the favourable properties of prior methods while simultaneously performing VI in a high-dimensional latent space, removing the requirement of predicting scalars and hence *breaking* the algorithmic bottleneck. We enable this high-dimensional VI execution by leveraging the latest advances in neural algorithmic reasoning [19]. This emerging area of research seeks to emulate iterations of classical algorithms (such as VI) directly within neural networks. As a result, we are able to seamlessly

run XLVINs with minimal configuration changes on a wide variety of discrete-action environments, including pixel-based ones (such as Atari), fully continuous-state control and navigation. Empirically, the XLVIN agent proves favourable in low-data environments against relevant model-free baselines as well as the ATreeC family of models.

Our contributions are thus three-fold: (a) we provide a detailed overview of the prior art in value iteration-based implicit planning, and discover an *algorithmic bottleneck* in impactful prior work; (b) we propose the XLVIN implicit planner, which breaks the algorithmic bottleneck while retaining the favourable properties of prior work; (c) we demonstrate a successful application of neural algorithmic reasoning within reinforcement learning, both in terms of quantitative analysis of XLVIN’s data efficiency in low-data environments, and qualitative alignment to VI.

7.2. Background and related work

We will now present the context of our work, by gradually surveying the key developments which bring VI into the implicit planning domain and introducing the building blocks of our XLVIN agent.

Planning has been studied under the umbrella of model-based RL [167, 8, 168]. However, having a good model of the environment’s dynamics is essential before being able to construct a good plan. We are instead interested in leveraging the progress of model-free RL [46, 42] by enabling planning through inductive biases in the policy network—a direction known as implicit planning. The planner could also be trained to optimise a supervised imitation learning objective [169, 170]. This is performed by UPNs [169] in a goal-conditioned setting. Our differentiable executors are instead applicable across a wide variety of domains where goals are not known upfront. Diff-MPC [170] leverages an algorithm in an explicit manner. However, explicit use of the algorithm often has issues of requiring a bespoke backpropagation rule, and the associated low-dimensional bottlenecks.

Throughout this section we pay special attention to implicit planners based on VI, and distinguish two categories of previously proposed planners: models which assume fixed and known environment dynamics [159, 162, 166] and models which derive scalars to be used for VI-style updates [165, 160].

Value iteration (VI) is a successive approximation method for finding the optimal value function of a discounted *Markov decision process* (MDPs) as the fixed-point of the so-called Bellman optimality operator [41]. 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*, $R : \mathcal{S} \times \mathcal{A} \rightarrow \mathbb{R}$ is a reward function, $P : \mathcal{S} \times \mathcal{A} \rightarrow \text{Dist}(\mathcal{S})$ is a *transition function* such that $P(s'|s,a)$ 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* $\pi : \mathcal{S} \times \mathcal{A} \rightarrow [0,1]$ 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, to find a π^* it suffices to find the unique optimal value function V^* as the fixed-point of the Bellman optimality operator. The optimal value function V^* is such a fixed-point and satisfies the *Bellman optimality equations* [40]: $V^*(s) = \max_{a \in \mathcal{A}} (R(s, a) + \gamma \sum_{s' \in \mathcal{S}} P(s'|s, a) V^*(s'))$. Accordingly, VI randomly initialises a value function $V_0(s)$, and then iteratively updates it as follows:

$$V_{t+1}(s) = \max_{a \in \mathcal{A}} \left(R(s, a) + \gamma \sum_{s' \in \mathcal{S}} P(s'|s, a) V_t(s') \right) . \quad (7.2.1)$$

VI is thus a powerful technique for optimal control in RL tasks, but its applicability hinges on knowing the MDP parameters (especially P and R) upfront—which is unfortunately not the case in most environments of interest. To make VI more broadly applicable, we need to leverage function approximators (such as neural networks) and representation learning to estimate such parameters.

Value iteration is message passing Progress towards broader applicability started by lifting the requirement of knowing R . Several implicit planners, including (G)VIN [159, 162] and GPPN [166], were proposed for discrete environments where P is fixed and known. Observing the VI update rule (Equation 7.2.1), we may conclude that it derives values by considering features of *neighbouring* states; i.e. the value $V(s)$ is updated based on states s' for which $P(s'|s, a) > 0$ for some a . Accordingly, it tightly aligns with *message passing* over the graph corresponding to the MDP, and hence a graph neural network (GNN) [13] over the MDP graph may be used to estimate the value function.

Graph neural networks (GNNs) have been intensively studied as a tool to process graph-structured inputs, and were successfully applied to various RL tasks [171–173]. For each state s in the graph, a set of messages is computed—one message for each neighbouring node s' , derived by applying a *message function* M to the relevant node $(\mathbf{h}_s, \mathbf{h}_{s'})$ and edge $(\mathbf{e}_{s' \rightarrow s})$ features. Incoming messages in a neighbourhood $\mathcal{N}(s)$ are then aggregated through a permutation-invariant operator \bigoplus (such as sum or max), obtaining a summarised message \mathbf{m}_s :

$$\mathbf{m}_s = \bigoplus_{s' \in \mathcal{N}(s)} M(\mathbf{h}_s, \mathbf{h}_{s'}, \mathbf{e}_{s' \rightarrow s}) \quad (7.2.2)$$

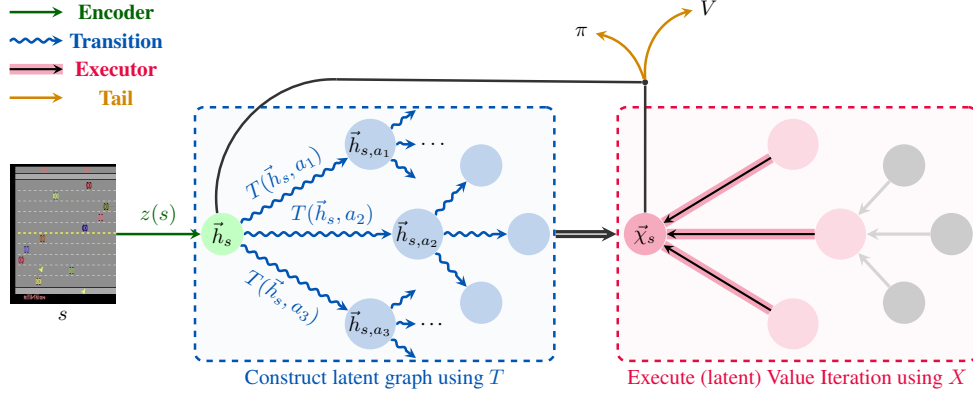


Figure 17 – XLVIN model summary. Its modules are explained (and colour-coded) in Section 7.3.1.

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

$$\mathbf{h}'_s = U(\mathbf{h}_s, \mathbf{m}_s) \quad (7.2.3)$$

GNNs can then emulate VI computations by setting the neighbourhoods according to the transitions in P ; that is, $\mathcal{N}(s) = \{s' \mid \exists a \in \mathcal{A}. P(s'|s, a) > 0\}$. For the special case of grid worlds, the neighbours of a grid cell correspond to exactly its neighbouring cells, and hence the rules in Equations 7.2.2–7.2.3 amount to a convolutional neural network over the grid [159].

While the above blueprint yielded many popular implicit planners, the requirement of knowing upfront the transition neighbourhoods is still limiting. Ideally, we would like to be able to, on-the-fly, generate states s' that are reachable from s as a result of applying some action. Generating neighbouring state representations corresponds to a learned *transition model*, and we present one such method, which we employed within XLVIN, next.

TransE The TransE [65] loss for embedding objects and relations can be adapted to RL [174, 5]. State embeddings are obtained by an *encoder* $z : \mathcal{S} \rightarrow \mathbb{R}^k$ and the effect of an action in a given state is modelled by a translation model $T : \mathbb{R}^k \times \mathcal{A} \rightarrow \mathbb{R}^k$. 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 taking action a in state s . This embedding should be as close as possible to $z(s')$, for the observed transition (s, a, s') , and also far away from negatively sampled state embeddings $z(\tilde{s})$. Therefore, the embedding function is optimised using the following variant of the triplet loss (with hinge hyperparameter ξ):

$$\mathcal{L}_{\text{TransE}}((s, a, s'), \tilde{s}) = d(z(s) + T(z(s), a), z(s')) + \max(0, \xi - d(z(\tilde{s}), z(s'))) \quad (7.2.4)$$

Having a trained T function, it is now possible to dynamically construct $\mathcal{N}(s)$. For every action a , applying $T(\mathbf{h}_s, a)$ yields embeddings $\mathbf{h}_{s,a}$ which correspond to one neighbour state embedding in $\mathcal{N}(s)$. We can, of course, roll out T further from $\mathbf{h}_{s,a}$ to simulate longer

trajectory outcomes—the amount of steps we do this for is denoted as the “thinking time”, K , of the planner. With neighbourhoods constructed, one final question remains: how to apply VI over them, especially given that the embeddings \mathbf{h}_s are high-dimensional, and VI is defined over scalar reward/value inputs?

If we also train a reward model, $R(\mathbf{h}_s, a)$, and a state-value function, $V(\mathbf{h}_s)$ from state embeddings, we can attach scalar values and rewards to our synthesised graph. Then, VI can be directly applied over the constructed tree to yield the final policy. As the VI update (Equation 7.2.1) is differentiable, it composes nicely with neural estimators and standard RL loss functions. Further, R and V can be directly trained from observed rewards and returns when interacting with the environment. This approach inspired a family of powerful implicit planners, including VPN [160], TreeQN and ATreeC [165]. However, it remains vulnerable to a specific bottleneck effect, which we discuss next.

Algorithmic bottleneck Through our efforts of learning transition and reward models, we reduced our (potentially highly complex) input state s into an abstractified graph with scalar values in its nodes and edges, so that VI can be directly applied. However, VI’s performance guarantees rely on having the *exactly correct* parameters of the underlying MDP. If there are any errors in the predictions of these scalars, they may propagate to the VI operations and yield suboptimal policies. We will study this effect in detail on synthetic environments in Section 7.4.3.

As the ATreeC-style approaches commit to using the scalar produced by their reward models, there is no way to recover from a poorly predicted value. This leaves the model vulnerable to an *algorithmic bottleneck*, especially early on during training when insufficient experience has been gathered to properly estimate R . Accordingly, the agent may struggle with data efficiency.

With our XLVIN agent, we set out to break this bottleneck, and do not project our state embeddings \mathbf{h}_s further to a low-dimensional space. This amounts to running a graph neural network directly over them. Given that these same embeddings are optimised to produce plausible graphs (via the TransE loss), how can we ensure that our GNN will stay aligned with VI computations?

Algorithmic reasoning An important research direction explores the use of neural networks for learning to execute algorithms [175, 19]—which was recently extended to algorithms on graph-structured data [176]. In particular, [6] establishes *algorithmic alignment* between GNNs and dynamic programming algorithms. Furthermore, [176] 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, and good results on executing VI emerged on synthetic graphs [152]—an observation we strongly leverage here.

7.3. XLVIN Architecture

Next, we specify the computations of the eXecuted Latent Value Iteration Network (XLVIN). We recommend referring to Figure 17 for a visualisation of the model dataflow (more compact overview in Appendix C.1) and to Algorithm 3 for a step-by-step description of the forward pass.

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 action $a \in \mathcal{A}$ in that state. Here, θ are the policy parameters, to be optimised with gradient ascent.

7.3.1. XLVIN modules

Encoder The encoder function, $z : \mathcal{S} \rightarrow \mathbb{R}^k$, consumes state representations $s \in \mathcal{S}$ and produces flat embeddings, $\mathbf{h}_s = z(s) \in \mathbb{R}^k$. 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, $T : \mathbb{R}^k \times \mathcal{A} \rightarrow \mathbb{R}^k$, models the effects of taking actions, in the *latent* space. Accordingly, it consumes a state embedding $z(s)$ and an action 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 Equation 7.3.1 and it is commonly realised as an MLP.

$$z(s) + T(z(s), a) \approx \mathbb{E}_{s' \sim P(s'|s,a)} z(s') \quad (7.3.1)$$

Executor The executor function, $X : \mathbb{R}^k \times \mathbb{R}^{|\mathcal{A}| \times k} \rightarrow \mathbb{R}^k$, processes an embedding \mathbf{h}_s of a given state s , alongside a neighbourhood set $\mathcal{N}(\mathbf{h}_s)$, which contains (expected) embeddings of states that immediately neighbour s —for example, through taking actions. Hence,

$$\mathcal{N}(\mathbf{h}_s) \approx \left\{ \mathbb{E}_{s' \sim P(s'|s,a)} z(s') \right\}_{a \in \mathcal{A}} \quad (7.3.2)$$

The executor combines the neighbourhood set features to produce an updated embedding of state s , $\chi_s = X(\mathbf{h}_s, \mathcal{N}(\mathbf{h}_s))$, 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, $A : \mathbb{R}^k \times \mathbb{R}^k \rightarrow [0, 1]^{|\mathcal{A}|}$ consumes the state embedding \mathbf{h}_s and the updated state embedding χ_s , producing action probabilities $\pi_\theta(a|s) = A(\mathbf{h}_s, \chi_s)_a$, 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 . For example, we train XLVINs using proximal policy optimisation (PPO) [46], which necessitates a state-value function: $V(\mathbf{h}_s, \chi_s)$.

Algorithm 3: XLVIN forward pass

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

$\mathbf{h}_s = z(s)$; // Embed the input state with the encoder
 $\mathbb{S}^0 = \{\mathbf{h}_s\}, \mathbb{E} = \emptyset$
for $k \in [0, K)$ **do**
 $\mathbb{S}^{k+1} = \emptyset$; // Initialise depth- $(k+1)$ embeddings
 for $\mathbf{h} \in \mathbb{S}^k, a \in \mathcal{A}$ **do**
 $\mathbf{h}' = \mathbf{h} + T(\mathbf{h}, a)$; // Get (expected) neighbour embedding
 $\mathbb{S}^{k+1} = \mathbb{S}^{k+1} \cup \{\mathbf{h}'\}, \mathbb{E} = \mathbb{E} \cup \{(\mathbf{h}, \mathbf{h}', a)\}$; // Attach \mathbf{h}' to the graph
/* 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 $\mathbf{h} = X(\mathbf{h}, \mathcal{N}(\mathbf{h}))$, for
every embedding $\mathbf{h} \in \mathbb{S}$, for K steps. */
 $\chi_s = \text{EXECUTE}(\mathbf{h}_s, \bigcup_{k=0}^K \mathbb{S}^k, \mathbb{E}, X, K)$; // See Appendix C.2 for details on the
EXECUTE function
/* Use the actor and tail to predict the policy and value functions
from the (updated) state embedding of s */
 $\pi_\theta(s, \cdot) = A(\mathbf{h}_s, \chi_s), V(s) = V(\mathbf{h}_s, \chi_s)$

Discussion. 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, while avoiding the algorithmic bottleneck. This achieves all of our initial aims.

The transition function produces state embeddings that correspond to the expectation of the successor state embedding over all possible outcomes (Equation 7.3.1). While taking expectations is an operation that aligns well with VI computations, it can pose limitations in the case of non-deterministic MDPs. This is because the obtained expected latent states may not be trivially further expandable, should we wish to plan further from them. One possible remedy we suggest is employing a probabilistic (e.g. variational) transition model from which we could repeatedly sample concrete next-state latents.

Our tree expansion strategy is *breadth-first*, which expands every action from every node, yielding $O(|\mathcal{A}|^K)$ time and space complexity. While this is prohibitive for scaling up K , we empirically found that performance plateaus by $K \leq 4$ for all studied environments, mirroring prior findings [165]. Even if these are shallow trees of states, we anticipate a compounding effect from optimising TransE together with PPO’s value/policy heads. We

defer allowing for deeper expansions and large action spaces to future work, but note that it will likely require a *rollout policy*, selecting actions to expand from a given state. For example, I2A [161] obtains a rollout policy by distilling the agent’s policy network. Extensions to continuous actions could also be achieved by rollout policies, or discretising the action space by binning [177].

7.3.2. XLVIN Training

As discussed, the success of XLVIN relies on our transition function, T , constructing plausible graphs, and our executor function, X , reliably imitating VI steps in a high dimensional latent space. Accordingly, we train both of them using established methods: TransE for T and [152] for X .

To optimise the neural network parameters θ , we use proximal policy optimisation (PPO)¹ [46]. Note that the PPO gradients also flow into T and X , which could *displace* them from the properties required by the above, leading to either poorly constructed graphs or lack of VI-aligned computation.

Without knowledge of the underlying MDP, we have no easy way of training the executor, X , online. We instead opt to *pre-train* the parameters of X and *freeze* them, treating them as constants rather than parameters to be optimised. In brief, the executor pre-training proceeds by first **generating** a dataset of synthetic MDPs, according to some underlying graph distribution. Then, we **execute** the VI algorithm on these MDPs by iterating Equation 7.2.1, keeping track of intermediate values $V_t(s)$ at each step t , until convergence. Finally, we **supervise** a GNN (operating over the MDP transitions as edges) to receive $V_t(s)$ —and all other parameters of the MDP—as inputs, and predict $V_{t+1}(s)$ (optimised using mean-squared error). Such a graph neural network has three parts: an *encoder*, mapping $V_t(s)$ to a latent representation, a *processor*, which performs a step of VI in the latent space, and a *decoder*, which decodes back $V_{t+1}(s)$ from the latent space. We only **retain** the *processor* as our executor function X , in order to avoid the algorithmic bottleneck in our architecture.

For the transition model, we found it sufficient to optimise TransE (Equation 7.2.4) using only on-policy trajectories. However, we do anticipate that some environments will require a careful tradeoff between exploration and exploitation for the data collection strategy for training the transition model.

Thus, after pre-training the GNN to predict one-step value iteration updates and freezing the processor, a step of the training algorithm corresponds to:

- (1) Sample on-policy rollouts (with multiple parallel actors acting for a fixed number of steps).

1. We use the PPO implementation and hyperparameters from Kostrikov [178].

- (2) Based on the transitions in these rollouts, evaluate the PPO [46] and TransE (Equation 7.2.4) losses. Negative sample states for TransE, \tilde{s} , are randomly sampled from the rollouts.
- (3) Update the policy network’s parameters, θ , using the combined loss. It is defined, for a single rollout, $\mathcal{T} = \{(s_t, a_t, r_t, s_{t+1})\}_t$, as follows:

$$\mathcal{L}(\mathcal{T}) = \mathcal{L}_{\text{PPO}}(\mathcal{T}) + \lambda \sum_{i=1}^{|\mathcal{T}|} \mathcal{L}_{\text{TransE}}((s_i, a_i, s_{i+1}), \tilde{s}_i) \quad \tilde{s}_i \sim \mathbb{P}(s|\mathcal{T}) \quad (7.3.3)$$

where we set $\lambda = 0.001$, and $\mathbb{P}(s|\mathcal{T})$ is the empirical distribution over the states in \mathcal{T} .

It should be highlighted that our approach can easily be modified to support value-based methods such as DQN [42]—merely by modifying the tail component of the network and the RL loss function.

7.4. Experiments

We now deploy XLVINS in *generic* discrete-action environments with unknown MDP dynamics (further details in Appendix C.3), and verify their potential as an implicit planner. Namely, we investigate whether XLVINS provide gains in data efficiency, by comparing them in low-data regimes against a relevant model-free PPO baseline, and ablating against the ATreeC implicit planner [165], which executes an explicit TD(λ) backup instead of a latent-space executor, and is hence prone to algorithmic bottlenecks. All chosen environments were previously explicitly studied in the context of planning within deep reinforcement learning, and were identified as environments that benefit from planning computations in order to generalise [21, 165, 160, 179].

7.4.1. Experimental setup

Common elements On all environments, the transition function, T , is a three-layer MLP with layer normalisation [180] after the second layer. The executor, X , is, for all environments, identical to the message passing executor of [152]. We train the executor from completely random deterministic graphs—making no assumptions on the underlying environment’s topology.

Continuous-space We focus on four OpenAI Gym environments [181]: classical continuous-state control tasks—CartPole, Acrobot and MountainCar, and a continuous-state spaceship navigation task, LunarLander. In all cases, we study data efficiency by presenting extremely limited data scenarios.

The encoder function is a three-layer MLP with ReLU activations, computing 50 output features and F hidden features, where $F = 64$ for CartPole, $F = 32$ for Acrobot, $F = 16$ for

MountainCar and $F = 64$ for LunarLander. The same hidden dimension is also used in the transition function MLP.

As before, we train our executor from random deterministic graphs. In this setting only, we also attempt to illustrate the potential benefits when the graph distribution is biased by our beliefs of the environment’s topology. Namely, we attempt training the executor from synthetic graphs that 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, for both of these approaches, is given in Appendix C.5. The same trained executor is then deployed across all environments, to demonstrate robustness to synthetic graph construction. For LunarLander, the XLVIN uses $K = 3$ executor layers; in all other cases, $K = 2$.

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, the remaining environments are all sparse-reward, and known to be challenging for policy gradient methods. For these environments, we sample 5 trajectories at a time, twenty times during training (for a total of 100 trajectories) for Acrobot and MountainCar, and fifty times during training (for a total of 250 trajectories) for LunarLander.

Pixel-space Lastly, we investigate how XLVINs perform on high-dimensional pixel-based observations, using the Atari-2600 [182]. We focus on four games: Freeway, Alien, Enduro and H.E.R.O.. These environments encompass various aspects of complexity: sparse rewards (in Freeway), larger action spaces (18 actions for Alien and H.E.R.O.) and visually rich observations (changing time-of-day on Enduro) and long-range credit assignment (on H.E.R.O.). Further, we successfully *re-use* the executor trained on random deterministic graphs, showing its transfer capability across vastly different settings. We evaluate the agents’ low-data performance by allowing only 1,000,000 observed transitions. We re-use exactly the environment and encoder from Kostrikov [178], and run the executor for $K = 2$ layers for Freeway and Enduro and $K = 1$ for Alien and H.E.R.O..

7.4.2. Results

In our results, we use “**XLVIN-CP**” to denote XLVIN executors pre-trained using CartPole-style synthetic graphs (where applicable), and “**XLVIN-R**” for pre-training them on random deterministic graphs. “**PPO**” denotes our baseline model-free agent; it has no transition/executor model, but otherwise matches the XLVIN hyperparameters. As planning-like computation was shown to emerge in entirely model-free agents [164], this serves as a check for the importance of the VI computation.

Table 7 – Mean scores for low-data CartPole-v0, Acrobot-v1, MountainCar-v0 and LunarLander-v2, averaged over 100 episodes and five seeds.

Agent	CartPole-v0	Acrobot-v1	MountainCar-v0	LunarLander-v2
	10 trajectories	100 trajectories	100 trajectories	250 trajectories
PPO	104.6 \pm 48.5	-500.0 \pm 0.0	-200.0 \pm 0.0	90.52 \pm 9.54
ATreeC	117.1 \pm 56.2	-500.0 \pm 0.0	-200.0 \pm 0.0	84.04 \pm 5.35
XLVIN-R	199.2 \pm 1.6	-353.1 \pm 120.3	-185.6 \pm 8.1	99.34 \pm 6.77
XLVIN-CP	195.2 \pm 5.0	-245.4 \pm 48.4	-168.9 \pm 24.7	N/A

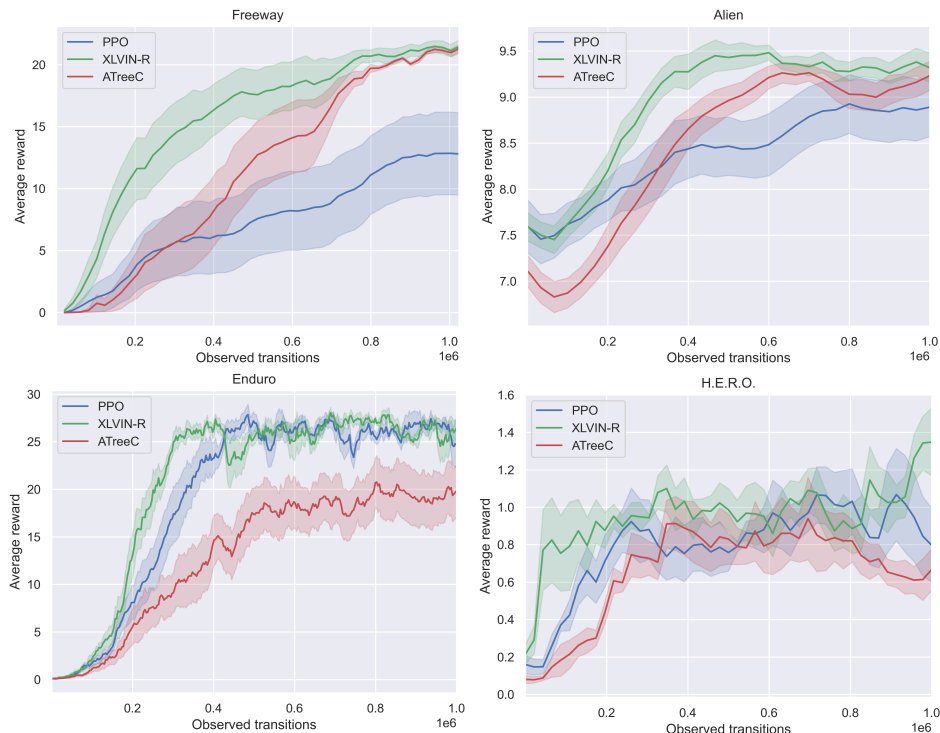


Figure 18 – Average clipped reward on Freeway, Alien, Enduro and H.E.R.O. over 1,000,000 transitions and ten seeds.

To analyse the impact of the algorithmic bottleneck, we use “**ATreeC**” [165] as one of our baselines, capturing the behavior of a larger class of VI-based implicit planners (including TreeQN and VPN [160]). For maximal comparability, we make ATreeC fully match XLVIN’s hyperparameters, except for the executor model. Since ATreeC’s policy is directly tied to the result of applying TD(λ), its ultimate performance is closely tied to the quality of its scalar value predictions. Comparing against ATreeC can thus give insight into negative effects of the algorithmic bottleneck at low-data regimes.

CartPole, Acrobot, MountainCar and LunarLander Results for the continuous-space control environments are provided in Table 7. We find that the XLVIN model solves

CartPole from only 10 trajectories, outperforming all the results given in [5] (incl. REINFORCE [43], Autoencoders, World Models [183], DeepMDP [184] and PRAE [5]), while using $10\times$ fewer samples. For more details, see Appendix C.4.

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, as well as ATreeC, are unable to get off the ground at all, remaining stuck at the lowest possible score in the environment until timing out. This still holds when XLVIN is trained on the random deterministic graphs, demonstrating that the executor training need not be dependent on knowing the underlying MDP specifics.

Mirroring the above findings, XLVIN also demonstrates clear low-data efficiency gains on LunarLander, compared to the PPO baseline and ATreeC. In this setting, ATreeC even underperformed compared to the PPO baseline, highlighting once again the issues with algorithmic bottlenecks.

Freeway, Alien, Enduro and H.E.R.O. Lastly, the average clipped reward of the Atari agents across the first million transitions can be visualised in Figure 18. From the inception of the training, the XLVIN model explores and exploits better, consistently remaining ahead of the baseline PPO model in the low-data regime (matching it in the latter stages of Enduro). In H.E.R.O., XLVIN is also the first agent to break away from the performance plateau, towards the end of the 1,000,000 transitions. The fact that the executor was transferred from randomly generated graphs (Appendix C.5) is a further statement to XLVIN’s robustness.

On all four games, ATreeC consistently trailed behind XLVIN during the first half of the training, and on Enduro, it underperformed even compared to the PPO baseline, indicating that overreliance on scalar predictions may damage low-data performance. It empirically validates our observation of the potential negative effects of algorithmic bottlenecks at low-data regimes.

7.4.3. Qualitative results

The success of XLVIN hinges on the appropriate operation of its two modules; the transition model T and the executor GNN X . In this section, we qualitatively study these two components, hoping to elucidate the mechanism in which XLVIN organises and executes its plan.

To faithfully ground the predictions of T and X on an underlying MDP, we analyse a pre-trained XLVIN agent with a CNN encoder and $K = 4$ executor steps on randomly-generated 8×8 grid-world environments. We chose grid-worlds because V^* can be computed exactly. We train on mazes of progressively increasing difficulty (expressed in terms of their *level*: the shortest-path length from the start state to the goal). We move on to the next level once

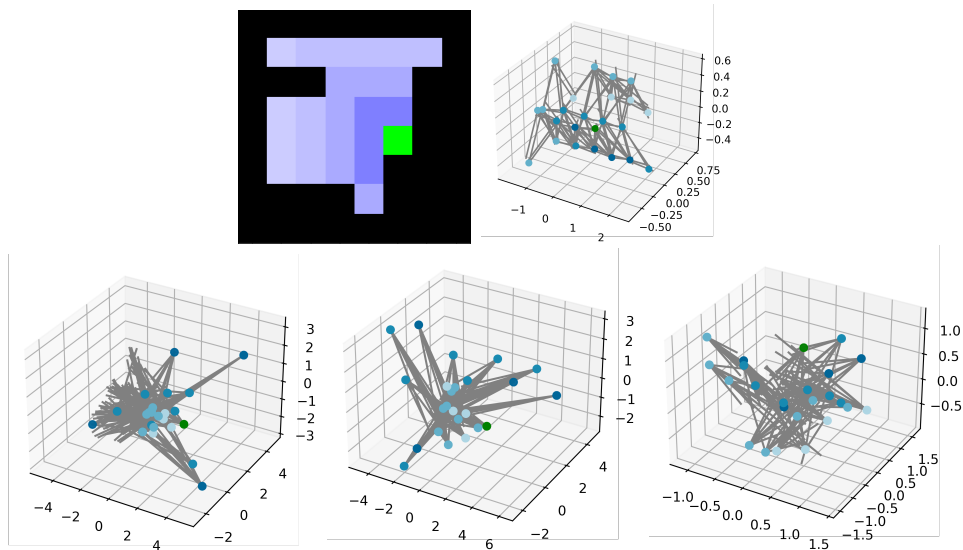


Figure 19 – Top: A test maze (*left*) and the PCA projection of its TransE state embeddings (*right*), colour-coded by distance to goal (in green). **Bottom:** PCA projection of the XLVIN state embeddings after passing the first (*left*), second (*middle*), and ninth (*right*) level of the continual maze.

the agent solves at least 95% of the mazes from the current level. On these environments, we generally found that XLVIN is competitive with implicit planners that are aware of the grid-world structure. See Appendix C.6 for details, including the hyperparameters of the XLVIN architecture.

Projecting the embeddings We begin by qualitatively studying the embeddings learnt by the encoder and transition model. At the top row of Figure 19, we (*left*) colour-coded a specific held-out 8×8 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), with the edges induced by the transition model. Such a model merely seeks to organise the data, rather than optimise for returns: hence, a grid-like structure emerges.

At the bottom row, we visualise how these embeddings and transitions evolve as the agent keeps solving levels; at levels one (*left*) and two (*middle*), the embedder learnt to distinguish all 1-step and 2-step neighbours of the goal, respectively, by putting them on opposite parts of the projection space. 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.

V^* predictability We hypothesise that the encoder function z is tasked with learning to map raw states to a latent space where the executor GNN can operate properly, and then the GNN performs VI-aligning computations in this latent space. We provide a qualitative study

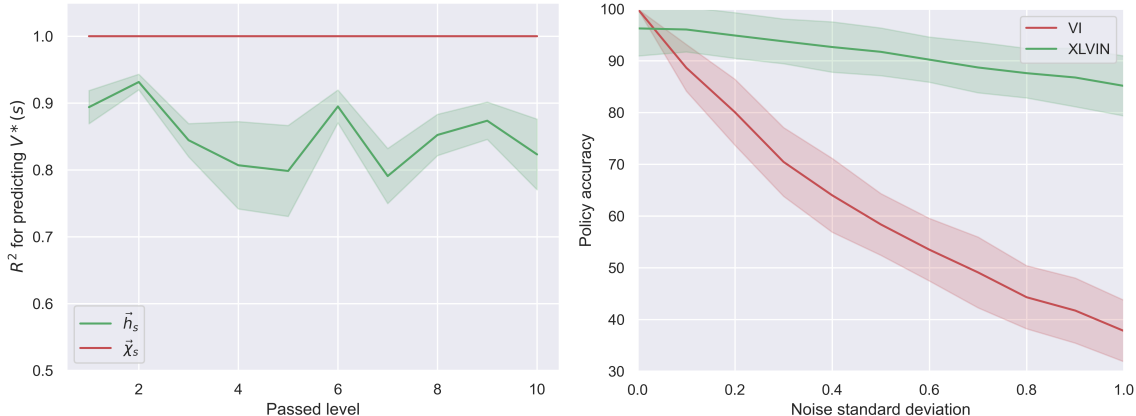


Figure 20 – **Left:** V^* predictability–Coefficient of determination from linearly regressing on the state embeddings obtained from the encoder (green) and from the executor (red). **Right:** Algorithmic bottleneck–Policy accuracy from introducing Gaussian noise in the scalar input fed into VI (red) and in the embeddings fed into the XLVIN executor (green).

to validate this: for all positions in held-out 8×8 test mazes, we computed the ground-truth values $V^*(s)$ (using VI), and performed linear regression to test how accurately they are decodable from the XLVIN state embeddings before (\mathbf{h}_s) and after (χ_s) applying the GNN. We computed the R^2 goodness-of-fit measure, after passing each of the ten training levels. Our results (Figure 20) are strongly in support of the hypothesis: while the embedding function ($z(s) = \mathbf{h}_s$; in green) is already reasonably predictive of $V^*(s)$ ($R^2 \approx 0.85$ on average), after the GNN computations are performed, the recovered state embeddings (χ_s ; in red) are consistently almost-perfectly linearly decodable to VI outputs $V^*(s)$ ($R^2 \approx 1$). Hence, the encoder maps s to a latent space from which the executor can perform VI.

Algorithmic bottleneck In formulating the algorithmic bottleneck, we assume that inaccuracies in the scalar values used for VI will have a larger impact on degrading the performance than perturbations in high-dimensional state embeddings inputted to the executor. To faithfully evaluate policy accuracy, we study this sensitivity on the randomly generated MDPs used to train the executor. Here, ground-truth values V^* can be explicitly computed, and we can compare the recovered policy directly to the greedy policy over these values. Hence, we study the effect of introducing Gaussian noise in the scalar inputs fed to VI compared to introducing Gaussian noise in the high-dimensional latents fed into the XLVIN executor. In Figure 20, we plot the policy accuracy as a function of noise standard deviation, showing that, while XLVIN is unable to predict policies perfectly at zero noise, it quickly dominates the VI’s policy predictions once the noise magnitude increases. Besides indicating that imperfections in TransE outputs can be handled with reasonable grace, this experiment provides direct evidence of the algorithmic bottleneck: errors in scalar inputs to an algorithm can impact its predictions substantially, while a high-dimensional latent space executor is able to more gracefully handle such perturbations.

7.5. Conclusions

We presented eXecuted Latent Value Iteration Networks (XLVINs), combining recent progress in self-supervised contrastive learning, graph representation learning and neural algorithm execution for implicit planning on irregular, continuous or unknown MDPs. Our results showed that XLVINs match or outperform appropriate baselines, often at low-data or out-of-distribution regimes. The learnt executors are robust and transferable across environments, despite being trained on purely random graphs. XLVINs represent, to the best of our knowledge, one of the first times neural algorithmic executors are used for implicit planning, and they successfully break the algorithmic bottleneck.

Chapter 8

Geometric Deep Learning for Reinforcement Learning: Prologue to the fourth article

8.1. Article Details

Equivariant MuZero. Andreea Deac, Théophane Weber, George Papamakarios. The paper is under submission at TMLR and was awarded an Oral Presentation at ICLR 2023 Domain Generalisation workshop.

Personal contribution: This work was done as part of my internship at DeepMind. Théo, George and I brainstormed together for the project idea, with George providing insights on group theory and Théo advising on the reinforcement learning side, and providing valuable information on MuZero implementations. I was responsible for the project’s full implementation and evaluation, with advice from George on equivariant layer implementation details. I ran all the experiments and wrote the proof of the agent’s equivariance. All authors participated in writing and improving the manuscript.

8.2. Context

After working on XLVIN, a question arose: in what other ways can geometric deep learning (GDL) help reinforcement learning? Reinforcement learning algorithms have been successful thanks to their general applicability, but at the same time their generality is also cause for inefficiency. In cases such as molecular discovery or robotics, data is limited and the function search space could be reduced by integrating known symmetries (such as equivariances under different camera angles). Therefore, encouraged by related works such as [185, 186], and with the underlying idea that model-based RL could be an even better target for GDL by grounding the world model using laws of the environment, we proposed Equivariant MuZero.

8.3. Modulator

In Equivariant MuZero [25], we demonstrate how to improve the low-data performance and robustness of state-of-the-art reinforcement learning agents by taking into account *geometric constraints*. Specifically, by exploiting the symmetries assumed in the RL environment, we are able to define and modulate specialised computational graphs between various *views* of that environment—for example, all possible 2D rotations of it. The computational graph is defined in a way that respects the assumed symmetries of the environment, and leads to significant empirical gains in data efficiency, along with provable equivariance properties of the underlying planning algorithm—in the case of MuZero, this is Monte Carlo tree search (MCTS). The modulator of the EqMuZero agent hence designs the computation graph based on the structure of the symmetry group used to capture the geometric constraints. It is worth noting that the processor networks in the context of EqMuZero are also geometrically constrained, and hence more specialised than off-the-shelf CNNs typically used for RL from pixel inputs.

8.4. Contributions and Research Impact

This paper presents a twofold research contribution, both theoretically and empirically. On the theoretical side, our work proves that making constituent neural networks of MuZero equivariant leads to having a MuZero agent that is equivariant overall: that is, its acting and planning is performed in an equivariant manner. Empirically, we have verified that our method exhibits significant data efficiency for environments that are symmetric under the C_4 group. We have shown that an Equivariant MuZero agent achieves performance that is similar on the training data and its rotations, while non-equivariant agents tend to struggle on the rotated inputs. Further, Equivariant MuZero achieves a significant performance improvement on unseen environments: a gap that becomes more pronounced in lower-data environments. As this work has been recently made public, there have not been any follow-ups yet. However, I am looking forward to relaxing the constraints on the equivariance so that larger symmetry groups can be tackled and testing Equivariant MuZero on limited-data domains, such as robotics [187], drug discovery [188] and autonomous vehicles.

Chapter 9

Equivariant MuZero

Deep reinforcement learning repeatedly succeeds in closed, well-defined domains such as games (Chess, Go, StarCraft). The next frontier is real-world scenarios, where setups are numerous and varied. For this, agents need to learn the underlying rules governing the environment, so as to robustly generalise to conditions that differ from those they were trained on. Model-based reinforcement learning algorithms, such as the highly successful MuZero, aim to accomplish this by learning a world model. However, leveraging a world model has not consistently shown greater generalisation capabilities compared to model-free alternatives. In this work, we propose improving the data efficiency and generalisation capabilities of MuZero by explicitly incorporating the *symmetries* of the environment in its world-model architecture. We prove that, so long as the neural networks used by MuZero are equivariant to a particular symmetry group acting on the environment, the entirety of MuZero’s action-selection algorithm will also be equivariant to that group. We evaluate Equivariant MuZero on procedurally-generated MiniPacman and on Chaser from the ProcGen suite: training on a set of mazes, and then testing on unseen rotated versions, demonstrating the benefits of equivariance. Further, we verify that our performance improvements hold even when only some of the components of Equivariant MuZero obey strict equivariance, which highlights the robustness of our construction.

9.1. Introduction

Reinforcement learning (RL) is a potent paradigm for solving sequential decision making problems in a dynamically changing environment. Successful examples of its uses include game playing [189], drug design [49], robotics [190] and theoretical computer science [191]. However, the generality of RL often leads to data inefficiency, poor generalisation and lack of safety guarantees. This is an issue especially in domains where data is scarce or difficult to obtain, such as medicine or human-in-the-loop scenarios.

Most RL approaches do not directly attempt to capture the regularities present in the environment. As an example, consider a grid-world: moving down in a maze is equivalent to moving left in the 90° clock-wise rotation of the same maze. Such equivalences can be formalised via Markov Decision Process homomorphisms [192, 193], and while some works incorporate them [e.g. 194, 195], most deep reinforcement learning agents would act differently in such equivalent states if they do not observe enough data. This becomes even more problematic when the number of equivalent states is large. One common example is 3D regularities, such as changing camera angles in robotic tasks.

In recent years, there has been significant progress in building deep neural networks that explicitly obey such regularities, often termed geometric deep learning [18]. In this context, the regularities are formalised using symmetry groups and architectures are built by composing transformations that are equivariant to these symmetry groups (e.g. convolutional neural networks for the translation group, graph neural networks and transformers for the permutation group).

As we are looking to capture the symmetries present in an environment, a fitting place is within the framework of model-based RL (MBRL). MBRL leverages explicit world-models to forecast the effect of action sequences, either in the form of next-state or immediate reward predictions. These imagined trajectories are used to construct plans that optimise the forecasted returns. In the context of state-of-the-art MBRL agent MuZero [8], a Monte-Carlo tree search is executed over these world-models in order to perform action selection.

In this paper, we demonstrate that equivariance and MBRL can be effectively combined by proposing Equivariant MuZero (EqMuZero, shown in Figure 22), a variant of MuZero where equivariance constraints are enforced by design in its constituent neural networks. As MuZero does not use these networks directly to act, but rather executes a search algorithm on top of their predictions, it is not immediately obvious that the actions taken by the EqMuZero agent would obey the same constraints—is it guaranteed to produce a rotated action when given a rotated maze? One of our key contributions is a proof that guarantees this: as long as all neural networks are equivariant to a symmetry group, all actions taken will also be equivariant to that same symmetry group. Consequently, EqMuZero can be more data-efficient than standard MuZero, as it knows by construction how to act in states it has never seen before. We empirically verify the generalisation capabilities of EqMuZero in two grid-worlds: procedurally-generated MiniPacman and the Chaser game in the ProcGen suite.

9.2. Background

Reinforcement Learning. The reinforcement learning problem is typically formalised as a Markov Decision Process (S, A, P, R, γ) formed from a set of states S , a set of actions A ,

a discount factor $\gamma \in [0, 1]$, and two functions that model the outcome of taking action a in state s : the transition distribution $P(s'|s, a)$ —specifying the next state probabilities— and the reward function $R(s, a)$ —specifying the expected reward. The aim is to learn a *policy*, $\pi(a|s)$, a function specifying (probabilities of) actions to take in state s , such that the agent maximises the (expected) cumulative reward $G(\tau) = \sum_{t=0}^{t=T} \gamma^t R(s_t, a_t)$, where $\tau = (s_0, a_0, s_1, a_1, \dots, s_T, a_T)$ is the trajectory taken by the agent starting in the initial state s_0 and following the policy to decide a_t based on s_t .

MuZero. Reinforcement learning agents broadly fall into two categories: *model-free* and *model-based*. The specific agent we extend here, MuZero [8], is a model-based agent for deterministic environments (where $P(s'|s, a) = 1$ for exactly one s' for all $s \in S$ and $a \in A$). MuZero relies on several neural-network components that are composed to create a *world model*. These components are: the *encoder*, $E : S \rightarrow Z$, which embeds states into a latent space Z (e.g. $Z = \mathbb{R}^k$), the *transition model*, $T : Z \times A \rightarrow Z$, which predicts embeddings of next states, the *reward model*, $R : Z \times A \rightarrow \mathbb{R}$, which predicts the immediate expected reward after taking an action in a particular state, the *value model*, $V : Z \rightarrow \mathbb{R}$, which predicts the value (expected cumulative reward) from this state, and the *policy model* $P : Z \rightarrow [0, 1]^{|A|}$, which predicts the probability of taking each action from the current state. To plan its next action, MuZero executes a Monte Carlo tree search (MCTS) over many simulated trajectories, generated using the above models.

MuZero has demonstrated state-of-the-art capabilities over a variety of deterministic or near-deterministic environments, such as Go, Chess, Shogi and Atari, and has been successfully applied to real-world domains such as video compression [196]. Although here we focus on MuZero for deterministic environments, we note that extensions to stochastic environments also exist [197] and are an interesting target for future work.

Groups and Representations. A *group* (\mathfrak{G}, \circ) is a set \mathfrak{G} equipped with a *composition* operation $\circ : \mathfrak{G} \times \mathfrak{G} \rightarrow \mathfrak{G}$ (written concisely as $\mathfrak{g} \circ \mathfrak{h} = \mathfrak{gh}$), satisfying the following axioms: (*associativity*) $(\mathfrak{gh})\mathfrak{l} = \mathfrak{g}(\mathfrak{hl})$ for all $\mathfrak{g}, \mathfrak{h}, \mathfrak{l} \in \mathfrak{G}$; (*identity*) there exists a unique $\mathfrak{e} \in \mathfrak{G}$ satisfying $\mathfrak{eg} = \mathfrak{ge} = \mathfrak{g}$ for all $\mathfrak{g} \in \mathfrak{G}$; (*inverse*) for every $\mathfrak{g} \in \mathfrak{G}$ there exists a unique $\mathfrak{g}^{-1} \in \mathfrak{G}$ such that $\mathfrak{gg}^{-1} = \mathfrak{g}^{-1}\mathfrak{g} = \mathfrak{e}$.

Groups are a natural way to describe *symmetries*: object transformations that leave them unchanged. They can be reasoned about in the context of linear algebra by using their *real representations*: functions $\rho_{\mathcal{V}} : \mathfrak{G} \rightarrow \mathbb{R}^{N \times N}$ that give, for every group element $\mathfrak{g} \in \mathfrak{G}$, a real matrix demonstrating how this element *acts* on a vector space \mathcal{V} . For example, for the rotation group $\mathfrak{G} = \text{SO}(n)$, the representation $\rho_{\mathcal{V}}$ would provide an appropriate $n \times n$ rotation matrix for each rotation \mathfrak{g} .

Equivariance and Invariance. As symmetries are assumed to not change the essence of the data they act on, we would like to construct neural networks that adequately represent such

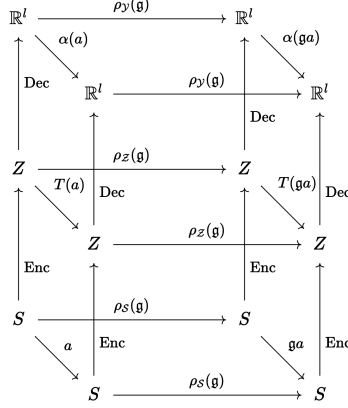


Figure 21 – Commutative diagram of symmetries in RL. State transitions due to an action a are back-to-front, transformations due to a symmetry \mathfrak{g} are left-to-right, state encoding and decoding by the model is bottom-to-top.

symmetry-transformed inputs. Assume we have a neural network $f : \mathcal{X} \rightarrow \mathcal{Y}$, mapping between vector spaces \mathcal{X} and \mathcal{Y} , and that we would like this network to respect the symmetries within a group \mathfrak{G} . Then we can impose the following condition, for all group elements $\mathfrak{g} \in \mathfrak{G}$ and inputs $\mathbf{x} \in \mathcal{X}$:

$$f(\rho_{\mathcal{X}}(\mathfrak{g})\mathbf{x}) = \rho_{\mathcal{Y}}(\mathfrak{g})f(\mathbf{x}). \quad (9.2.1)$$

This condition is known as \mathfrak{G} -equivariance—for any group element, it does not matter whether we act with it on the input or on the output of the function f —the end result is the same. A special case of this, \mathfrak{G} -invariance, is when the output representation is trivial ($\rho_{\mathcal{Y}}(\mathfrak{g}) = \mathbf{I}$):

$$f(\rho_{\mathcal{X}}(\mathfrak{g})\mathbf{x}) = f(\mathbf{x}). \quad (9.2.2)$$

In geometric deep learning, equivariance to reflections, rotations, translations and permutations has been of particular interest [18].

Generally speaking, there are three ways to obtain an equivariant model: a) data augmentation, b) data canonicalisation and c) specialised architectures. Data augmentation creates additional training data by applying group elements \mathfrak{g} to input/output pairs (\mathbf{x}, \mathbf{y}) —equivariance is encouraged by training on the transformed data and/or minimising auxiliary losses such as $\|\rho_{\mathcal{Y}}(\mathfrak{g})f(\mathbf{x}) - f(\rho_{\mathcal{X}}(\mathfrak{g})\mathbf{x})\|$. Data augmentation can be simple to apply, but it results in only approximate equivariance. Data canonicalisation requires a method to standardise the input, such as breaking the translation symmetry for molecular representation by centering the atoms around the origin [198]—however, in many cases, such as the relatively simple MiniPacman environment we use in our experiments, such a canonical transformation may not exist. Specialised architectures have the downside of being harder to build, but they can guarantee exact equivariance—as such, they reduce the search space of functions, potentially reducing the number of parameters and increasing training efficiency.

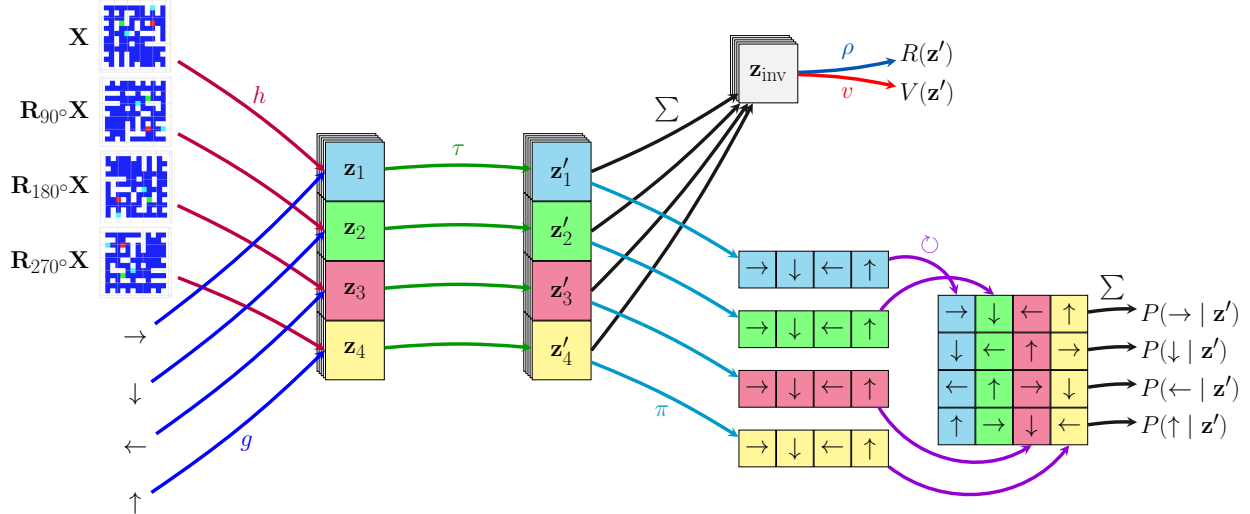


Figure 22 – Architecture of Equivariant MuZero, where h , g are encoders, τ is the transition model, ρ is the reward model, v is the value model and π is the policy predictor. Each colour represents an element of the C_4 group $\{\mathbf{I}, \mathbf{R}_{90^\circ}, \mathbf{R}_{180^\circ}, \mathbf{R}_{270^\circ}\}$ applied to the input (observation and action).

Equivariance in RL. There has been previous work at the intersection of reinforcement learning and equivariance. While leveraging multi-agent symmetries was repeatedly shown to hold promise [199, 200], of particular interest to us are the symmetries emerging from the environment, in a single-agent scenario. Related work in this space can be summarised by the commutative diagram in Figure 21. When considering only the cube at the bottom, we recover [186]—a supervised learning task where a latent transition model T learns to predict the next state embedding. They show that if T is equivariant, the encoder can pick up the symmetries of the environment even if it is not fully equivariant by design. Mondal et al. [201] build a model-free agent by combining an equivariant-by-design encoder and enforcing the remaining equivariances via regularisation losses. They also consider the invariance of the reward, captured in Figure 21 by taking the decoder to be the reward model and $l = 1$. The work of [194] can be described by having the value model as the decoder, while the work of [202] has the decoder as the policy model and $l = |A|$.

9.3. Equivariant MuZero

In what follows, we describe how the various components of EqMuZero (Figure 22) are designed to obey C_4 -equivariance. For simplicity, we assume there are only four directional movement actions in the environment ($A = \{\rightarrow, \downarrow, \leftarrow, \uparrow\}$). Any additional non-movement actions (such as the “do nothing” action) can be included without difficulty.

To enforce C_4 -equivariance in the encoder, we first need to specify the effect of rotations on the latent state \mathbf{z} . In our implementation, the latent state consists of 4 equally shaped

arrays, $\mathbf{z} = (\mathbf{z}_1, \mathbf{z}_2, \mathbf{z}_3, \mathbf{z}_4)$, and we prescribe that a 90° clock-wise rotation manifests as a cyclical permutation: $\mathbf{R}_{90^\circ}\mathbf{z} = (\mathbf{z}_2, \mathbf{z}_3, \mathbf{z}_4, \mathbf{z}_1)$. Then, our equivariant encoder embeds state \mathbf{X} and action a as follows:

$$E(\mathbf{X}, a) = (h(\mathbf{X}) + g(a), h(\mathbf{R}_{90^\circ}\mathbf{X}) + g(\mathbf{R}_{90^\circ}a), h(\mathbf{R}_{180^\circ}\mathbf{X}) + g(\mathbf{R}_{180^\circ}a), h(\mathbf{R}_{270^\circ}\mathbf{X}) + g(\mathbf{R}_{270^\circ}a)) \quad (9.3.1)$$

where h is a CNN and g is an MLP. The output of g is accordingly broadcasted across all pixels of h 's output. This equation satisfies C_4 -equivariance, that is, $E(\mathbf{R}_{90^\circ}\mathbf{X}, \mathbf{R}_{90^\circ}a) = \mathbf{R}_{90^\circ}E(\mathbf{X}, a)$.

We can build a C_4 -equivariant transition model by maintaining the structure in the latent space:

$$T(\mathbf{z}) = (\tau(\mathbf{z}_1), \tau(\mathbf{z}_2), \tau(\mathbf{z}_3), \tau(\mathbf{z}_4)). \quad (9.3.2)$$

A less constrained T would allow components of \mathbf{z} to *interact*, while still retaining C_4 -equivariance:

$$T(\mathbf{z}) = (\tau(\mathbf{z}_1, \mathbf{z}_2, \mathbf{z}_3, \mathbf{z}_4), \tau(\mathbf{z}_2, \mathbf{z}_3, \mathbf{z}_4, \mathbf{z}_1), \tau(\mathbf{z}_3, \mathbf{z}_4, \mathbf{z}_1, \mathbf{z}_2), \tau(\mathbf{z}_4, \mathbf{z}_1, \mathbf{z}_2, \mathbf{z}_3)). \quad (9.3.3)$$

In our experiments, we use the more constrained variant for MiniPacman, and the less constrained variant for Chaser, as more data is available for the latter. In either case, we take τ to be a ResNet.

The policy is made C_4 -equivariant by combining state and action embeddings from all four latents:

$$P(a | \mathbf{z}) = \frac{\pi(a | \mathbf{z}_1) + \pi(\mathbf{R}_{90^\circ}a | \mathbf{z}_2) + \pi(\mathbf{R}_{180^\circ}a | \mathbf{z}_3) + \pi(\mathbf{R}_{270^\circ}a | \mathbf{z}_4)}{4} \quad (9.3.4)$$

where $\pi(\cdot | \mathbf{z}_i)$ is an MLP followed by a softmax, which produces a probability distribution over actions given the map encoded by \mathbf{z}_i . It is easy to show that $\sum_{a \in \mathcal{A}} P(a | \mathbf{z}) = 1$, i.e. $P(\cdot | \mathbf{z})$ is properly normalised, and that $P(\mathbf{R}_{90^\circ}a | \mathbf{R}_{90^\circ}\mathbf{z}) = P(a | \mathbf{z})$, i.e. it satisfies C_4 -equivariance.

Lastly, the reward and value networks (R, V), modeled by MLPs ρ and v respectively, should be C_4 -invariant. We can satisfy this constraint by *aggregating* the latent space with any C_4 -invariant function, such as sum, average or max. Here we use summation:

$$R(\mathbf{z}) = \rho(\mathbf{z}_1 + \mathbf{z}_2 + \mathbf{z}_3 + \mathbf{z}_4), \quad V(\mathbf{z}) = v(\mathbf{z}_1 + \mathbf{z}_2 + \mathbf{z}_3 + \mathbf{z}_4). \quad (9.3.5)$$

Composing the equivariant components described above (Equations 9.3.1–9.3.5), we construct the end-to-end equivariant EqMuZero agent, displayed in Figure 22. Indeed, we can show that EqMuZero will provably behave in an equivariant manner when selecting actions:

Theorem 1 *If all the relevant neural networks used by MuZero are \mathfrak{G} -equivariant, the proposed EqMuZero agent will select actions in a \mathfrak{G} -equivariant manner, that is for every*

state $s \in S$ and for every $\mathbf{g} \in \mathfrak{G}$, if *EqMuZero* selects action a while in s , then it must select $\mathbf{g}a$ while in $\mathbf{g}s$.

Proof. Assume our neural networks are: h for the encoder, τ for the transition model, π for the policy model, v for the value model and ρ for the reward model. By design, we make h, τ and π be \mathfrak{G} -equivariant, and v and ρ be \mathfrak{G} -invariant.

The reward, value, policy and transition respect the equivariances, as compositions of equivariant functions:

$$\begin{aligned} R &= \rho\tau^k h \\ V &= v\tau^k h \\ P &= \pi\tau^k h \\ T &= \tau^k h. \end{aligned} \tag{9.3.6}$$

Then, the return is also a \mathfrak{G} -invariant function as it is the sum of two \mathfrak{G} -invariant functions:

$$G(s^k) = \sum_{\tau=0}^{l-1-k} \gamma^\tau \rho(s^{k+\tau}, a^{k+1+\tau}) + \gamma^{l-k} v(s^l, a^{l+1}). \tag{9.3.7}$$

For proving that one planning step is equivariant, we need to show that the action selection is \mathfrak{G} -equivariant.

Since the outcome of MuZero’s MCTS function is based on the initial observation, o , we denote MCTS’s internal state as $\{Q^o(s, a), N^o(s, a), \dots\}$. We use identical notation as Schrittwieser et al. [8] for these states, even though we express the MuZero models R, V, P, T somewhat differently.

Knowing how they are updated:

$$a^k = \operatorname{argmax}_a \left[Q^o(s^{k-1}, a) + P^o(s^{k-1}, a) \frac{\sqrt{\sum_b N^o(s^{k-1}, b)}}{1 + N^o(s^{k-1}, a)} \left(c_1 + \log \left(\frac{\sum_b N^o(s^{k-1}, b) + c_2 + 1}{c_2} \right) \right) \right] \tag{9.3.8}$$

$$Q_t^o(s^{k-1}, a^k) = \frac{N_{t-1}^o(s^{k-1}, a^k) Q_{t-1}^o(s^{k-1}, a^k) + G(s^{k-1})}{N_{t-1}^o(s^{k-1}, a^k) + 1} \tag{9.3.9}$$

$$N_t^o(s^{k-1}, a^k) = N_{t-1}^o(s^{k-1}, a^k) + 1.$$

As discussed previously, we need to show that, for each MCTS internal state (e.g. N^o), if we assume π, v, τ, ρ, h to be equivariant functions, the resulting state would also be equivariant under transformations of the initial observation. That is, for all s, a :

$$N^{\mathbf{g}o}(s, a) = N^o(s, a). \tag{9.3.10}$$

To prove this, we will use induction on the number of backups performed by MCTS, t . We proceed:

$$\begin{aligned} \text{Base case } (t = 0) : N_0^{\mathbf{g}o}(s, a) &= N_0^o(s, a) = 0 \\ Q_0^{\mathbf{g}o}(s, a) &= Q_0^o(s, a) = 0. \end{aligned} \tag{9.3.11}$$

Assume:

$$\begin{aligned} \text{Case } t : N_t^{\mathfrak{g}_o o}(\mathfrak{g}_s s, \mathfrak{g}_a a) &= N_t^o(s, a) \\ Q_t^{\mathfrak{g}_o o}(\mathfrak{g}_s s, \mathfrak{g}_a a) &= Q_t^o(s, a). \end{aligned} \quad (9.3.12)$$

We will start by showing that the states and actions expanded by MCTS under initial \mathfrak{G} -transformed observation $\mathfrak{g}_o o$ ($\tilde{s}^0, \tilde{a}^1, \tilde{s}^1, \tilde{a}^2, \dots$), would exactly correspond to $(\mathfrak{g}_s s^0, \mathfrak{g}_a a^1, \mathfrak{g}_s s^1, \mathfrak{g}_a a^2, \dots)$, where $(s^0, a^1, s^1, a^2, \dots)$ are states expanded under the non-transformed observation, o .

By equivariance of h , $\tilde{s}^0 = h(\mathfrak{g}_o o) = \mathfrak{g}_s h(o) = \mathfrak{g}_s s^0$, as expected.

Next, we show that the actions selected by MCTS also obey a \mathfrak{G} -equivariance constraint, in the sense that: if $\tilde{s}^{k-1} = \mathfrak{g}_s s^{k-1}$, then $\tilde{a}^k = \mathfrak{g}_a a^k$.

As we assumed N_t^o to be \mathfrak{G} -equivariant, it must hold that $\sum_b N_t^o(s, b)$ is \mathfrak{G} -invariant (as a sum-reduction of equivariant functions). Hence, we can rewrite Equation 9.3.8 as:

$$a^k = \arg \max_a \left[Q_t^o(s^{k-1}, a) + P_t^o(s^{k-1}, a) \frac{\epsilon(s^{k-1})}{1 + N_t^o(s^{k-1}, a)} \right] \quad (9.3.13)$$

where ϵ is \mathfrak{G} -invariant, P^o is \mathfrak{G} -equivariant by composition of functions that are \mathfrak{G} -equivariant by assumption, and Q^o is \mathfrak{G} -equivariant by assumption of Case t .

Hence, using this formula to define \tilde{a}^k , we recover:

$$\begin{aligned} \tilde{a}^k &= \arg \max_a \left[Q_t^{\mathfrak{g}_o o}(\tilde{s}^{k-1}, a) + P_t^{\mathfrak{g}_o o}(\tilde{s}^{k-1}, a) \frac{\epsilon(\tilde{s}^{k-1})}{1 + N_t^{\mathfrak{g}_o o}(\tilde{s}^{k-1}, a)} \right] \\ &= \arg \max_a \left[Q_t^{\mathfrak{g}_o o}(\mathfrak{g}_s s^{k-1}, a) + P_t^{\mathfrak{g}_o o}(\mathfrak{g}_s s^{k-1}, a) \frac{\epsilon(\mathfrak{g}_s s^{k-1})}{1 + N_t^{\mathfrak{g}_o o}(\mathfrak{g}_s s^{k-1}, a)} \right] \\ &= \arg \max_a \left[Q_t^{\mathfrak{g}_o o}(\mathfrak{g}_s s^{k-1}, \mathfrak{g}_a \mathfrak{g}_a^{-1} a) + P_t^{\mathfrak{g}_o o}(\mathfrak{g}_s s^{k-1}, \mathfrak{g}_a \mathfrak{g}_a^{-1} a) \frac{\epsilon(\mathfrak{g}_s s^{k-1})}{1 + N_t^{\mathfrak{g}_o o}(\mathfrak{g}_s s^{k-1}, \mathfrak{g}_a \mathfrak{g}_a^{-1} a)} \right] \\ &= \arg \max_a \left[Q_t^o(s^{k-1}, \mathfrak{g}_a^{-1} a) + P_t^o(s^{k-1}, \mathfrak{g}_a^{-1} a) \frac{\epsilon(s^{k-1})}{1 + N_t^o(s^{k-1}, \mathfrak{g}_a^{-1} a)} \right] \\ &= \mathfrak{g}_a \arg \max_a \left[Q_t^o(s^{k-1}, a) + P_t^o(s^{k-1}, a) \frac{\epsilon(s^{k-1})}{1 + N_t^o(s^{k-1}, a)} \right] \\ &= \mathfrak{g}_a a^k. \end{aligned}$$

Note that we have taken the \mathfrak{g}_a out of the $\arg \max$, which is an unambiguous operation only if there is a unique action a^k that maximises the expression in Equation 9.3.13. To avoid breaking the symmetry in practice, we propose that tiebreaks for a^k are resolved in a purely randomised fashion.

Showing this, we now only need to verify that the updates to N_t and Q_t (in Equation 9.3.9) are equivariant for all state-action pairs along the trajectory. Values of N and Q for all other state-action pairs will be unchanged from N_t , and therefore trivially still \mathfrak{G} -equivariant.

First we show this for N :

$$\begin{aligned}
N_{t+1}^{\mathfrak{g}oo}(\tilde{s}^{k-1}, \tilde{a}^k) &= N_{t+1}^{\mathfrak{g}oo}(\mathfrak{g}_s s^{k-1}, \mathfrak{g}_a a^k) \\
&= N_t^{\mathfrak{g}oo}(\mathfrak{g}_s s^{k-1}, \mathfrak{g}_a a^k) + 1 \\
&= N_t^o(s^{k-1}, a^k) + 1 \\
&= N_{t+1}^o(s^{k-1}, a^k).
\end{aligned}$$

Hence, Case $t + 1$ still holds for N . Now we turn our attention to Q .

First, by invariance of ρ and w , we can show that $G(s^k)$ is a sum of \mathfrak{G} -invariant functions and therefore also invariant. Plugging into the Q update:

$$\begin{aligned}
Q_{t+1}^{\mathfrak{g}oo}(\tilde{s}^{k-1}, \tilde{a}^k) &= Q_{t+1}^{\mathfrak{g}oo}(\mathfrak{g}_s s^{k-1}, \mathfrak{g}_a a^k) \\
&= \frac{N_t^{\mathfrak{g}oo}(\mathfrak{g}_s s^{k-1}, \mathfrak{g}_a a^k) Q_t^{\mathfrak{g}oo}(\mathfrak{g}_s s^{k-1}, \mathfrak{g}_a a^k) + G(\mathfrak{g}_s s^{k-1})}{N_t^{\mathfrak{g}oo}(\mathfrak{g}_s s^{k-1}, \mathfrak{g}_a a^k) + 1} \\
&= \frac{N_t^o(s^{k-1}, a^k) Q_t^o(s^{k-1}, a^k) + G(s^{k-1})}{N_t^o(s^{k-1}, a^k) + 1} \\
&= Q_{t+1}^o(s^{k-1}, a^k).
\end{aligned}$$

Hence, Case $t + 1$ also holds for Q . As discussed before, we assume it holds by composition for all other state stored by MCTS (P, T, R). \square

Having proved that all internal state of of MCTS consistently remains transformed by \mathfrak{G} under transformed input observations, we can conclude that the final policy given by MCTS will be exactly \mathfrak{G} -equivariant.

9.4. Experiments and results

Environments. We consider two 2D grid-world environments, MiniPacman [203] and Chaser [7], that feature an agent navigating in a 2D maze. In both environments, the state is the grid-world map \mathbf{X} and an action is a direction to move. Both of these grid-worlds are symmetric with respect to 90° rotations, in the sense that moving down in some map is the same as moving left in the 90° clock-wise rotated version of the same map. Hence, we take our symmetry group to be $\mathfrak{G} = C_4 = \{\mathbf{I}, \mathbf{R}_{90^\circ}, \mathbf{R}_{180^\circ}, \mathbf{R}_{270^\circ}\}$, the 4-element cyclic group, which in our case represents rotating the map by all four possible multiples of 90° .

Results. We compare EqMuZero with a standard MuZero that uses non-equivariant components: ResNet-style networks for the encoder and transition models, and MLP-based policy, value and reward models, following [21]. As the encoder and the policy of EqMuZero are the only two components which require knowledge of how the symmetry group acts on the environment, we include the following ablations in order to evaluate the trade-off between

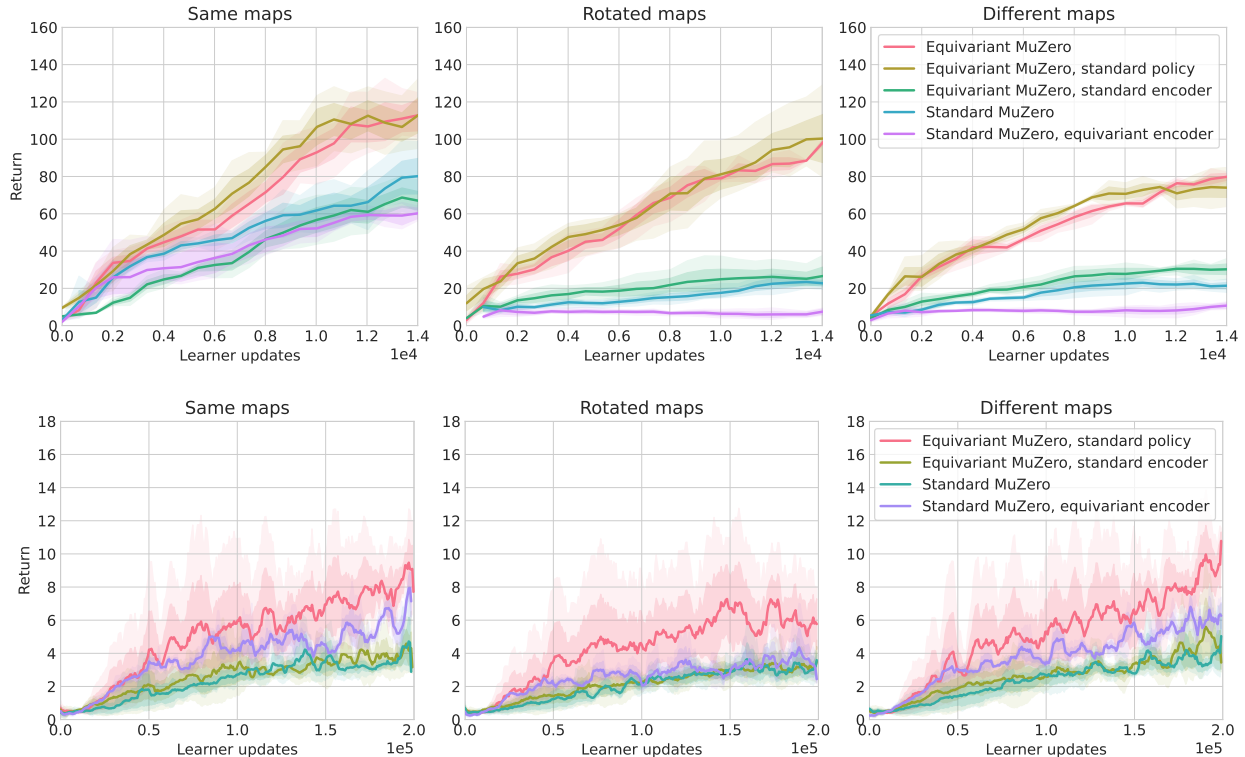


Figure 23 – Results on procedurally-generated MiniPacman (top) and Chaser from ProcGen (bottom).

end-to-end equivariance and general applicability: Standard MuZero with an equivariant encoder, equivariant MuZero with a standard encoder and equivariant MuZero with a standard policy model.

We train each agent on a set of maps, \mathbf{X} . To test for generalisation, we measure the agent’s performance on three, progressively harder, settings. Namely, we evaluate the agent on \mathbf{X} , with randomised initial agent position (denoted by *same* in our results), on the set of rotated maps $\mathbf{R}\mathbf{X}$, where $\mathbf{R} \in \{\mathbf{R}_{90^\circ}, \mathbf{R}_{180^\circ}, \mathbf{R}_{270^\circ}\}$ (denoted by *rotated*) and, lastly, on a set of maps \mathbf{Y} , such that $\mathbf{Y} \cap \mathbf{X} = \emptyset$ and $\mathbf{Y} \cap \mathbf{R}\mathbf{X} = \emptyset$ (denoted by *different*).

Figure 23 (top) presents the results of the agents on MiniPacman. First, we empirically confirm that the average reward on layouts \mathbf{X} , seen during training, matches the average reward gathered on the rotations of the same mazes, $\mathbf{R}\mathbf{X}$, for EqMuZero. Second, we notice that changing the equivariant policy with a non-equivariant one does not significantly impact performance. However, the same swap in the encoder brings the performance of the agent down to that of Standard MuZero—this suggests that the structure in the latent space of the transition model, when not combined with some explicit method of imposing equivariance in the encoder, does not provide noticeable benefits. Third, we notice that Equivariant MuZero is generally robust to layout variations, as the learnt high-reward behaviours also transfer to \mathbf{Y} . At the same time, Standard MuZero significantly drops in performance for both \mathbf{Y} and

RX. We note that experiments on MiniPacman were done in a low-data scenario, using 5 maps of size 14×14 for training; we observed that the differences between agents diminished when all agents were trained with at least 20 times more maps.

Figure 23 (bottom) compares the performance of the agents on the ProcGen game, Chaser, which has similar dynamics to MiniPacman, but larger mazes of size 64×64 and a more complex action space. Due to the complexity of the action space, we only use Eq-MuZero with a standard policy, rather than a fully equivariant version. We use 500 maze instances for training. Our results demonstrate that, even when the problem complexity is increased in such a way, Equivariant MuZero still consistently outperforms the other agents, leading to more robust plans being discovered.

9.5. Limitations and future work

While the theory of Equivariant MuZero generalizes to any symmetry group, in this work we test an instance where the component neural networks satisfy the criteria for the C_4 group. Scaling it up to continuous rotations, such as the $SO(3)$ group, would make the architecture applicable to different problems, such as molecular tasks. However, for parts such as the encoder, the transition model and the policy, a different strategy would be required, as it is impossible to transform the input observation and action with every element of the group. Future work could consider enforcing the equivariance constraints via additional losses, possibly combining with an approach such as in Park et al. [186], keeping in mind that the theoretical guarantees will no apply as they are in their current form anymore. More generally, this work can also be composed with a module that discovers symmetries, such as in [204].

9.6. Conclusions

We present Equivariant MuZero, a model-based agent that is, by construction, equivariant. We theoretically verify its properties with respect to general symmetry groups, proving the agent’s overall equivariance given the appropriate conditions are met by its constituent neural networks. Moreover, we empirically demonstrate that an Equivariant MuZero agent that is C_4 -equivariant generalizes to unseen rotations of the training data, as well as more robustly performing on test mazes, with diminishing returns when presented with $100\times$ more data.

Chapter 10

General Conclusions

I started this PhD at a time where graph neural networks were a niche topic and I am finishing now, when it's constantly been the second most popular area at Machine Learning Conferences, after Reinforcement Learning. In the process, I saw how GNNs were applied to tasks where machine learning was already used, with data where the structure was present, but not being leveraged, as well as completely novel applications that were made possible because of GNNs. Benchmarks were proposed and explored to the point of exposing GNN limitations that we didn't know about at the beginning, and also uncovering GNN strengths that were also waiting to be found. All in all, impressive results were obtained thanks to this new manner of learning representations and likely more are to be found.

In a sense, my thesis reflects this journey – from applying GNNs to drug-like molecules, which were previously modelled as strings, to tackling the problem of antibody-antigen interaction, only lightly explored in ML before, to studying GNNs' limitations on data with long-range interactions or a high degree of heterophily, to leveraging GNNs' algorithmic alignment with planning algorithms and using them as neural algorithmic reasoners in reinforcement learning, and, lastly, extending beyond GNNs in RL, to rotation-equivariance in model-based agents.

In most early explorations of structure within deep learning architectures, including many works that have been published in machine learning venues during the development of this thesis, the provided structure was used “as-is”, often without any questioning about its utility or origin. One underlying theme for the presented chapters is that, in order to squeeze the most potential out of geometric deep learning, it might be in fact necessary to *modulate* this input structure. For the first two core chapters, it is beneficial to obtain the computation graph not directly from the input graph, as it is standard, but from an evolved version of it, for example using “template graphs” with excellent communication properties (low diameter, low commute times, etc.), or using weak classifiers to connect nodes that are feature-wise or structurally similar. For the last two core chapters, modulation is not just

advantageous, but necessary – deploying neural algorithmic reasoners requires the formation of a graph-structured input in latent space, similarly to how applying equivariant neural networks requires command of the symmetry group’s elements. In both implicit planning and model-based reinforcement learning, such structures are not readily given in the input.

While the proposed approaches provided ways to improve GNNs’ applicability, they also emphasise the need for more scalable and robust solutions. Going forward, I think the heuristics of how to modify the computation graph should be replaced by automatic search of the true connectivity underlying the graph generation process, guided by the downstream task—this is, arguably, part of the reason behind Transformers’ success—starting from the given input graph and taking into account local and global properties of information propagation. When extended to geometric deep learning, this can provide the solution to many problems where there are symmetries in the input, such as tasks across many areas of natural sciences.

The other utility of graph-based methods that I found exciting along my PhD was using them to store, process and leverage information about the world. One instance of this is a graph of states visited by an agent when acting in an environment, and the other a graph of knowledge about the world, on which an agent can reason. The latter constitutes an emerging area of interest, mainly through the huge progress in large language models – a recent talk by John Schulman remarked that LLMs implicitly learn a knowledge graph.

While, in my view, these developments successfully move the “starting point” from specialist architectures that need to (re)learn using backpropagation to generalists where information is readily available from a foundation model, it still leaves fundamental questions unanswered. Many of these questions might have already been tackled by previous GNN and RL research, and they can often be related to problems of long-range interactions, heterophilic and path-finding problems on a graph. Just to name a few examples, such questions may include: how to adapt to new knowledge and situations in scalable ways, how to both explore new skills and discover skills that are latent in the training data, how to optimize not just one-step answers, but answers that truly consider the time axis, how to leverage different types of data, from multi-modality to varying levels of expertise and, generally, how to reason robustly and extrapolate beyond previously seen data.

With all these parallels in mind, I am hopeful that my PhD thesis research can form an exciting stepping-stone towards methods that can ameliorate or even resolve such fundamental problems. I look forward to exploring these directions in future work!

References

- [1] Keyulu Xu, Weihua Hu, Jure Leskovec, and Stefanie Jegelka. How powerful are graph neural networks? *arXiv preprint arXiv:1810.00826*, 2018.
- [2] Weihua Hu, Matthias Fey, Marinka Zitnik, Yuxiao Dong, Hongyu Ren, Bowen Liu, Michele Catasta, and Jure Leskovec. Open graph benchmark: Datasets for machine learning on graphs. *arXiv preprint arXiv:2005.00687*, 2020.
- [3] Oleg Platonov, Denis Kuznedelev, Michael Diskin, Artem Babenko, and Liudmila Prokhorenkova. A critical look at the evaluation of gnns under heterophily: are we really making progress? *arXiv preprint arXiv:2302.11640*, 2023.
- [4] Uri Alon and Eran Yahav. On the bottleneck of graph neural networks and its practical implications. *arXiv preprint arXiv:2006.05205*, 2020.
- [5] Elise van der Pol, Thomas Kipf, Frans A. Oliehoek, and Max Welling. Plannable approximations to mdp homomorphisms: Equivariance under actions. In *AAMAS 2020*, 2020.
- [6] Keyulu Xu, Jingling Li, Mozhi Zhang, Simon S. Du, Ken-ichi Kawarabayashi, and Stefanie Jegelka. What can neural networks reason about? In *8th International Conference on Learning Representations*, 2020. URL <https://openreview.net/forum?id=rJxbJeHFPS>.
- [7] Karl Cobbe, Chris Hesse, Jacob Hilton, and John Schulman. Leveraging procedural generation to benchmark reinforcement learning. In *International Conference on Machine Learning*, pages 2048–2056. PMLR, 2020.
- [8] Julian Schrittwieser, Ioannis Antonoglou, Thomas Hubert, Karen Simonyan, Laurent Sifre, Simon Schmitt, Arthur Guez, Edward Lockhart, Demis Hassabis, Thore Graepel, et al. Mastering atari, go, chess and shogi by planning with a learned model. *Nature*, 588(7839):604–609, 2020.
- [9] Yu He, Petar Veličković, Pietro Liò, and Andreea Deac. Continuous neural algorithmic planners. In *Learning on Graphs Conference*, pages 54–1. PMLR, 2022.
- [10] Yann LeCun, Yoshua Bengio, et al. Convolutional networks for images, speech, and time series. *The handbook of brain theory and neural networks*, 3361(10):1995, 1995.

- [11] Corentin Tallec and Yann Ollivier. Can recurrent neural networks warp time? *arXiv preprint arXiv:1804.11188*, 2018.
- [12] Chaitanya Joshi. Transformers are graph neural networks. *The Gradient*, 2020.
- [13] 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.
- [14] Thomas N Kipf and Max Welling. Semi-supervised classification with graph convolutional networks. *arXiv preprint arXiv:1609.02907*, 2016.
- [15] Petar Veličković, Guillem Cucurull, Arantxa Casanova, Adriana Romero, Pietro Lio, and Yoshua Bengio. Graph attention networks. *arXiv preprint arXiv:1710.10903*, 2017.
- [16] Jessica B Hamrick, Kelsey R Allen, Victor Bapst, Tina Zhu, Kevin R McKee, Joshua B Tenenbaum, and Peter W Battaglia. Relational inductive bias for physical construction in humans and machines. *arXiv preprint arXiv:1806.01203*, 2018.
- [17] Petar Veličković. Message passing all the way up. *arXiv preprint arXiv:2202.11097*, 2022.
- [18] Michael M Bronstein, Joan Bruna, Taco Cohen, and Petar Veličković. Geometric deep learning: Grids, groups, graphs, geodesics, and gauges. *arXiv preprint arXiv:2104.13478*, 2021.
- [19] Petar Veličković and Charles Blundell. Neural algorithmic reasoning. *arXiv preprint arXiv:2105.02761*, 2021.
- [20] Richard S Sutton and Andrew G Barto. *Reinforcement learning: An introduction*. MIT press, 2018.
- [21] Jessica B Hamrick, Abram L Friesen, Feryal Behbahani, Arthur Guez, Fabio Viola, Sims Witherspoon, Thomas Anthony, Lars Buesing, Petar Veličković, and Théophane Weber. On the role of planning in model-based deep reinforcement learning. *arXiv preprint arXiv:2011.04021*, 2020.
- [22] Andreea Deac, Marc Lackenby, and Petar Veličković. Expander graph propagation. In *Learning on Graphs Conference*, pages 38–1. PMLR, 2022.
- [23] Andreea Deac and Jian Tang. Evolving computation graphs, 2023.
- [24] Andreea Deac, Petar Velickovic, Ognjen Milinkovic, Pierre-Luc Bacon, Jian Tang, and Mladen Nikolic. Neural algorithmic reasoners are implicit planners. In *Advances in Neural Information Processing Systems 34*, pages 15529–15542, 2021. URL <https://proceedings.neurips.cc/paper/2021/hash/82e9e7a12665240d13d0b928be28f230-Abstract.html>.
- [25] Andreea Deac, Théophane Weber, and George Papamakarios. Equivariant muzero, 2023.
- [26] Michaël Defferrard, Xavier Bresson, and Pierre Vandergheynst. Convolutional neural networks on graphs with fast localized spectral filtering. In *Advances in neural information processing systems*, pages 3844–3852, 2016.

- [27] Xavier Bresson and Thomas Laurent. Residual gated graph convnets, 2018.
- [28] Vijay Prakash Dwivedi, Chaitanya K Joshi, Thomas Laurent, Yoshua Bengio, and Xavier Bresson. Benchmarking graph neural networks. *arXiv preprint arXiv:2003.00982*, 2020.
- [29] Vijay Prakash Dwivedi, Anh Tuan Luu, Thomas Laurent, Yoshua Bengio, and Xavier Bresson. Graph neural networks with learnable structural and positional representations. *arXiv preprint arXiv:2110.07875*, 2021.
- [30] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition, 2015.
- [31] Boris Weisfeiler and Andrei Leman. The reduction of a graph to canonical form and the algebra which appears therein. *nti, Series*, 2(9):12–16, 1968.
- [32] Qimai Li, Zhichao Han, and Xiao-Ming Wu. Deeper insights into graph convolutional networks for semi-supervised learning. In *Proceedings of the AAAI conference on artificial intelligence*, volume 32, 2018.
- [33] Pablo Barceló, Egor V Kostylev, Mikael Monet, Jorge Pérez, Juan Reutter, and Juan Pablo Silva. The logical expressiveness of graph neural networks. In *8th International Conference on Learning Representations (ICLR 2020)*, 2020.
- [34] Kenta Oono and Taiji Suzuki. Graph neural networks exponentially lose expressive power for node classification. *arXiv preprint arXiv:1905.10947*, 2019.
- [35] Jiong Zhu, Yujun Yan, Lingxiao Zhao, Mark Heimann, Leman Akoglu, and Danai Koutra. Beyond homophily in graph neural networks: Current limitations and effective designs. *Advances in Neural Information Processing Systems*, 33, 2020.
- [36] Andrew J Dudzik and Petar Veličković. Graph neural networks are dynamic programmers. *Advances in Neural Information Processing Systems*, 35:20635–20647, 2022.
- [37] Manzil Zaheer, Satwik Kottur, Siamak Ravanbakhsh, Barnabas Poczos, Russ R Salakhutdinov, and Alexander J Smola. Deep sets. *Advances in neural information processing systems*, 30, 2017.
- [38] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. In I. Guyon, U. V. Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett, editors, *Advances in Neural Information Processing Systems*, volume 30. Curran Associates, Inc., 2017. URL <https://proceedings.neurips.cc/paper/2017/file/3f5ee243547dee91fbd053c1c4a845aa-Paper.pdf>.
- [39] Vijay Prakash Dwivedi and Xavier Bresson. A generalization of transformer networks to graphs. *arXiv preprint arXiv:2012.09699*, 2020.
- [40] Richard Bellman. Dynamic programming. *Science*, 153(3731):34–37, 1966.
- [41] Martin L Puterman. *Markov decision processes: discrete stochastic dynamic programming*. John Wiley & Sons, 2014.

- [42] 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.
- [43] Ronald J Williams. Simple statistical gradient-following algorithms for connectionist reinforcement learning. *Machine learning*, 8(3-4):229–256, 1992.
- [44] Volodymyr Mnih, Adria Puigdomenech Badia, Mehdi Mirza, Alex Graves, Timothy Lillicrap, Tim Harley, David Silver, and Koray Kavukcuoglu. Asynchronous methods for deep reinforcement learning. In *International conference on machine learning*, pages 1928–1937. PMLR, 2016.
- [45] John Schulman, Sergey Levine, Pieter Abbeel, Michael Jordan, and Philipp Moritz. Trust region policy optimization. In *International conference on machine learning*, pages 1889–1897. PMLR, 2015.
- [46] John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. Proximal policy optimization algorithms. *arXiv preprint arXiv:1707.06347*, 2017.
- [47] Long Ouyang, Jeffrey Wu, Xu Jiang, Diogo Almeida, Carroll Wainwright, Pamela Mishkin, Chong Zhang, Sandhini Agarwal, Katarina Slama, Alex Ray, et al. Training language models to follow instructions with human feedback. *Advances in Neural Information Processing Systems*, 35:27730–27744, 2022.
- [48] 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.
- [49] Marwin H S Segler, Mike Preuss, and Mark P Waller. Planning chemical syntheses with deep neural networks and symbolic AI. *Nature*, 555(7698):604–610, 2018.
- [50] Danijar Hafner, Jurgis Pasukonis, Jimmy Ba, and Timothy Lillicrap. Mastering diverse domains through world models. *arXiv preprint arXiv:2301.04104*, 2023.
- [51] Richard S Sutton. Integrated architectures for learning, planning, and reacting based on approximating dynamic programming. In *Machine learning proceedings 1990*, pages 216–224. Elsevier, 1990.
- [52] Fan-Ming Luo, Tian Xu, Hang Lai, Xiong-Hui Chen, Weinan Zhang, and Yang Yu. A survey on model-based reinforcement learning. *arXiv preprint arXiv:2206.09328*, 2022.
- [53] Thomas M Moerland, Joost Broekens, Aske Plaat, Catholijn M Jonker, et al. Model-based reinforcement learning: A survey. *Foundations and Trends® in Machine Learning*, 16(1):1–118, 2023.
- [54] Christopher D Rosin. Multi-armed bandits with episode context. *Annals of Mathematics and Artificial Intelligence*, 61(3):203–230, 2011.

- [55] Jonathan M Stokes, Kevin Yang, Kyle Swanson, Wengong Jin, Andres Cubillos-Ruiz, Nina M Donghia, Craig R MacNair, Shawn French, Lindsey A Carfrae, Zohar Bloom-Ackermann, et al. A deep learning approach to antibiotic discovery. *Cell*, 180(4): 688–702, 2020.
- [56] Austin Derrow-Pinion, Jennifer She, David Wong, Oliver Lange, Todd Hester, Luis Perez, Marc Nunkesser, Seongjae Lee, Xueying Guo, Brett Wiltshire, et al. Eta prediction with graph neural networks in google maps. In *Proceedings of the 30th ACM International Conference on Information & Knowledge Management*, pages 3767–3776, 2021.
- [57] Azalia Mirhoseini, Anna Goldie, Mustafa Yazgan, Joe Wenjie Jiang, Ebrahim Songhori, Shen Wang, Young-Joon Lee, Eric Johnson, Omkar Pathak, Azade Nazi, et al. A graph placement methodology for fast chip design. *Nature*, 594(7862):207–212, 2021.
- [58] Alex Davies, Petar Veličković, Lars Buesing, Sam Blackwell, Daniel Zheng, Nenad Tomašev, Richard Tanburn, Peter Battaglia, Charles Blundell, András Juhász, et al. Advancing mathematics by guiding human intuition with ai. *Nature*, 600(7887):70–74, 2021.
- [59] Charles Blundell, Lars Buesing, Alex Davies, Petar Veličković, and Geordie Williamson. Towards combinatorial invariance for kazhdan-lusztig polynomials. *arXiv preprint arXiv:2111.15161*, 2021.
- [60] Weihua Hu, Matthias Fey, Hongyu Ren, Maho Nakata, Yuxiao Dong, and Jure Leskovec. Ogb-lsc: A large-scale challenge for machine learning on graphs. *arXiv preprint arXiv:2103.09430*, 2021.
- [61] Alice Del Vecchio, Andreea Deac, Pietro Liò, and Petar Veličković. Neural message passing for joint paratope-epitope prediction. *arXiv preprint arXiv:2106.00757*, 2021.
- [62] Marco Pegoraro, Clémentine Dominé, Emanuele Rodolà, Petar Veličković, and Andreea Deac. Geometric epitope and paratope prediction. *bioRxiv*, 2023. doi: 10.1101/2023.06.29.546973. URL <https://www.biorxiv.org/content/early/2023/07/01/2023.06.29.546973>.
- [63] Shengchao Liu, Andreea Deac, Zhaocheng Zhu, and Jian Tang. Structured multi-view representations for drug combinations. In *NeurIPS 2020 ML for Molecules Workshop*, volume 19, 2020.
- [64] Andreea Deac, Yu-Hsiang Huang, Petar Veličković, Pietro Liò, and Jian Tang. Drug-drug adverse effect prediction with graph co-attention. *arXiv preprint arXiv:1905.00534*, 2019.
- [65] 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.

- [66] Hanchen Wang, Tianfan Fu, Yuanqi Du, Wenhao Gao, Kexin Huang, Ziming Liu, Payal Chandak, Shengchao Liu, Peter Van Katwyk, Andreea Deac, et al. Scientific discovery in the age of artificial intelligence. *Nature*, 620(7972):47–60, 2023.
- [67] Ravichandra Addanki, Peter W Battaglia, David Budden, Andreea Deac, Jonathan Godwin, Thomas Keck, Wai Lok Sibon Li, Alvaro Sanchez-Gonzalez, Jacklynn Stott, Shantanu Thakoor, et al. Large-scale graph representation learning with very deep gnnns and self-supervision. *arXiv preprint arXiv:2107.09422*, 2021.
- [68] Jonathan Godwin, Michael Schaarschmidt, Alexander Gaunt, Alvaro Sanchez-Gonzalez, Yulia Rubanova, Petar Veličković, James Kirkpatrick, and Peter Battaglia. Very deep graph neural networks via noise regularisation. *arXiv preprint arXiv:2106.07971*, 2021.
- [69] Marc Brockschmidt. Gnn-film: Graph neural networks with feature-wise linear modulation. In *International Conference on Machine Learning*, pages 1144–1152. PMLR, 2020.
- [70] Andreea Deac, Petar Veličković, and Pietro Sormanni. Attentive cross-modal paratope prediction. *arXiv preprint arXiv:1806.04398*, 2018.
- [71] Hamed Shirzad, Ameya Velingker, Balaji Venkatachalam, Danica J. Sutherland, and Ali Kemal Sinop. Exphormer: Sparse transformers for graphs, 2023.
- [72] Jake Topping, Francesco Di Giovanni, Benjamin Paul Chamberlain, Xiaowen Dong, and Michael M Bronstein. Understanding over-squashing and bottlenecks on graphs via curvature. *arXiv preprint arXiv:2111.14522*, 2021.
- [73] Luis Müller, Mikhail Galkin, Christopher Morris, and Ladislav Rampásek. Attending to graph transformers, 2023.
- [74] Mitchell Black, Zhengchao Wan, Amir Nayyeri, and Yusu Wang. Understanding over-squashing in gnnns through the lens of effective resistance, 2023.
- [75] Francesco Di Giovanni, T. Konstantin Rusch, Michael M. Bronstein, Andreea Deac, Marc Lackenby, Siddhartha Mishra, and Petar Veličković. How does over-squashing affect the power of gnnns?, 2023.
- [76] William L Hamilton. Graph representation learning. *Synthesis Lectures on Artificial Intelligence and Machine Learning*, 14(3):1–159, 2020.
- [77] Qian Huang, Horace He, Abhay Singh, Ser-Nam Lim, and Austin R Benson. Combining label propagation and simple models out-performs graph neural networks. *arXiv preprint arXiv:2010.13993*, 2020.
- [78] Shyam A Tailor, Felix Opolka, Pietro Lio, and Nicholas Donald Lane. Do we need anisotropic graph neural networks? In *International Conference on Learning Representations*, 2021.
- [79] Felix Wu, Amauri Souza, Tianyi Zhang, Christopher Fifty, Tao Yu, and Kilian Weinberger. Simplifying graph convolutional networks. In *International conference on*

- machine learning*, pages 6861–6871. PMLR, 2019.
- [80] David K Duvenaud, Dougal Maclaurin, Jorge Iparraguirre, Rafael Bombarell, Timothy Hirzel, Alán Aspuru-Guzik, and Ryan P Adams. Convolutional networks on graphs for learning molecular fingerprints. *Advances in neural information processing systems*, 28, 2015.
- [81] Federico Errica, Marco Podda, Davide Bacciu, and Alessio Micheli. A fair comparison of graph neural networks for graph classification. *arXiv preprint arXiv:1912.09893*, 2019.
- [82] Enxhell Luzhnica, Ben Day, and Pietro Liò. On graph classification networks, datasets and baselines. *arXiv preprint arXiv:1905.04682*, 2019.
- [83] Peter W Battaglia, Jessica B Hamrick, Victor Bapst, Alvaro Sanchez-Gonzalez, Viniçius 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.
- [84] Peter W Battaglia, Razvan Pascanu, Matthew Lai, Danilo Rezende, and Koray Kavukcuoglu. Interaction networks for learning about objects, relations and physics. *arXiv preprint arXiv:1612.00222*, 2016.
- [85] Devin Kreuzer, Dominique Beaini, Will Hamilton, Vincent Létourneau, and Prudencio Tossou. Rethinking graph transformers with spectral attention. *Advances in Neural Information Processing Systems*, 34, 2021.
- [86] Grégoire Mialon, Dexiong Chen, Margot Selosse, and Julien Mairal. Graphit: Encoding graph structure in transformers. *arXiv preprint arXiv:2106.05667*, 2021.
- [87] Adam Santoro, David Raposo, David G Barrett, Mateusz Malinowski, Razvan Pascanu, Peter Battaglia, and Timothy Lillicrap. A simple neural network module for relational reasoning. *Advances in neural information processing systems*, 30, 2017.
- [88] Chengxuan Ying, Tianle Cai, Shengjie Luo, Shuxin Zheng, Guolin Ke, Di He, Yanming Shen, and Tie-Yan Liu. Do transformers really perform badly for graph representation? *Advances in Neural Information Processing Systems*, 34, 2021.
- [89] Giorgos Bouritsas, Fabrizio Frasca, Stefanos P Zafeiriou, and Michael Bronstein. Improving graph neural network expressivity via subgraph isomorphism counting. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 2022.
- [90] Aditya Grover and Jure Leskovec. node2vec: Scalable feature learning for networks. In *Proceedings of the 22nd ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 855–864, 2016.
- [91] Bryan Perozzi, Rami Al-Rfou, and Steven Skiena. Deepwalk: Online learning of social representations. In *Proceedings of the 20th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 701–710. ACM, 2014.

- [92] Jian Tang, Meng Qu, Mingzhe Wang, Ming Zhang, Jun Yan, and Qiaozhu Mei. Line: Large-scale information network embedding. In *Proceedings of the 24th international conference on world wide web*, pages 1067–1077, 2015.
- [93] Shantanu Thakoor, Corentin Tallec, Mohammad Gheshlaghi Azar, Mehdi Azabou, Eva L Dyer, Remi Munos, Petar Veličković, and Michal Valko. Large-scale representation learning on graphs via bootstrapping. In *International Conference on Learning Representations*, 2021.
- [94] Petar Veličković, William Fedus, William L Hamilton, Pietro Liò, Yoshua Bengio, and R Devon Hjelm. Deep graph infomax. *arXiv preprint arXiv:1809.10341*, 2018.
- [95] Johannes Gasteiger, Stefan Weißenberger, and Stephan Günnemann. Diffusion improves graph learning. *arXiv preprint arXiv:1911.05485*, 2019.
- [96] Pradeep Kr Banerjee, Kedar Karhadkar, Yu Guang Wang, Uri Alon, and Guido Montúfar. Oversquashing in gns through the lens of information contraction and graph expansion. *arXiv preprint arXiv:2208.03471*, 2022.
- [97] Cristian Bodnar, Fabrizio Frasca, Nina Otter, Yuguang Wang, Pietro Lio, Guido F Montufar, and Michael Bronstein. Weisfeiler and lehman go cellular: Cw networks. *Advances in Neural Information Processing Systems*, 34:2625–2640, 2021.
- [98] Cristian Bodnar, Fabrizio Frasca, Yuguang Wang, Nina Otter, Guido F Montufar, Pietro Lio, and Michael Bronstein. Weisfeiler and lehman go topological: Message passing simplicial networks. In *International Conference on Machine Learning*, pages 1026–1037. PMLR, 2021.
- [99] Matthias Fey, Jan-Gin Yuen, and Frank Weichert. Hierarchical inter-message passing for learning on molecular graphs. *arXiv preprint arXiv:2006.12179*, 2020.
- [100] Christopher Morris, Gaurav Rattan, and Petra Mutzel. Weisfeiler and leman go sparse: Towards scalable higher-order graph embeddings. *Advances in Neural Information Processing Systems*, 33:21824–21840, 2020.
- [101] Christopher Morris, Martin Ritzert, Matthias Fey, William L Hamilton, Jan Eric Lenssen, Gaurav Rattan, and Martin Grohe. Weisfeiler and leman go neural: Higher-order graph neural networks. In *Proceedings of the AAAI conference on artificial intelligence*, volume 33, pages 4602–4609, 2019.
- [102] Kimberly Stachenfeld, Jonathan Godwin, and Peter Battaglia. Graph networks with spectral message passing. *arXiv preprint arXiv:2101.00079*, 2020.
- [103] Johannes F Lutzeyer, Changmin Wu, and Michalis Vazirgiannis. Sparsifying the update step in graph neural networks. *arXiv preprint arXiv:2109.00909*, 2021.
- [104] Ameya Prabhu, Girish Varma, and Anoop Nambodiri. Deep expander networks: Efficient deep networks from graph theory. In *Proceedings of the European Conference on Computer Vision (ECCV)*, pages 20–35, 2018.

- [105] Fan R. K. Chung. *Spectral graph theory*, volume 92 of *CBMS Regional Conference Series in Mathematics*. Published for the Conference Board of the Mathematical Sciences, Washington, DC; by the American Mathematical Society, Providence, RI, 1997. ISBN 0-8218-0315-8.
- [106] Alexander Lubotzky. *Discrete groups, expanding graphs and invariant measures*, volume 125 of *Progress in Mathematics*. Birkhäuser Verlag, Basel, 1994. ISBN 3-7643-5075-X. doi: 10.1007/978-3-0346-0332-4. URL <https://doi.org/10.1007/978-3-0346-0332-4>. With an appendix by Jonathan D. Rogawski.
- [107] Giuliana Davidoff, Peter Sarnak, and Alain Valette. *Elementary number theory, group theory, and Ramanujan graphs*, volume 55 of *London Mathematical Society Student Texts*. Cambridge University Press, Cambridge, 2003. ISBN 0-521-82426-5; 0-521-53143-8. doi: 10.1017/CBO9780511615825. URL <https://doi.org/10.1017/CBO9780511615825>.
- [108] Emmanuel Kowalski. *An introduction to expander graphs*, volume 26 of *Cours Spécialisés [Specialized Courses]*. Société Mathématique de France, Paris, 2019. ISBN 978-2-85629-898-5.
- [109] Noga Alon. Eigenvalues and expanders. *Combinatorica*, 6(2):83–96, 1986.
- [110] N. Alon and V. D. Milman. λ_1 , isoperimetric inequalities for graphs, and superconcentrators. *J. Combin. Theory Ser. B*, 38(1):73–88, 1985. ISSN 0095-8956. doi: 10.1016/0095-8956(85)90092-9. URL [https://doi.org/10.1016/0095-8956\(85\)90092-9](https://doi.org/10.1016/0095-8956(85)90092-9).
- [111] Jozef Dodziuk. Difference equations, isoperimetric inequality and transience of certain random walks. *Trans. Amer. Math. Soc.*, 284(2):787–794, 1984. ISSN 0002-9947. doi: 10.2307/1999107. URL <https://doi.org/10.2307/1999107>.
- [112] R. Michael Tanner. Explicit concentrators from generalized N -gons. *SIAM J. Algebraic Discrete Methods*, 5(3):287–293, 1984. ISSN 0196-5212. doi: 10.1137/0605030. URL <https://doi.org/10.1137/0605030>.
- [113] Bojan Mohar. Eigenvalues, diameter, and mean distance in graphs. *Graphs Combin.*, 7(1):53–64, 1991. ISSN 0911-0119. doi: 10.1007/BF01789463. URL <https://doi.org/10.1007/BF01789463>.
- [114] Atle Selberg. On the estimation of Fourier coefficients of modular forms. In *Proc. Sympos. Pure Math., Vol. VIII*, pages 1–15. Amer. Math. Soc., Providence, R.I., 1965.
- [115] Forman. Bochner’s method for cell complexes and combinatorial ricci curvature. *Discrete & Computational Geometry*, 29:323–374, 2003.
- [116] Yann Ollivier. Ricci curvature of metric spaces. *Comptes Rendus Mathématique*, 345(11):643–646, 2007.
- [117] Yann Ollivier. Ricci curvature of markov chains on metric spaces. *Journal of Functional Analysis*, 256(3):810–864, 2009.

- [118] Justin Salez. Sparse expanders have negative curvature. *arXiv preprint arXiv:2101.08242*, 2021.
- [119] Zhenqin Wu, Bharath Ramsundar, Evan N Feinberg, Joseph Gomes, Caleb Geniesse, Aneesh S Pappu, Karl Leswing, and Vijay Pande. Moleculenet: a benchmark for molecular machine learning. *Chemical science*, 9(2):513–530, 2018.
- [120] Damian Szklarczyk, Annika L Gable, David Lyon, Alexander Junge, Stefan Wyder, Jaime Huerta-Cepas, Milan Simonovic, Nadezhda T Doncheva, John H Morris, Peer Bork, et al. String v11: protein–protein association networks with increased coverage, supporting functional discovery in genome-wide experimental datasets. *Nucleic acids research*, 47(D1):D607–D613, 2019.
- [121] Marinka Zitnik, Rok Sosič, Marcus W Feldman, and Jure Leskovec. Evolution of resilience in protein interactomes across the tree of life. *Proceedings of the National Academy of Sciences*, 116(10):4426–4433, 2019.
- [122] Yao Ma, Xiaorui Liu, Neil Shah, and Jiliang Tang. Is homophily a necessity for graph neural networks? *arXiv preprint arXiv:2106.06134*, 2021.
- [123] Jiong Zhu, Ryan A Rossi, Anup Rao, Tung Mai, Nedim Lipka, Nesreen K Ahmed, and Danai Koutra. Graph neural networks with heterophily. *arXiv preprint arXiv:2009.13566*, 2020.
- [124] Xiyuan Wang and Muhan Zhang. How powerful are spectral graph neural networks. In *International Conference on Machine Learning*, pages 23341–23362. PMLR, 2022.
- [125] Dongxiao He, Chundong Liang, Huixin Liu, Mingxiang Wen, Pengfei Jiao, and Zhiyong Feng. Block modeling-guided graph convolutional neural networks. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 36, pages 4022–4029, 2022.
- [126] Prithviraj Sen, Galileo Namata, Mustafa Bilgic, Lise Getoor, Brian Galligher, and Tina Eliassi-Rad. Collective classification in network data. *AI magazine*, 29(3):93–93, 2008.
- [127] Benedek Rozemberczki, Carl Allen, and Rik Sarkar. Multi-scale attributed node embedding. *Journal of Complex Networks*, 9(2):cnab014, 2021.
- [128] Jie Tang, Jimeng Sun, Chi Wang, and Zi Yang. Social influence analysis in large-scale networks. In *Proceedings of the 15th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 807–816, 2009.
- [129] Hongbin Pei, Bingzhe Wei, Kevin Chen-Chuan Chang, Yu Lei, and Bo Yang. Geomcn: Geometric graph convolutional networks. *arXiv preprint arXiv:2002.05287*, 2020.
- [130] Oleg Platonov, Denis Kuznedelev, Artem Babenko, and Liudmila Prokhorenkova. Characterizing graph datasets for node classification: Beyond homophily-heterophily dichotomy. *arXiv preprint arXiv:2209.06177*, 2022.
- [131] Deyu Bo, Xiao Wang, Chuan Shi, and Huawei Shen. Beyond low-frequency information in graph convolutional networks. *arXiv preprint arXiv:2101.00797*, 2021.

- [132] Sitao Luan, Chenqing Hua, Qincheng Lu, Jiaqi Zhu, Mingde Zhao, Shuyuan Zhang, Xiao-Wen Chang, and Doina Precup. Revisiting heterophily for graph neural networks. *arXiv preprint arXiv:2210.07606*, 2022.
- [133] Jiong Zhu, Yujun Yan, Lingxiao Zhao, Mark Heimann, Leman Akoglu, and Danai Koutra. Generalizing graph neural networks beyond homophily. *arXiv preprint arXiv:2006.11468*, 2020.
- [134] Cristian Bodnar, Francesco Di Giovanni, Benjamin Paul Chamberlain, Pietro Liò, and Michael M Bronstein. Neural sheaf diffusion: A topological perspective on heterophily and oversmoothing in gnns. *arXiv preprint arXiv:2202.04579*, 2022.
- [135] William L Hamilton, Rex Ying, and Jure Leskovec. Inductive representation learning on large graphs. *arXiv preprint arXiv:1706.02216*, 2017.
- [136] Yunsheng Shi, Zhengjie Huang, Shikun Feng, Hui Zhong, Wenjin Wang, and Yu Sun. Masked label prediction: Unified message passing model for semi-supervised classification. *arXiv preprint arXiv:2009.03509*, 2020.
- [137] Susheel Suresh, Vinith Budde, Jennifer Neville, Pan Li, and Jianzhu Ma. Breaking the limit of graph neural networks by improving the assortativity of graphs with local mixing patterns. *arXiv preprint arXiv:2106.06586*, 2021.
- [138] Luca Franceschi, Mathias Niepert, Massimiliano Pontil, and Xiao He. Learning discrete structures for graph neural networks. In *International conference on machine learning*, pages 1972–1982. PMLR, 2019.
- [139] Anees Kazi, Luca Cosmo, Seyed-Ahmad Ahmadi, Nassir Navab, and Michael M Bronstein. Differentiable graph module (dgm) for graph convolutional networks. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 45(2):1606–1617, 2022.
- [140] Yue Wang, Yongbin Sun, Ziwei Liu, Sanjay E Sarma, Michael M Bronstein, and Justin M Solomon. Dynamic graph cnn for learning on point clouds. *Acm Transactions On Graphics (tog)*, 38(5):1–12, 2019.
- [141] Jean-Bastien Grill, Florian Strub, Florent Altché, Corentin Tallec, Pierre H Richemond, Elena Buchatskaya, Carl Doersch, Bernardo Avila Pires, Zhaohan Daniel Guo, Mohammad Gheshlaghi Azar, et al. Bootstrap your own latent: A new approach to self-supervised learning. *arXiv preprint arXiv:2006.07733*, 2020.
- [142] Sami Abu-El-Haija, Bryan Perozzi, Amol Kapoor, Nazanin Alipourfard, Kristina Lerman, Hrayr Harutyunyan, Greg Ver Steeg, and Aram Galstyan. Mixhop: Higher-order graph convolutional architectures via sparsified neighborhood mixing. In *international conference on machine learning*, pages 21–29. PMLR, 2019.
- [143] Dan Hendrycks and Kevin Gimpel. Gaussian error linear units (gelus). *arXiv preprint arXiv:1606.08415*, 2016.
- [144] Minjie Wang, Da Zheng, Zihao Ye, Quan Gan, Mufei Li, Xiang Song, Jinjing Zhou, Chao Ma, Lingfan Yu, Yu Gai, Tianjun Xiao, Tong He, George Karypis, Jinyang Li,

- and Zheng Zhang. Deep graph library: A graph-centric, highly-performant package for graph neural networks. *arXiv preprint arXiv:1909.01315*, 2019.
- [145] Yu Rong, Wenbing Huang, Tingyang Xu, and Junzhou Huang. Dropedge: Towards deep graph convolutional networks on node classification. In *International Conference on Learning Representations*, 2020. URL <https://openreview.net/forum?id=Hkx1qkrKPr>.
- [146] Yujun Yan, Milad Hashemi, Kevin Swersky, Yaoqing Yang, and Danai Koutra. Two sides of the same coin: Heterophily and oversmoothing in graph convolutional neural networks. *arXiv preprint arXiv:2102.06462*, 2021.
- [147] Laurens Van der Maaten and Geoffrey Hinton. Visualizing data using t-sne. *Journal of machine learning research*, 9(11), 2008.
- [148] Eli Chien, Jianhao Peng, Pan Li, and Olgica Milenkovic. Adaptive universal generalized pagerank graph neural network. In *International Conference on Learning Representations*. <https://openreview.net/forum/>, 2021.
- [149] Sunil Kumar Maurya, Xin Liu, and Tsuyoshi Murata. Simplifying approach to node classification in graph neural networks. *Journal of Computational Science*, 62:101695, 2022.
- [150] Xiang Li, Renyu Zhu, Yao Cheng, Caihua Shan, Siqiang Luo, Dongsheng Li, and Weining Qian. Finding global homophily in graph neural networks when meeting heterophily. *arXiv preprint arXiv:2205.07308*, 2022.
- [151] Lun Du, Xiaozhou Shi, Qiang Fu, Xiaojun Ma, Hengyu Liu, Shi Han, and Dongmei Zhang. Gbk-gnn: Gated bi-kernel graph neural networks for modeling both homophily and heterophily. In *Proceedings of the ACM Web Conference 2022*, pages 1550–1558, 2022.
- [152] Andreea Deac, Pierre-Luc Bacon, and Jian Tang. Graph neural induction of value iteration. *arXiv preprint arXiv:2009.12604*, 2020.
- [153] Louis-Pascal Xhonneux, Andreea-Ioana Deac, Petar Veličković, and Jian Tang. How to transfer algorithmic reasoning knowledge to learn new algorithms? *Advances in Neural Information Processing Systems*, 34:19500–19512, 2021.
- [154] Borja Ibarz, Vitaly Kurin, George Papamakarios, Kyriacos Nikiforou, Mehdi Benani, Róbert Csordás, Andrew Joseph Dudzik, Matko Bošnjak, Alex Vitvitskiy, Yulia Rubanova, et al. A generalist neural algorithmic learner. In *Learning on Graphs Conference*, pages 2–1. PMLR, 2022.
- [155] Petar Veličković, Adrià Puigdomènech Badia, David Budden, Razvan Pascanu, Andrea Banino, Misha Dashevskiy, Raia Hadsell, and Charles Blundell. The clrs algorithmic reasoning benchmark. In *International Conference on Machine Learning*, pages 22084–22102. PMLR, 2022.

- [156] Danilo Numeroso, Davide Bacciu, and Petar Veličković. Dual algorithmic reasoning, 2023.
- [157] Petar Veličković, Matko Bošnjak, Thomas Kipf, Alexander Lerchner, Raia Hadsell, Razvan Pascanu, and Charles Blundell. Reasoning-modulated representations. In Bastian Rieck and Razvan Pascanu, editors, *Proceedings of the First Learning on Graphs Conference*, volume 198 of *Proceedings of Machine Learning Research*, pages 50:1–50:17. PMLR, 09–12 Dec 2022. URL <https://proceedings.mlr.press/v198/velickovic22a.html>.
- [158] Emanuel Todorov, Tom Erez, and Yuval Tassa. Mujoco: A physics engine for model-based control. In *2012 IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 5026–5033. IEEE, 2012. doi: 10.1109/IROS.2012.6386109. URL <https://doi.org/10.1109/IROS.2012.6386109>.
- [159] 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.
- [160] Junhyuk Oh, Satinder Singh, and Honglak Lee. Value prediction network. In *NIPS*, pages 6118–6128, 2017.
- [161] 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.
- [162] 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.
- [163] 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.
- [164] 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.
- [165] Gregory Farquhar, Tim Rocktäschel, Maximilian Igl, and Shimon Whiteson. Treecq and atreec: Differentiable tree-structured models for deep reinforcement learning. In *ICLR*, 2018.
- [166] Lisa Lee, Emilio Parisotto, Devendra Singh Chaplot, Eric Xing, and Ruslan Salakhutdinov. Gated path planning networks. In *International Conference on Machine Learning*, pages 2947–2955. PMLR, 2018.

- [167] Ramanan Sekar, Oleh Rybkin, Kostas Daniilidis, Pieter Abbeel, Danijar Hafner, and Deepak Pathak. Planning to explore via self-supervised world models. In *International Conference on Machine Learning*, pages 8583–8592. PMLR, 2020.
- [168] 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, pages 2555–2565, 2019.
- [169] Aravind Srinivas, Allan Jabri, Pieter Abbeel, Sergey Levine, and Chelsea Finn. Universal planning networks: Learning generalizable representations for visuomotor control. In *International Conference on Machine Learning*, pages 4732–4741. PMLR, 2018.
- [170] Brandon Amos, Ivan Dario Jimenez Rodriguez, Jacob Sacks, Byron Boots, and J Zico Kolter. Differentiable mpc for end-to-end planning and control. *arXiv preprint arXiv:1810.13400*, 2018.
- [171] Tingwu Wang, Renjie Liao, Jimmy Ba, and Sanja Fidler. Nervenet: Learning structured policy with graph neural networks. In *International Conference on Learning Representations*, 2018.
- [172] Martin Klissarov and Doina Precup. Reward propagation using graph convolutional networks. *arXiv preprint arXiv:2010.02474*, 2020.
- [173] 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.
- [174] Thomas N. Kipf, Elise van der Pol, and Max Welling. Contrastive learning of structured world models. In *ICLR*, 2020.
- [175] Quentin Cappart, Didier Chételat, Elias Khalil, Andrea Lodi, Christopher Morris, and Petar Veličković. Combinatorial optimization and reasoning with graph neural networks. *arXiv preprint arXiv:2102.09544*, 2021.
- [176] Petar Veličković, Rex Ying, Matilde Padovano, Raia Hadsell, and Charles Blundell. Neural execution of graph algorithms. *arXiv preprint arXiv:1910.10593*, 2019.
- [177] Yunhao Tang and Shipra Agrawal. Discretizing continuous action space for on-policy optimization. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 34, pages 5981–5988, 2020.
- [178] Ilya Kostrikov. Pytorch implementations of reinforcement learning algorithms. <https://github.com/ikostrikov/pytorch-a2c-ppo-acktr-gail>, 2018.
- [179] Lukasz Kaiser, Mohammad Babaeizadeh, Piotr Milos, Blazej Osinski, Roy H Campbell, Konrad Czechowski, Dumitru Erhan, Chelsea Finn, Piotr Kozakowski, Sergey Levine, et al. Model-based reinforcement learning for atari. *arXiv preprint arXiv:1903.00374*, 2019.

- [180] Jimmy Lei Ba, Jamie Ryan Kiros, and Geoffrey E Hinton. Layer normalization. *arXiv preprint arXiv:1607.06450*, 2016.
- [181] Greg Brockman, Vicki Cheung, Ludwig Pettersson, Jonas Schneider, John Schulman, Jie Tang, and Wojciech Zaremba. Openai gym. *arXiv preprint arXiv:1606.01540*, 2016.
- [182] 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.
- [183] David Ha and Jürgen Schmidhuber. World models. *arXiv preprint arXiv:1803.10122*, 2018.
- [184] 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, pages 2170–2179, 2019.
- [185] Emilien Dupont, Miguel Bautista Martin, Alex Colburn, Aditya Sankar, Josh Susskind, and Qi Shan. Equivariant neural rendering. In *International Conference on Machine Learning*, pages 2761–2770. PMLR, 2020.
- [186] Jung Yeon Park, Ondrej Biza, Linfeng Zhao, Jan Willem van de Meent, and Robin Walters. Learning symmetric embeddings for equivariant world models. *arXiv preprint arXiv:2204.11371*, 2022.
- [187] Daniel DeTone, Tomasz Malisiewicz, and Andrew Rabinovich. Toward geometric deep slam. *arXiv preprint arXiv:1707.07410*, 2017.
- [188] Victor Garcia Satorras, Emiel Hooeboom, and Max Welling. E (n) equivariant graph neural networks. *arXiv preprint arXiv:2102.09844*, 2021.
- [189] Oriol Vinyals, Igor Babuschkin, Wojciech M Czarnecki, Michaël Mathieu, Andrew Dudzik, Junyoung Chung, David H Choi, Richard Powell, Timo Ewalds, Petko Georgiev, et al. Grandmaster level in StarCraft II using multi-agent reinforcement learning. *Nature*, 575(7782):350–354, 2019.
- [190] Julian Ibarz, Jie Tan, Chelsea Finn, Mrinal Kalakrishnan, Peter Pastor, and Sergey Levine. How to train your robot with deep reinforcement learning: lessons we have learned. *The International Journal of Robotics Research*, 40(4-5):698–721, 2021.
- [191] Alhussein Fawzi, Matej Balog, Aja Huang, Thomas Hubert, Bernardino Romera-Paredes, Mohammadamin Barekatain, Alexander Novikov, Francisco J R Ruiz, Julian Schrittwieser, Grzegorz Swirszcz, David Silver, Demis Hassabis, and Pushmeet Kohli. Discovering faster matrix multiplication algorithms with reinforcement learning. *Nature*, 610(7930):47–53, 2022.
- [192] Balaraman Ravindran. *An algebraic approach to abstraction in reinforcement learning*. University of Massachusetts Amherst, 2004.

- [193] Balaraman Ravindran and Andrew G Barto. Approximate homomorphisms: A framework for non-exact minimization in Markov decision processes. *International Conference on Knowledge Based Computer Systems*, 2004.
- [194] Elise van der Pol, Daniel Worrall, Herke van Hoof, Frans Oliehoek, and Max Welling. MDP homomorphic networks: Group symmetries in reinforcement learning. *Advances in Neural Information Processing Systems*, 33:4199–4210, 2020.
- [195] Sahand Rezaei-Shoshtari, Rosie Zhao, Prakash Panangaden, David Meger, and Doina Precup. Continuous MDP homomorphisms and homomorphic policy gradient. *arXiv preprint arXiv:2209.07364*, 2022.
- [196] Amol Mandhane, Anton Zhernov, Maribeth Rauh, Chenjie Gu, Miaosen Wang, Flora Xue, Wendy Shang, Derek Pang, Rene Claus, Ching-Han Chiang, et al. MuZero with self-competition for rate control in VP9 video compression. *arXiv preprint arXiv:2202.06626*, 2022.
- [197] Ioannis Antonoglou, Julian Schrittwieser, Sherjil Ozair, Thomas K Hubert, and David Silver. Planning in stochastic environments with a learned model. In *International Conference on Learning Representations*, 2021.
- [198] Felix Musil, Andrea Grisafi, Albert P Bartók, Christoph Ortner, Gábor Csányi, and Michele Ceriotti. Physics-inspired structural representations for molecules and materials. *Chemical Reviews*, 121(16):9759–9815, 2021.
- [199] Elise van der Pol, Herke van Hoof, Frans A Oliehoek, and Max Welling. Multi-agent MDP homomorphic networks. *arXiv preprint arXiv:2110.04495*, 2021.
- [200] Darius Muglich, Christian Schroeder de Witt, Elise van der Pol, Shimon Whiteson, and Jakob Foerster. Equivariant networks for zero-shot coordination. *arXiv preprint arXiv:2210.12124*, 2022.
- [201] Arnab Kumar Mondal, Vineet Jain, Kaleem Siddiqi, and Siamak Ravanbakhsh. EqR: Equivariant representations for data-efficient reinforcement learning. In *International Conference on Machine Learning*, pages 15908–15926. PMLR, 2022.
- [202] Dian Wang, Robin Walters, and Robert Platt. SO(2)-equivariant reinforcement learning. *arXiv preprint arXiv:2203.04439*, 2022.
- [203] Arthur Guez, Mehdi Mirza, Karol Gregor, Rishabh Kabra, Sébastien Racanière, Théophane Weber, David Raposo, Adam Santoro, Laurent Orseau, Tom Eccles, Greg Wayne, David Silver, and Timothy Lillicrap. An investigation of model-free planning. In *International Conference on Machine Learning*, pages 2464–2473. PMLR, 2019.
- [204] Jianke Yang, Robin Walters, Nima Dehmamy, and Rose Yu. Generative adversarial symmetry discovery. *arXiv preprint arXiv:2302.00236*, 2023.
- [205] Grigorii A Margulis. Explicit constructions of graphs without short cycles and low density codes. *Combinatorica*, 2(1):71–78, 1982.
- [206] Jean-Pierre Serre. *Trees*. Springer Science & Business Media, 2002.

- [207] 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.
- [208] Richard S Sutton. Generalization in reinforcement learning: Successful examples using sparse coarse coding. In *Advances in neural information processing systems*, pages 1038–1044, 1996.
- [209] Andrew William Moore. Efficient memory-based learning for robot control. 1990.
- [210] David Ha and Jürgen Schmidhuber. Recurrent world models facilitate policy evolution. In *NeurIPS*, pages 2455–2467, 2018.
- [211] Ankesh Anand, Evan Racah, Sherjil Ozair, Yoshua Bengio, Marc-Alexandre Côté, and R Devon Hjelm. Unsupervised state representation learning in atari. In *Advances in Neural Information Processing Systems*, pages 8769–8782, 2019.
- [212] Prafulla Dhariwal, Christopher Hesse, Oleg Klimov, Alex Nichol, Matthias Plappert, Alec Radford, John Schulman, Szymon Sidor, Yuhuai Wu, and Peter Zhokhov. Openai baselines. <https://github.com/openai/baselines>, 2017.

Appendix A

Appendix of Chapter 2

A.1. Proof of Proposition 3.4.1

Let s be one of

$$\begin{pmatrix} 1 & 1 \\ 0 & 1 \end{pmatrix}, \quad \begin{pmatrix} 1 & 0 \\ 1 & 1 \end{pmatrix}.$$

Let $n, n' > 18$ and let e and e' be s -labelled edges in G_n and $G_{n'}$. Then there is a graph isomorphism between $N_2(e)$ and $N_2(e')$ taking e to e' .

PROOF. Note first that, by the homogeneity of the Cayley graphs G_n and $G_{n'}$, we may assume that e and e' emanate from the identity vertex of each graph.

Let G_∞ be the Cayley graph of $\mathrm{SL}(2, \mathbb{Z})$ with respect to the generators

$$S_\infty = \left\{ \begin{pmatrix} 1 & 1 \\ 0 & 1 \end{pmatrix}, \begin{pmatrix} 1 & 0 \\ 1 & 1 \end{pmatrix} \right\}.$$

Let e_∞ be the s -labelled edge emanating from the identity vertex of G_∞ . The quotient homomorphism

$$\mathrm{SL}(2, \mathbb{Z}) \rightarrow \mathrm{SL}(2, \mathbb{Z}_n)$$

induces a graph homomorphism $G_\infty \rightarrow G_n$ sending e_∞ to e . We will show that it restricts to a graph isomorphism

$$N_2(e_\infty) \rightarrow N_2(e).$$

As there is a similar graph isomorphism $N_2(e_\infty) \rightarrow N_2(e')$, the proposition will follow.

Note that two elements of $\mathrm{SL}(2, \mathbb{Z})$ map to the same element of $\mathrm{SL}(2, \mathbb{Z}_n)$ if and only if they differ by multiplication by an element of the kernel K_n . This is

$$K_n = \left\{ \begin{pmatrix} a & b \\ c & d \end{pmatrix} \in \mathrm{SL}(2, \mathbb{Z}) : a \equiv d \equiv 1 \pmod{n} \text{ and } b \equiv c \equiv 0 \pmod{n} \right\}.$$

The graph homomorphism sends edges to edges, and so it is distance non-increasing. Hence it certainly sends $N_2(e_\infty)$ to $N_2(e)$. It is also clearly surjective, because any element of $N_2(e)$ is reached from an endpoint of e by a path of length at most 2, and there is a corresponding path in $N_2(e_\infty)$.

We just need to show that this is an injection. If not, then two distinct vertices g_1 and g_2 in $N_2(e_\infty)$ map to the same vertex in $N_2(e)$. Note then that as elements of $\text{SL}(2, \mathbb{Z})$, $g_2 = g_1 k$ for some $k \in K_n$. There are paths with length at most 3 joining the identity 1 to g_1 and g_2 respectively. Hence, the distance in G_∞ between g_1 and g_2 is at most 6. Therefore, the distance between 1 and $g_1^{-1}g_2$ is at most 6. This element $g_1^{-1}g_2$ lies in K_n . We will show that when $n > 18$, the only element of K_n that has distance at most 6 from the identity is the identity itself. This will imply that $g_1^{-1}g_2 = 1$ and hence $g_1 = g_2$. But this contradicts the assumption that g_1 and g_2 are distinct vertices. Our argument follows that of [205].

The operator norm $\|A\|$ of a matrix $A \in \text{SL}(2, \mathbb{Z})$ is

$$\|A\| = \sup\{|A(v)| : v \in \mathbb{R}^2, |v| = 1\}.$$

This is submultiplicative: $\|AB\| \leq \|A\| \|B\|$ for matrices A and B . It can be calculated as the square root of the largest eigenvalue of $A^t A$. In our case, the operator norms satisfy

$$\left\| \begin{pmatrix} 1 & 1 \\ 0 & 1 \end{pmatrix} \right\| = \left\| \begin{pmatrix} 1 & 0 \\ 1 & 1 \end{pmatrix} \right\| = \frac{1 + \sqrt{5}}{2}.$$

Consider an element

$$K = \begin{pmatrix} a & b \\ c & d \end{pmatrix}$$

of K_n that is not the identity. Since $a \equiv d \equiv 1$ modulo n and $b \equiv c \equiv 0$ modulo n , we deduce that at least one $|a|$, $|b|$, $|c|$ and $|d|$ is at least $n - 1$. Therefore, this matrix acts on one of the vectors $(1,0)^t$ or $(0,1)^t$ by scaling its length by at least $n - 1$. Therefore, $\|K\| \geq n - 1$. Suppose now that K has distance at most 6 from the identity. Then K can be written as a word in the generators of $\text{SL}(2, \mathbb{Z})$ with length at most 6. Therefore, we obtain the inequality

$$\|K\| \leq \left(\frac{1 + \sqrt{5}}{2} \right)^6 < 17.95.$$

Hence, $n < 18.95$ and therefore, as n is integral, $n \leq 18$. □

A.2. Proof of Theorem 3.4.3

For any $\delta > 0$ and $\Delta > 0$, there are only finitely many graphs with maximum vertex degree Δ , Cheeger constant at least δ and non-negative Ollivier curvature.

PROOF. This is a consequence of the main result of Salez [118, Theorem 3]. This states if $G_n = (V_n, E_n)$ is a sequence of graphs with the following properties:

$$\sup_{n \geq 1} \left\{ \frac{1}{|V_n|} \sum_{v \in V_n} \deg(v) \log \deg(v) \right\} < \infty \quad (\text{A.2.1})$$

$$\forall \epsilon > 0, \quad \frac{1}{|E_n|} |\{e \in E_n : \kappa(e) < -\epsilon\}| \rightarrow 0 \text{ as } n \rightarrow \infty, \quad (\text{A.2.2})$$

then

$$\forall \rho < 1, \quad \liminf_{n \rightarrow \infty} \left\{ \frac{1}{|V_n|} |\{i : \mu_i(G_n) \geq \rho\}| \right\} > 0.$$

Here, $\kappa(e)$ is the Ollivier curvature of an edge e and

$$1 = \mu_0(G) \geq \mu_1(G) \geq \dots \geq 0$$

are the eigenvalues of the lazy random walk operator. To prove the theorem, we suppose that on the contrary, there are infinitely many distinct graphs $G_n = (V_n, E_n)$ with maximum vertex degree Δ , Cheeger constant at least δ and non-negative Ollivier curvature. Then

$$\sum_{v \in V_n} \deg(v) \log \deg(v) \leq |V_n| \Delta \log \Delta$$

and so condition A.2.1 is satisfied. Condition A.2.2 is trivially satisfied because the Ollivier curvature of each graph is non-negative. Thus, we deduce that the conclusion of Salez' theorem holds. Setting $\rho = 1 - (\delta^2/4\Delta^2)$, we deduce that a definite proportion of the eigenvalues of the lazy random walk operator are at least $1 - (\delta^2/4\Delta^2)$. In particular, $\mu_1(G_n) \geq 1 - (\delta^2/4\Delta^2)$. Denote the eigenvalues of the normalised Laplacian by

$$0 = \lambda'_0(G_n) \leq \lambda'_1(G_n) \leq \dots$$

These are related to the eigenvalues of the lazy random walk operator by $\lambda'_i(G_n) = 2 - 2\mu_i(G_n)$. Hence, $\lambda'_1(G_n) \leq \delta^2/(2\Delta^2)$. There is a variation of Cheeger's inequality that relates λ'_1 to the *conductance* of the graph. To define this, one considers subsets A of the vertex set, and defines their *volume* to be $\text{vol}(A) = \sum_{v \in A} \deg(v)$. The conductance $\phi(G)$ of a graph G is

$$\phi(G) = \min \left\{ \frac{|\partial A|}{\text{vol}(A)} : A \subset V(G), 0 < \text{vol}(A) \leq \text{vol}(V(G))/2 \right\}.$$

Then, by Chung [105, Theorem 2.2],

$$\phi(G) \leq \sqrt{2\lambda'_1(G)}$$

Hence, in our case,

$$\phi(G_n) \leq \delta/\Delta.$$

Consider any subset A_n of the vertex set that realises $\phi(G_n)$. Thus $0 < \text{vol}(A_n) \leq \text{vol}(V_n)/2$ and $|\partial A_n|/\text{vol}(A_n) = \phi(G_n) \leq \delta/\Delta$. If A_n is at most half the vertices of G_n , then this implies that the Cheeger constant $h(G_n) \leq \delta$. On the other hand, if A_n is more than half the vertices of G_n , we consider its complement A_n^c . Its cardinality $|A_n^c|$ satisfies

$$|A_n^c| \geq \text{vol}(A_n^c)/\Delta.$$

Hence,

$$h(G_n) \leq \frac{|\partial A_n^c|}{|A_n^c|} \leq \frac{|\partial A_n| \Delta}{\text{vol}(A_n^c)} \leq \frac{|\partial A_n| \Delta}{\text{vol}(A_n)} = \phi(G_n) \Delta \leq \delta.$$

In either case, we deduce that the Cheeger constant of G_n is at most δ , contradicting one of our hypotheses. Hence, there must have been only finitely many graphs satisfying the conditions of the theorem. \square

A.3. Cayley graph at infinity is quasi-isometric to a tree

As all vertices of G_n look the same, we focus attention on $N_r(1)$, the r -neighbourhood of the identity vertex. The proof of Proposition 3.4.1 immediately gives the following.

Proposition A.3.1. *Let r be a positive integer satisfying*

$$r < \frac{1}{2} \left(\log \left(\frac{1 + \sqrt{5}}{2} \right) \right)^{-1} \log(n - 1).$$

Then there is a graph isomorphism between the r -neighbourhood of the identity vertex in G_n and the r -neighbourhood of the identity vertex in G_∞ . This isomorphism takes the identity vertex to the identity vertex.

PROOF. As shown in the proof of Proposition 3.4.1, there is a graph homomorphism from $N_r(1)$ in G_∞ to $N_r(1)$ in G_n that is a surjection. If it fails to be an injection, then there is a non-trivial element K in the kernel K_n of $\text{SL}(2, \mathbb{Z}) \rightarrow \text{SL}(2, \mathbb{Z}_n)$ satisfying

$$\|K\| \leq \left(\frac{1 + \sqrt{5}}{2} \right)^{2r}.$$

But any non-trivial element K in K_n satisfies

$$\|K\| \geq n - 1.$$

Rearranging gives the required inequality. \square

This raises the question of the local structure of G_∞ . The answer is well-known: it is ‘tree-like’. Specifically, it is quasi-isometric to a tree. The formal definition of quasi-isometry is as follows.

Definition A.3.2. A *quasi-isometry* between two metric spaces (X_1, d_1) and (X_2, d_2) is a function $f: X_1 \rightarrow X_2$ that satisfies the following two conditions:

(1) there are constants $c, C > 0$ such that, for every $x, x' \in X_1$

$$c d_1(x, x') - c \leq d_2(f(x), f(x')) \leq C d_1(x, x') + C,$$

(2) there is a constant $K \geq 0$ such that for every $y \in X_2$, there is an $x \in X_1$ with $d_2(f(x), y) \leq K$.

If there is such a quasi-isometry, we say that (X_1, d_1) and (X_2, d_2) are *quasi-isometric*.

This forms an equivalence relation on metric spaces. When two metric spaces are quasi-isometric, they are viewed as being ‘essentially the same’ at large scales.

When S and S' are finite generating sets for a group Γ , the graphs $\text{Cay}(\Gamma; S)$ and $\text{Cay}(\Gamma; S')$ are quasi-isometric. Hence, the quasi-isometry type of a finitely generated group is well-defined, and this is the central object of study in geometric group theory.

The group $\text{SL}(2, \mathbb{Z})$ has a finite-index subgroup that is a free group F [206]. If S' denotes a free generating set for F , then $\text{Cay}(F; S')$ is a tree. As passing to a finite-index subgroup preserves its quasi-isometry class, we deduce that the Cayley graph $G_\infty = \text{Cay}(\text{SL}(2, \mathbb{Z}); S_\infty)$ is indeed quasi-isometric to a tree, as claimed above.

A.4. Mixing time properties of expander graphs

Expanders are well known to have small mixing time, in the following sense.

Let G be a graph. We will consider probability distributions π on $V(G)$. The lazy random walk operator M acts on probability distributions as follows. We think of $\pi(v)$ as being the probability of the random walk being at vertex v . If the current location of the walk is at v , then at the next step of the walk, either we stay put with probability $1/2$ or we move to one of its neighbours with equal probability. Then $M\pi$ is the new probability distribution.

In the case when G is k -regular, this takes a particular simple form. The operator M is represented by the matrix $(1/2)I + (1/2k)A$, where A is the adjacency matrix. In that case, any initial distribution π converges under powers of M to the uniform distribution.

This is true for any reasonable notion of convergence, but we will use the $\|\cdot\|_1$ norm, where for two probability distributions π and π' ,

$$\|\pi - \pi'\|_1 = \sum_{v \in V(G)} |\pi(v) - \pi'(v)|.$$

Definition A.4.1. The *mixing time* for a regular graph G is the minimum value of ℓ such that for any starting probability distribution π on the vertex set of G ,

$$\|M^\ell \pi - u\|_1 \leq \frac{1}{4}.$$

Here, u is the uniform probability distribution on the vertex set, and M is the lazy random walk operator.

Expanders have small mixing times in the following very strong sense.

Table 8 – Statistics of the three graph classification datasets studied in our evaluation.

Name	Number of graphs	Avg. nodes/graph	Avg. edges/graph	Metric
ogbg-molhiv	41,127	25.5	27.5	ROC-AUC
ogbg-molpcba	437,929	26.0	28.1	Avg. precision
ogbg-ppa	158,100	243.4	2,266.1	Accuracy
ogbg-code2	452,741	125.2	124.2	F ₁ score

Theorem A.4.2. *For any $k > 0$ and $\delta > 0$, there is a constant $c > 0$ with the following property. If G is a connected k -regular graph on n vertices with Cheeger constant at least $\delta > 0$, then the mixing time for G is at most $c \log(n)$.*

A.5. Additional experimental details and ablations

OGB dataset statistics. We provide additional details on the dataset statistics for the OGB tasks we used in Table 8. More substantial details can be found in the OGB paper [2].

Ablations on propagation graph. Our work concerns sparse expander graphs, determined using the Cayley graphs of the special linear group. We acknowledge that this approach, while theoretically beneficial, is not the only possible way to aid global information propagation in a GNN. Therefore, in this subsection we compare against other classes of approaches.

Our additional baseline methods include: GINs with a *master node*, GINs with a *fully connected* layer (FA), as done in Alon and Yahav [4], and GINs with applying a recently proposed rewiring method, G-RLEF [96].

Note that both the FA method and G-RLEF have motivations related to expanders: the fully-connected graph in the FA method is a trivial *dense* expander, whereas G-RLEF’s rewiring iterations can converge to an expander for certain input graph distributions. Therefore, comparing against these methods allows us to also evaluate the impacts of expander density, as well as proximity to the input graph (since G-RLEF iteratively modifies the input graph). We run G-RLEF for $O(V)$ steps.

The results of our ablative analysis are summarised in Table 9. We find that, as expected, all of our added methods outperform the baseline GIN, demonstrating that oversquashing had been alleviated. When comparing them against each other, however, we find that EGP tends to be highly competitive on two out of the three datasets considered (having the largest average overall). The fully-adjacent dense expander method remains strong on both `ogbg-molhiv` and `ogbg-molpcba`, but runs out of memory as graphs increase in size (as is the case with `ogbg-ppa`).

We find that this collection of ablation studies further supplements the analysis of EGP we have conducted, and serves as a good starting point for further investigations of expander propagation templates with various properties.

Table 9 – Comparative ablation performance of various propagation templates on ogbg-molhiv, ogbg-molpcba and ogbg-ppa. Our baseline model is a GIN [1], using exactly the same implementation as in [2]. All models have *exactly* the same number of parameters—we only modify the connectivity in certain layers depending on the scheme. **N.B.** The fully-connected graph, used in the FA approach [4] can be seen as a dense expander graph, i.e. a special case of EGP. 'OOT' indicates that the method failed to approach baseline performance within five days of training time (while not converging within this time), and 'OOM' indicates out-of-memory (on a V100 GPU).

Model	ogbg-molhiv	ogbg-molpcba	ogbg-ppa
GIN	0.7558 ± 0.0140	0.2266 ± 0.0028	0.6892 ± 0.0100
GIN + master node	0.7668 ± 0.0096	0.2527 ± 0.0064	0.6916 ± 0.0154
GIN + FA [4]	0.7850 ± 0.0090	0.2595 ± 0.0049	OOM
GIN + G-RLEF [96]	0.7802 ± 0.0024	OOT	OOM
GIN + EGP (ours)	0.7934 ± 0.0035	0.2329 ± 0.0019	0.7027 ± 0.0159

Appendix B

Appendix of Chapter 4

B.1. Training information

We used the same experimental setup as presented in Platonov et al. [3]. Results are aggregated over ten random splits of the data, with each run taking 50% of the nodes for training, 25% for validation, and 25% for testing. The following hyperparameters are tuned for all models and baselines, using the average validation performance across the splits:

- Number of GNN/ResNet layers, $L \in \{1, 2, 3, 4, 5\}$.
- The dimensionality of the GNN/ResNet’s latent embeddings, $d \in \{256, 512, 1024\}$.

Additionally, for the ECG models only, the following hyperparameters were swept:

- Embeddings used by ECG, $\tau \in \{MLP, BGRL, MLPBGRL, MLP \rightarrow GNN\}$, referring to:

MLP: Using the embeddings from a pre-trained ResNet;

BGRL: Using the embeddings from a pre-trained BGRL model;

MLPBGRL: Using the normalised concatenation of the ResNet and BGRL embeddings;

MLP→GNN: Using the embeddings from a pre-trained MLP-ECG model of the same type.

Note that, for methods requiring access to labels (such as MLP), a separate set of embeddings is computed for every dataset split (to avoid test data contamination). For self-supervised methods like BGRL, no labels are used, and hence a single set of embeddings is produced for all experiments.

- The number of neighbours sampled per node, $k \in \{3, 10, 20\}$.
- The DropEdge rate, $p_{de} \in \{0.0, 0.5\}$.

The model configuration with the best-performing average validation performance is then evaluated on the corresponding test splits, producing the aggregated performances reported in Table 4.

Table 10 – The best-performing hyperparameters for each GNN propagation rule in our experiments. The only experiment where the baseline model outperforms ECG is the SAGE propagation layer on `amazon-ratings`; hence, the hyperparameters k and p_{de} are irrelevant.

	roman-empire	amazon-ratings	minesweeper	tolokers	questions
ResNet					
L	2	1	5	5	1
d	512	512	512	512	512
GCN					
L	5	2	4	4	3
d	512	512	256	512	256
τ	MLPBGRL	MLP→GNN	MLP	MLP	BGRL
k	3	3	3	3	3
p_{de}	0.5	0.5	0.5	0.5	0.0
SAGE					
L	5	2	5	4	5
d	512	1024	256	256	256
τ	BGRL	Baseline	BGRL	BGRL	BGRL
k	10	—	20	20	10
p_{de}	0.5	—	0.5	0.5	0.0
GAT-sep					
L	5	2	5	5	4
d	512	512	256	256	256
τ	BGRL	MLP→GNN	MLP	BGRL	BGRL
k	10	3	20	20	10
p_{de}	0.5	0.5	0.5	0.5	0.5
GT-sep					
L	5	2	5	5	4
d	512	512	256	256	256
τ	BGRL	MLP→GNN	MLP	MLPBGRL	BGRL
k	20	3	20	20	10
p_{de}	0.5	0.5	0.0	0.0	0.5

The best-performing hyperparameters for each model type on each dataset are given in Table 10. Each individual experiment has been executed on a single NVIDIA Tesla P100 GPU, and the longest training time allocated to an individual experiment has been six hours (on the `questions` dataset).

For convenience, and to assess the relative benefits of various ECG embedding sources, we provide in Table 11 an expanded version of 4, showing the test performance obtained by the tuned version of each ECG variant, for every embedding type.

For additional information, the anonymised code can be found at https://anonymous.4open.science/r/evolving_computation_graphs-97B7/.

Table 11 – Detailed breakdown of model performance on the datasets proposed by Platonov et al. [3]. ResNet, GCN, SAGE, GAT-sep and GT-sep are the baselines, while all the other models are variants of ECG. Red marks the best performance on each dataset for each of the considered GNN architectures and the corresponding ECGs. Accuracy is reported for `roman-empire` and `amazon-ratings`, and ROC AUC is reported for `minesweeper`, `tolokers`, and `questions`.

	roman-empire	amazon-ratings	minesweeper	tolokers	questions
ResNet	65.88 ± 0.38	45.90 ± 0.52	50.89 ± 1.39	72.95 ± 1.06	70.34 ± 0.76
GCN	73.69 ± 0.74	48.70 ± 0.63	89.75 ± 0.52	83.64 ± 0.67	76.09 ± 1.27
MLP-ECG-GCN	83.55 ± 0.39	50.99 ± 0.64	92.63 ± 0.10	84.81 ± 0.25	76.25 ± 0.59
BGRL-ECG-GCN	80.59 ± 0.48	48.99 ± 0.28	92.35 ± 0.10	84.25 ± 0.22	77.50 ± 0.35
MLPBGRL-ECG-GCN	84.53 ± 0.26	50.11 ± 0.60	92.47 ± 0.50	84.73 ± 0.23	77.32 ± 0.31
MLP->GNN-ECG-GCN	84.39 ± 0.22	51.12 ± 0.38	92.56 ± 0.23	84.35 ± 0.31	75.16 ± 0.87
SAGE	85.74 ± 0.67	53.63 ± 0.39	93.51 ± 0.57	82.43 ± 0.44	76.44 ± 0.62
MLP-ECG-SAGE	85.82 ± 0.62	53.32 ± 0.39	94.10 ± 0.08	82.60 ± 0.23	76.13 ± 0.41
BGRL-ECG-SAGE	87.88 ± 0.25	53.12 ± 0.32	94.11 ± 0.07	82.61 ± 0.29	77.23 ± 0.36
MLPBGRL-ECG-SAGE	86.50 ± 0.34	52.34 ± 0.92	94.01 ± 0.07	82.55 ± 0.18	76.55 ± 0.33
MLP->GNN-ECG-SAGE	85.94 ± 0.57	53.45 ± 0.27	93.77 ± 0.12	82.52 ± 0.22	75.53 ± 0.64
GAT-sep	88.75 ± 0.41	52.70 ± 0.62	93.91 ± 0.35	83.78 ± 0.43	76.79 ± 0.71
MLP-ECG-GAT-sep	88.22 ± 0.36	52.98 ± 0.30	94.52 ± 0.20	83.91 ± 0.32	77.30 ± 0.47
BGRL-ECG-GAT-sep	89.62 ± 0.18	52.20 ± 0.57	94.24 ± 0.15	84.23 ± 0.25	77.38 ± 0.18
MLPBGRL-GAT-sep	88.73 ± 0.37	51.06 ± 0.73	94.39 ± 0.20	84.11 ± 0.23	76.97 ± 0.45
MLP->GNN-ECG-GAT-sep	88.04 ± 0.32	53.65 ± 0.39	93.97 ± 0.19	83.75 ± 0.30	75.61 ± 0.74
GT-sep	87.32 ± 0.39	52.18 ± 0.80	92.29 ± 0.47	82.52 ± 0.92	78.05 ± 0.93
MLP-ECG-GT-sep	88.56 ± 0.35	52.68 ± 0.65	93.62 ± 0.27	83.65 ± 0.29	77.82 ± 0.43
BGRL-ECG-GT-sep	89.56 ± 0.16	52.37 ± 0.30	93.55 ± 0.18	82.97 ± 0.26	78.12 ± 0.32
MLPBGRL-GT-sep	88.70 ± 0.30	52.29 ± 0.60	93.52 ± 0.25	84.00 ± 0.24	77.85 ± 0.45
MLP->GNN-ECG-GT-sep	88.62 ± 0.46	53.25 ± 0.39	92.69 ± 0.34	83.41 ± 0.44	75.50 ± 1.13

Appendix C

Appendix of Chapter 6

C.1. Alternate rendition of XLVIN dataflow

See Figure 24 for an alternate visualisation of the dataflow of XLVIN—which is more compact, but does not explicitly sequentialise the operations of the transition model with the operations of the executor.

C.2. Additional description of the Execute function

In Algorithm 2, we provide a symbolic overview of running the executor network X over the local state embedding graph constructed in Algorithm 1.

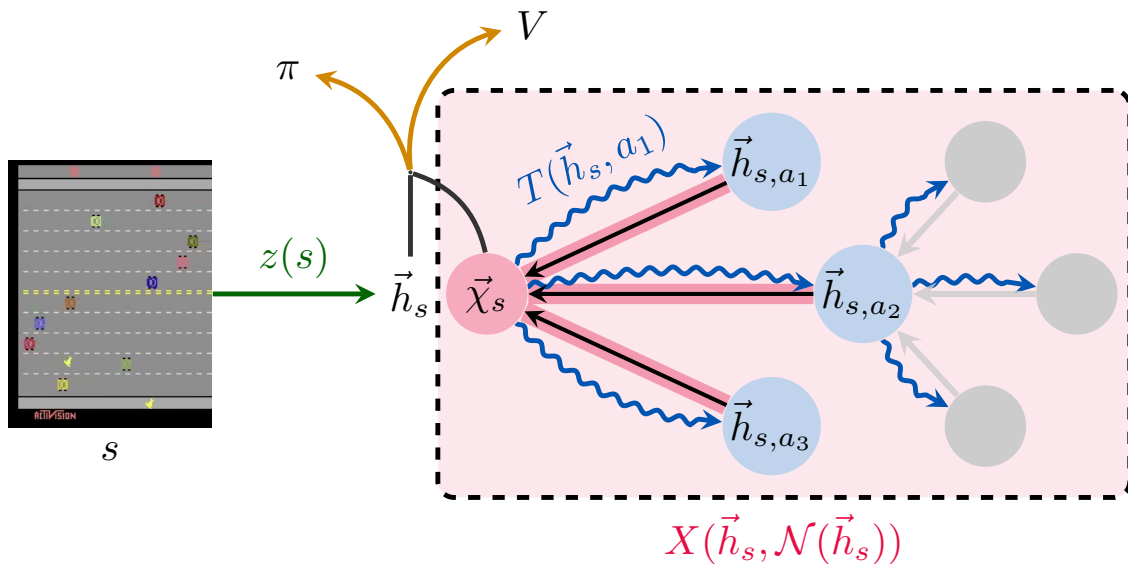


Figure 24 – XLVIN model summary with compact dataflow. The individual modules are explained (and colour-coded) in Section 7.3.1, and the dataflow is outlined in Algorithm 3.

Algorithm 4: Forward propagation of the executor

Input : State embedding \mathbf{h}_s , executor depth K , graph with nodes $\mathbb{S} = \bigcup_{k=0}^K \mathbb{S}^k$ and edges \mathbb{E}

Output: Updated state embedding χ_s

for $\mathbf{h} \in \mathbb{S}^k$ **do**

$\mathcal{N}(\mathbf{h}) = \{\mathbf{h}' \mid \exists \alpha. (\mathbf{h}, \mathbf{h}', \alpha) \in \mathbb{E}\}$; // Construct neighbourhood of node embedding \mathbf{h}

$\mathbb{X}^0 = \bigcup_{k=0}^K \mathbb{S}^k$; // We will use \mathbb{X}^k to store the executor embeddings after k steps; initially, $\mathbb{X}^0 = \mathbb{S}$

for $k \in [0, K)$ **do**

for $\mathbf{h} \in \mathbb{X}^k$ **do**

$\chi = X(\mathbf{h}, \mathcal{N}(\mathbf{h}))$; // Run executor on the neighbourhood of node embedding $\mathbf{h} \in \mathbb{X}^k$

$M_k(\mathbf{h}) = \chi$; // Maintain a mapping, M_k , from input to output embeddings of X at step k

$\mathbb{X}^{k+1} = \{\chi \mid \exists \mathbf{h}. \mathbf{h} \in \mathbb{X}^k \wedge M_k(\mathbf{h}) = \chi\}$; // \mathbb{X}^{k+1} consists of all outputs of M_k

for $\chi \in \mathbb{X}^{k+1}$; // Rebuild neighbourhoods for node embeddings in \mathbb{X}^{k+1} **do**

$\mathcal{N}(\chi) = \{\chi' \mid \exists \mathbf{h} \exists \mathbf{h}'. \mathbf{h} \in \mathbb{X}^k \wedge \mathbf{h}' \in \mathbb{X}^k \wedge M_k(\mathbf{h}) = \chi \wedge M_k(\mathbf{h}') = \chi' \wedge \mathbf{h}' \in \mathcal{N}(\mathbf{h})\}$

$\chi_s = M_{K-1}(\dots M_1(M_0(\mathbf{h}_s)) \dots)$; // To recover χ_s , follow the mappings M_k starting from \mathbf{h}_s

C.3. Environments under study

We provide a visual overview of all eight environments considered in Figure 25.

CartPole. The CartPole environment is a classic example of continuous control, first proposed by [207]. 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.

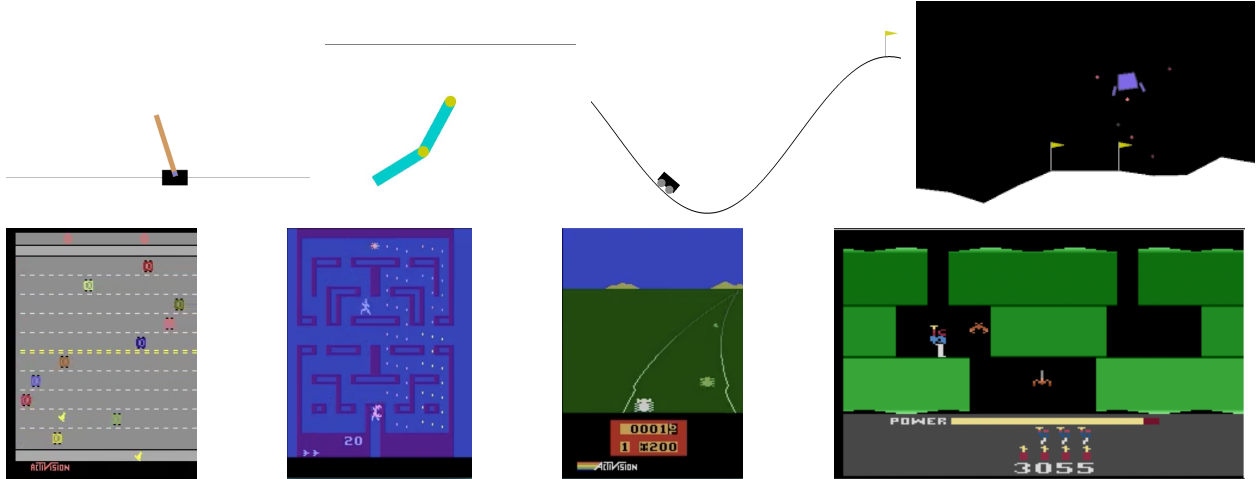


Figure 25 – The eight environments considered within our evaluation: continuous control environments (CartPole-v0, Acrobot-v1, MountainCar-v0, LunarLander-v2) and pixel-based environments (Atari Freeway, Alien, Enduro and H.E.R.O.).

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 [208]. 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 [209]. 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.

LunarLander. The LunarLander task concerns rocket trajectory optimization—a classic topic in optimal control. It concerns navigating a spaceship in two dimensions to a landing pad (at coordinates $(0, 0)$). Successful landing can be achieved by firing the ship’s engines, however this expenses fuel and therefore must be done in a parsimonious manner. The observations are eight-dimensional vectors that include the spaceship’s coordinates, velocity, angle of attack, angular velocity, and whether either of its two legs are in ground contact. Actions correspond to firing the main engine, firing one of the two side engines, or idling. The agent receives shaped negative reward corresponding to its distance to the landing pad, and the

magnitude of its velocity and angle. Further, fixed negative rewards are incurred whenever the engines are fired (more so for the main engine than the side engines). The agent receives shaped positive rewards of +10 whenever its legs make contact with the ground, and a reward of either +100 or -100 upon completing the episode, dependent on whether landing on the landing pad was successful.

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 [42]. 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.

Enduro. Enduro is a game for the Atari 2600, published by Activision in 1983. The goal of the game is to complete an endurance race, overtaking a certain number of cars each day of the race to continue to the next day. 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 [42]. This game is one of the first games with day/night cycles as well as weather changes which makes it particularly visually rich. There are nine different actions we can take in this environment corresponding to staying still as well as accelerating, decelerating, moving left/right and combinations of two of them.

Alien. Alien is a game for the Atari 2600, published by 20th Century Fox in 1982. The goal of the game is to destroy the alien eggs laid in the hallways (similar to the pellets in Pac-Man) while running away from three aliens on the ship. It is a standard part of 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 [42]. There are 18 different actions we can take in this environment corresponding to staying still, firing the flamethrower and moving or firing the flamethrower in eight directions.

H.E.R.O.. H.E.R.O. is a game from Atari 2600, whose goal is to navigate through a mine, clearing obstacles and destroying enemies on the way, in order to rescue a miner at the end of each level. Similarly to Alien, observations in this environment are the full framebuffer of the Atari console while playing the game, which has been appropriately preprocessed as in [42] and the action space is formed of 18 different actions.

Table 12 – Mean scores for CartPole-v0 after training, averaged over 100 episodes and five seeds. Baseline CartPole results reprinted from [5].

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-R	-	199.2 \pm 1.6
XLVIN-CP	-	195.2 \pm 5.0

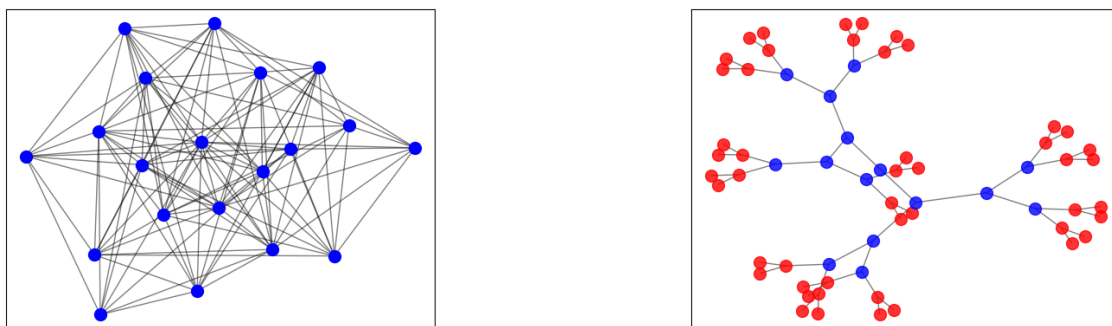


Figure 26 – Synthetic graphs constructed for pre-training the GNN executor: random deterministic (20 states, 8 actions) (**left**) and CartPole (**right**)

C.4. Additional CartPole results

In Table 12, we provide a comparison between XLVIN and several baselines from [5].

C.5. Synthetic graphs

Figure 26 presents the two kinds of synthetic graphs used for pretraining the GNN executor.

In most cases, we pre-train the executor using randomly generated deterministic graphs (left): for $|\mathcal{S}| = 20$ states and $|\mathcal{A}| = 8$ actions, we create a $|\mathcal{S}|$ -node graph. For each state-action pair we select, uniformly at random, the state it transitions to, deterministically. We sample the reward model using the standard normal distribution. Overall, the graphs are

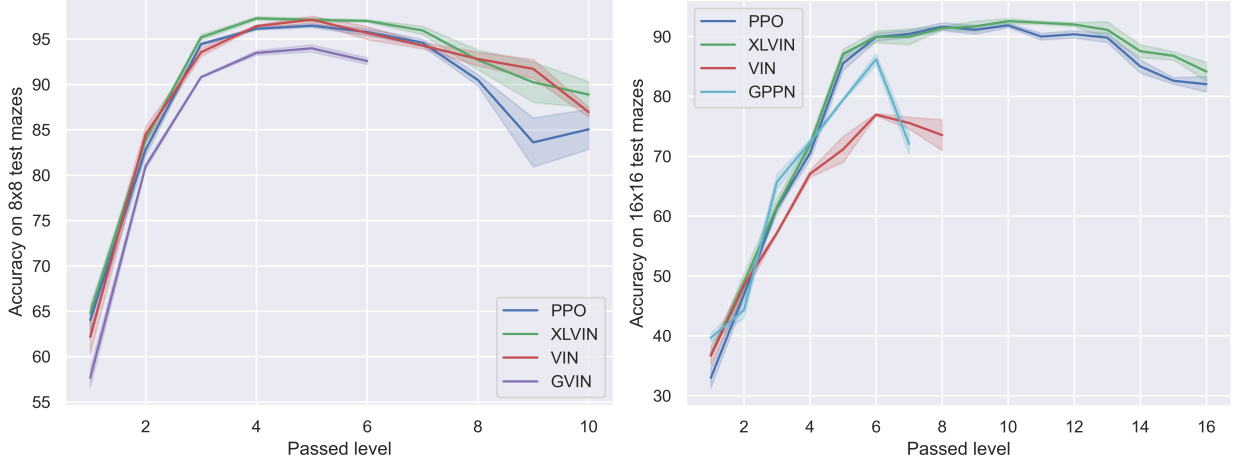


Figure 27 – Success rate on 8×8 (**left**) and on 16×16 (**right**) held-out mazes obtained after passing each level of their respective train mazes. Cut-off curves imply **failure** to pass a difficulty level.

sampled as follows:

$$\tilde{T}(s, a) \sim \text{Uniform}(|\mathcal{S}|) \quad (\text{C.5.1})$$

$$P(s' | s, a) = \begin{cases} 1 & s' = \tilde{T}(s, a) \\ 0 & \text{otherwise} \end{cases} \quad (\text{C.5.2})$$

$$R(s, a) \sim \mathcal{N}(0, 1) \quad (\text{C.5.3})$$

These k -NN style graphs do not assume upfront any structural properties of the underlying MDP, and are a good prior distribution for evaluating the performance of XLVIN.

For CartPole-style environments, we attempt a different type of graph (right). It is 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.

We also attempt using the CartPole graph for pre-training the executor for the other two continuous-observation environments (MountainCar, Acrobot). 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.

C.6. Maze results

As described in the main text, in order to qualitatively assess the transition and executor modules in XLVIN, we evaluated them on a known, fixed and discrete MDP—where optimal values $V^*(s)$ can be trivially computed. Accordingly, we use the 8×8 and 16×16 grid-world mazes proposed by [159]. The observation for this environment consists of the maze image,

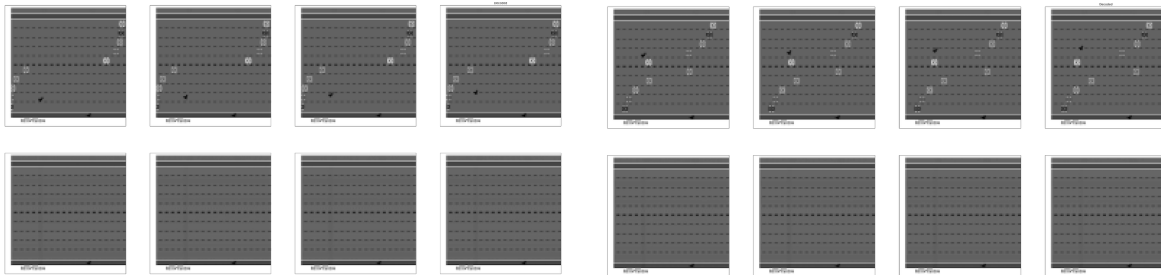


Figure 28 – Freeway frames (**above**) and reconstructions (**below**) using a VAE-style world model.

the starting position of the agent and the goal position. Every maze is associated with a *difficulty* level, equal to the shortest path length between the start and the goal.

Using this concept, we formulate the *continual maze* task: the agent is, initially, trained to solve only mazes of difficulty level 1. Once the agent reaches 95% success rate on the last 1,000 sampled episodes of level d , it advances to level $d+1$ (without observing level d again). If the agent fails to reach 95% within 1,000,000 trajectories, it is not allowed to progress. After each passed difficulty, the agent is evaluated by computing its success rate on held-out test mazes.

Given the grid-world structure, our encoder for the maze environment is a three-layer CNN computing 128 latent features and 10 outputs. The transition function is a three-layer MLP with layer normalisation [180] after the second layer, computing 128 hidden features. We apply the executor until depth $K = 4$, with layer normalisation applied after every step.

Beyond its use for qualitative evaluation, we also perform a comparison of XLVINs against several standard implicit planners in this space (including (G)VIN and GPPN). The results are summarised in Figure 27, and indicate that XLVIN is competitive with all other models, while not making any upfront assumptions about the dynamics of the environment.

C.7. Pixel-based world models

XLVIN is, in principle, agnostic to the choice of transition model. We chose a latent-space transition model in the style of TransE because this aligned the closest with the ATreeC baseline, which also used a latent-space transition model. World models that predict full observations are also plausible.

We attempt replacing our Atari transition model with a variant that learns representations through pixel-based reconstructions (using a VAE objective, as done by [210]). We found that representations obtained in this way were not useful; we observed that most of our state encodings converged to a fixed-point, and that the pixel-space reconstructions completely ignored the foreground observations (see Figure 28). This aligns with prior investigations of VAE-style losses on Atari, which found they tend to overly focus on reconstructing

the background and were less predictive of RAM state than latent-space models, as well as randomly-initialised CNNs [211]. This comparison stands in favour of our approach to using a transition model optimised purely in the lower-dimensional latent space.

C.8. Compute details

We used one V100 GPU from an internally provided cluster for training our model on the Atari environments, for which the training time for one seed per environment was always less than 24 hours. For the classical control and navigation tasks, a 2.7GHz i7 CPU was used.

We used the OpenAI Gym [181] for access to environments, PytorchRL [178] for the PPO implementation and encoder parameters and OpenAI Baselines [212] for environment wrapper capabilities. All of the above are licensed under the MIT license.

C.9. Potential societal impact

Our work studies fundamental insights related to implicit planners. The problem of improving data efficiency, while building better plans is highly important for real world applications. However, this work does not explicitly focus on the engineering efforts for such applications or implications. Instead, we analyse the problem from a theoretical and empirical angle, first identifying bottleneck issues in prior art and then empirically verifying the effects of alleviating the bottleneck on classical control and standard game-playing benchmarks. Therefore, we consider direct societal impact not to be applicable in this setting.