Technical Report

UCAM-CL-TR-990
ISSN 1476-2986

Number 990

UNIVERSITY OF CAMBRIDGE
Computer Laboratory

Probing the foundations of neural algorithmic reasoning

Euan Ong

December 2023

15 JJ Thomson Avenue
Cambridge CB3 0FD
United Kingdom
phone +44 1223 763500

https://www.cl.cam.ac.uk/

This technical report is based on a dissertation submitted May 2023 by the author for the degree of Bachelor of Arts (Computer Science Tripos) to the University of Cambridge, Trinity College.

ISSN 1476-2986
DOI *https://doi.org/10.48456/tr-990*

Abstract

While the field of neural algorithmic reasoning (NAR) – training neural networks to imitate algorithms and using them as algorithmic inductive biases in real-world problems – has risen in popularity, there has been no investigation confirming that its fundamental claims hold in general. Indeed, we argue that such an investigation has so far been infeasible, due to the lack of a general extensible library creating a very high barrier to entry for reproductions and systematic studies.

As such, we develop an extensible laboratory for NAR, by introducing a novel framework for multi-domain, type-driven, declarative ML, and using its components to derive flexible NAR pipelines from first principles through the paradigm of representations-as-types. We use this laboratory to perform systematic analyses, reproductions and comparisons of prior work in NAR, matching (and often beating) state-of-the-art performance across various domains by identifying and alleviating bottlenecks across popular NAR frameworks and architectures.

We then conduct a systematic investigation into the fundamental claims of NAR, in the context of a new synthetic dataset inspired by recent work in neural algorithmics. Through a series of statistically-robust ablation tests, while we confirm the established result that algorithmic modules beat non-algorithmic baselines, we find evidence to refute one of the central claims of NAR, showing that neural algorithmic processors (NAPs) do not overcome the 'scalar bottleneck' of differentiable algorithmic black-boxes (sDABs).

Based on our observations, we develop a new hypothesis to replace this claim: that sDABs instead suffer from an 'ensembling bottleneck' of not being able to execute multiple instances of the same algorithm in parallel, which is alleviated not by NAPs, but by simply using an unfrozen, structurally-aligned neural network. And, through exploring the effects of parallelising sDABs, we not only find strong evidence in support of this hypothesis, but also achieve a long-standing goal of neural algorithmics: developing a way to deterministically distill an algorithm into a robust, high-dimensional processor network that preserves both the efficiency and correctness guarantees of sDABs while avoiding their performance bottleneck.

Contents

1 Introduction

This dissertation explores **neural algorithmic reasoning (NAR)**: a popular subfield of machine learning making a number of bold (but unsubstantiated) fundamental claims. Specifically, **we seek to build a laboratory to find out whether or not these claims are too good to be true**. In this section, we set the scene with a high-level overview of NAR, and briefly outline the aims of this project.

§1.1 Motivation: investigating neural algorithmic reasoning

Setting the scene: building neural networks with algorithmic inductive biases. Throughout the history of computing, algorithms have been used to automatically solve *complex real-world problems* (e.g. finding the shortest path between two cities), provided we can model them mathematically (e.g. as a graph with edge weights in $\mathbb{R}$). In machine learning, many tasks we wish to solve (e.g. finding the shortest path between two cities given weather and traffic conditions) are *algorithmic in nature* but *hard to model mathematically.* As such, there is considerable interest in building neural networks with an **algorithmic inductive bias** – in other words, neural networks incentivised to learn computations that 'look like' those of some algorithm $A : S \to T$ – through the use of **algorithmic modules**: differentiable functions, imitating the behaviour of a particular algorithm, that can be used as a module within a larger neural network.

Designing algorithmic modules: sDABs vs NAPs. One popular way to build an algorithmic module [Vlastelica et al., 2019, Farquhar et al., 2017, Wang et al., 2019, Petersen et al., 2021] is to take an implementation of algorithm A and simply make it differentiable, thereby constructing a **scalar differentiable algorithmic black-box (sDAB)** that can be used as a module in a larger network. An alternative approach is that of *neural algorithmic reasoning* [Veličković and Blundell, 2021]: training a neural network $\hat{A} : \mathbb{R}^n \to \mathbb{R}^n$ to be a **neural algorithmic processor (NAP)** imitating A in high-dimensional space, and using $\hat{A}$ as a module in a larger network.

The fundamental claim of NAR: NAPs alleviate the *scalar bottleneck* of sDABs. Although NAPs require costly pre-training and lack the correctness guarantees of sDABs (especially out-of-distribution [Xu et al., 2020]), Veličković and Blundell [2021] claim that the performance of sDABs is impaired by what they call the **scalar bottleneck** – the requirement that we project rich latent states down to single scalar inputs – and that NAPs alleviate this bottleneck by virtue of operating over a high-dimensional latent space.

This claim has not been systematically explored, and leaves lots of open questions. But while Deac et al. [2021] have shown NAP modules to outperform sDAB modules in the context of the *value iteration* algorithm, **there has been no systematic investigation confirming that NAPs outperform sDABs in general**. And even assuming this result holds, not only do Veličković and Blundell [2021] remain vague about the nature of the scalar bottleneck itself, but **it remains an open problem [Cappart et al., 2021] as to whether there are ways to alleviate the scalar bottleneck while still retaining the correctness and efficiency guarantees of sDABs**.

§1.2 Project aims

As such, this project seeks to **perform an investigation into the fundamental claims of NAR** – to understand *whether NAPs outperform sDABs, and if so, why* – through building a laboratory for neural algorithmics. Note that, not only would such a laboratory both *increase the reproducibility of research* within NAR and *lower the barrier-to-entry for exploration*, but systematically exploring the claims of NAR can help the research community to either increase its confidence that it's pursuing the right ideas, or steer its efforts away from dead-ends.

Specifically, our aims are as follows:

Core aims

As per our project proposal, the core aims of our project are to *lay the groundwork* for a systematic investigation of NAR, by *designing, building and evaluating a laboratory for neural algorithmics.* Specifically, we seek

(a) to **build a laboratory** for training, testing and analysing neural algorithmic reasoners across a variety of domains, consisting of

- a **multi-domain algorithmic reasoning framework (MDARF)** for designing and training step-level NAPs (Chapter 3),
- the **VI-Implicit-Planner** real-world benchmark (Section 4.2), and
- the **Warcraft-Shortest-Path** synthetic benchmark (Section 4.3),

(b) to **evaluate its correctness** (Section 4.1) by using our MDARF to train step-level NAPs and comparing their performance to the state-of-the-art,

(c) to **evaluate its utility for reproduction** (Section 4.2) by using these NAPs to reproduce recent results in NAR (e.g. [Deac et al., 2021]), and

(d) to **evaluate its utility for research** (Section 4.3.2) by using these NAPs to perform a systematic comparison of step-level NAP and natively-differentiable sDAB architectures within our benchmark environments.

Extension aims

Building on the results of our systematic comparison, our main extension is to use our laboratory to *begin exploring the foundations of NAR.* Specifically, it would be desirable

(a) to **expand our systematic comparison** (Section 4.3.2) with a range of ablation tests, in order to understand the influence of each aspect of the NAR regime on model performance,

(b) to **build evidence-based hypotheses** (Section 4.3.3) as to why NAPs behave the way they do, and

(c) to **design more principled DABs** (Section 4.3.4) that preserve both the empirical performance of NAPs and the efficiency and correctness guarantees of sDABs.

2 Preparation

Recall that the overarching question of this dissertation is *whether NAPs outperform sDABs, and if so, why.* In this chapter, we first motivate this question by presenting NAR within its broader context of *neural algorithmics* (Section 2.1). We then analyse how we can build a laboratory to answer it (Section 2.3), before considering the software-engineering components used throughout the course of this project (Section 2.4).

We assume a working knowledge of deep learning; we provide supplementary introductory material and establish a few elements of non-standard notation in Appendix A.

§2.1 A survey of neural algorithmics

The NAR paradigm is situated within **neural algorithmics**: *designing neural networks with an inductive bias towards performing computations similar to those of some given algorithm A, typically for use on real-world tasks for which we believe A to be relevant.* In this section, we conduct a *taxonomy* of neural algorithmics, and analysing the way in which NAR and its claims fit into this picture.

§2.1.1 Building algorithmic modules: sDABs and NAPs

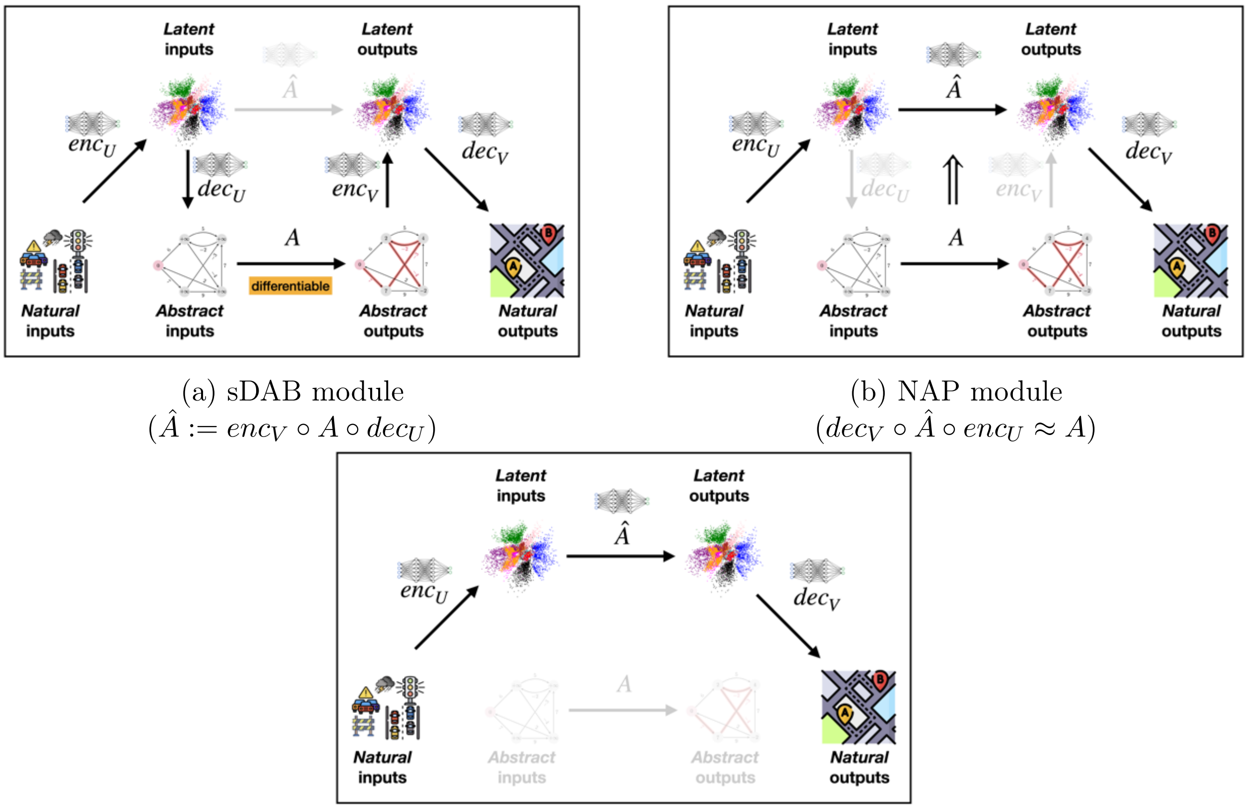
Arguably, the simplest way to give a neural network an inductive bias towards some algorithm A is to give it access to a differentiable module performing computations that 'look like' those of A. Most work in neural algorithmics, therefore, is centred around developing **algorithmic modules**: differentiable algorithms $\hat{A} : \mathcal{U} \to \mathcal{V}$, mimicking the behaviour of some algorithm $A : U \to V$, that can be used as a module within a larger network.

Now, the main technical challenge when building such modules is the *non-differentiability* of most classical algorithms [Paulus et al., 2021], which prevents them from being directly used in an end-to-end neural pipeline. In practice, there are three main ways by which we design algorithmic modules in order to overcome this (Figure G.1):

sDAB: 'Differentiate through the algorithm' In cases where $A : U \to V$ is almost everywhere differentiable (or if we have a differentiable relaxation of A), the most straightforward solution is to use A itself as the algorithmic module – i.e. to set $\hat{A} := enc_V \circ A \circ dec_U$ for learnable functions $dec_U : \mathcal{U} \to U$ and $enc_V : V \to \mathcal{V}$. We refer to such a module as a **scalar differentiable algorithmic black-box (A-sDAB)**, so called because we decode latent states to scalars before passing them through A.

Note that we can further classify sDABs by their method of construction:

- **Natively-differentiable sDABs** are those for which we can calculate gradients for A almost everywhere (e.g. [Petersen et al., 2021]).
- **Gradient-approximation sDABs** are those where, while A has a non-differentiable forward pass, we implement a backward pass that approximates the gradient of A (e.g. [Vlastelica et al., 2019, 2021, Sahoo et al., 2022, Paulus et al., 2021]).
- **Continuous-relaxation sDABs** are those which use a *continuous relaxation* A' of the desired algorithm A – in other words, while A' is differentiable, its forward pass may only be an approximation of A (e.g. [Wang et al., 2019, Petersen et al., 2021]).



(a) sDAB module ($\hat{A} := enc_V \circ A \circ dec_U$)

(b) NAP module ($dec_V \circ \hat{A} \circ enc_U \approx A$)

(c) SNN module ($\hat{A}$ designed to *align* with A)

Figure 2.1: Example usage of algorithmic modules in a natural-input to natural-output pipeline, in increasing order of expressivity

NAP: 'Neuralise the algorithm' Now, especially for complex, discrete algorithms, we may not always be able to transform our desired algorithmic prior A into an sDAB. As such, an alternative way to obtain a differentiable version of A – the approach of **neural algorithmic reasoning** – is to train a neural network to imitate the action of A in a high-dimensional latent space. Specifically, we define our algorithmic module as a learnable function $\hat{A} : \mathcal{U} \rightarrow \mathcal{V}$, and train it alongside $enc_U : U \rightarrow \mathcal{U}$ and $dec_V : \mathcal{V} \rightarrow V$ to satisfy $A \approx dec_V \circ \hat{A} \circ enc_U$.[1] Once trained, we call $\hat{A}$ a **neural algorithmic processor (A-NAP)**, and typically freeze its weights before deploying it in real-world tasks. Following [Cappart et al., 2021], we can further classify NAPs by the level of supervision they receive during training:

- **Algorithm-level NAPs** (e.g. [Veličković et al., 2021]) are those which we train to imitate an algorithm end-to-end, from inputs to outputs. In other words, given an algorithm A and a neural network P, we simply train $\hat{A}, enc_U, dec_V$ to satisfy $A \approx dec_V \circ \hat{A} \circ enc_U$.
- **Step-level NAPs** (e.g. [Deac et al., 2021, Numeroso et al., 2023]) are those which we train to imitate an algorithm end-to-end, but with the extra condition that submodules of the NAP must learn to imitate the behaviour of corresponding submodules (or 'steps') of the algorithm. For instance, given an algorithm $A = A_n \circ ... \circ A_1$ and a neural network $\hat{A} = P_n \circ ... \circ P_1$, we may want $P_i, enc_{T_i}, dec_{T_i}$ to satisfy $A_j \circ ... \circ A_i \approx dec_{T_j} \circ (P_j \circ ... \circ P_i) \circ enc_{T_i}$ for all $1 \leq i \leq j \leq n$. (Note, though, that in practice we might only enforce a weaker property at training time – e.g. that $A_i \circ ... \circ A_1 \approx dec_{T_i} \circ (P_i \circ ... \circ P_1) \circ enc_{T_1}$ for $1 \leq i \leq n$.)

SNN: 'Structurally align with the algorithm' Finally, the weakest way to enforce an algorithmic prior on a neural network is to relax the constraint of our algorithmic module im-

[1]For learnable functions $f, g : \mathcal{U} \rightarrow \mathcal{V}$, we write 'training f, g to satisfy $f \approx g$' to refer to 'training f, g to minimise $L(f(\mathbf{x}), g(\mathbf{x}))$ for $\mathbf{x} \in \mathcal{U}$, for an appropriate loss function L'.

itating some specific algorithm A, and instead use a neural network whose computational structure is 'aligned' with that of some *class* of algorithms C containing A. Specifically, such a **structurally-aligned neural network (SNN)** has the property that the compositional structure of its submodules matches the compositional structure of subroutines in algorithms of class C [Xu et al., 2019]. An important example is the *graph neural network* (and its algorithmically-motivated variants [Veličković et al., 2020a, Tang et al., 2020, Strathmann et al., 2021]), which is known to align with the class of dynamic programming algorithms [Dudzik and Veličković, 2022].

§2.1.2 The claims of NAR

Now, let's dissect the claims of NAR as made in [Veličković and Blundell, 2021] in the context of the tradeoffs between these three approaches to designing algorithmic modules.

Desiderata for algorithmic modules

Recall that the primary goal of developing an algorithmic module is to be able to deploy it in *real-world tasks with algorithmic priors* – in other words, tasks which we suspect might require the use of some algorithm A. In light of this, we ideally want these modules to be *faithful* to their target algorithm A, to be *adaptable* to tasks involving variants A' of their target algorithm A,[2] to be *robust* to noisy / low-data environments, and to have *low cost* at training / inference time.

sDABs, NAPs and the scalar bottleneck

So, given these desiderata, what are the tradeoffs between our algorithmic modules? Recall that, while *we can't always build an sDAB* for an arbitrary algorithm with perfect fidelity, when we are able to build them they *require no training*, tend to be *efficient at inference time* [Vlastelica et al., 2019], and have *strong performance guarantees* both in and out of distribution. By contrast, while NAR gives us a recipe to build an NAP for *any arbitrary algorithm A*, using NAPs incurs the cost of *pre-training* a neural model, *decreased efficiency at inference time* and *weaker performance guarantees* out of distribution [Xu et al., 2020, Veličković et al., 2020b]. Likewise, while we can easily build an SNN for any arbitrary algorithm, these confer *extremely weak algorithmic priors* and have *no performance guarantees* out-of-distribution.

As such, if we are able to choose between all three, the sDAB (with its efficiency and correctness guarantees) seems the obvious choice. But Veličković and Blundell [2021] argue that, despite this theoretical correctness guarantee, sDABs should actually perform *worse* than NAPs, especially on problems involving *complex real-world data*. More specifically, they make the following claim:

Fundamental claim of NAR: NAPs outperform sDABs by avoiding the *scalar bottleneck*.

Suppose we have a real-world task with an algorithmic prior towards $A : U \to V$.

- sDABs $enc_V \circ A \circ dec_U$ perform poorly on tasks for which it is difficult (or impossible) for dec_U to estimate algorithmic inputs satisfying the preconditions of A – for instance, those with *noisy, low-data or partially-observed environments* (where we may not be able to estimate such inputs precisely), and those for which *the underlying algorithm differs slightly from A* (where we may not be able to estimate such inputs at all).
- This problem, known as the **scalar bottleneck**, is likely due to dec_U being forced to

[2]For instance, a task we approximate as involving a shortest-path problem may actually involve a *time-dependent* shortest-path problem.

predict a *imperfect scalar estimate* of each input of A: as A assumes its inputs are free of noise and estimated correctly, any errors in our input will be propagated (and potentially amplified) through the algorithm, and may lead to a suboptimal result.

- NAPs, by virtue of learning to perform A in a *high-dimensional latent space*, avoid this problem, and outperform sDABs when deployed in such environments.

§2.2 The case for investigating the claims of NAR

While some recent results support the fundamental claim of NAR – specifically, that NAPs trained to imitate value iteration *outperform equivalent sDABs* when used as a planning module within model-free RL agents [Deac et al., 2021], and that *restricting the input / output dimensionality* of NAPs trained to imitate an Atari CPU impairs performance when used in a pixel-based contrastive learning pipeline [Veličković et al., 2021] – so far, there has been **no systematic investigation exploring whether NAPs outperform sDABs across a range of tasks**.

As such, **our project aims to investigate the fundamental claim of NAR**, by performing a *systematic comparison of NAP and sDAB architectures* in carefully-selected benchmark problems, verifying *whether NAPs outperform sDABs in general*, and, if so, trying to understand *why this might be the case.*

§2.2.1 Restricting the scope of our investigation to graph-based NAR

We note that most work in NAR [Deac et al., 2021, Veličković et al., 2021, Beurer-Kellner et al., 2022, Numeroso et al., 2023] explores **graph-based NAR**, where our algorithms are functions $A : G[U_N, U_E] \rightarrow G[V_N, V_E]$,[3] and our processors are learnable functions $\hat{A} : G[\mathcal{U}_N, \mathcal{U}_E] \rightarrow G[\mathcal{V}_N, \mathcal{V}_E]$. Accordingly, **we restrict the scope of our investigation to a comparison of graph-based, step-level NAPs**.

These processors are typically implemented as **graph neural networks (GNNs)**: neural networks acting over graphs $G = (V, E)$ (whose nodes u have one-hop neighbourhoods $\mathcal{N}_u = \{v \in V \mid (v, u) \in E\}$ and features $\mathbf{x}_u$), of the form

$$\mathbf{h}_u = \phi(\mathbf{x}_u, \bigoplus_{v \in \mathcal{N}_u} \psi(\mathbf{x}_u, \mathbf{x}_v))$$

for ψ a learnable *message function*, ϕ a learnable *readout function* and $\oplus$ a permutation-invariant *aggregation function.*[4]

§2.2.2 The case for a laboratory for graph-based neural algorithmics

In order to actually conduct this investigation, we need both a *range of sDABs and step-level NAPs*, and a *range of problems to evaluate them on.* But, while sDABs are in general easy to build, implementing a training pipeline for every NAP in every problem domain we want to explore would quickly become cumbersome – and there currently exists no framework flexible enough to support a systematic investigation of NAPs out-of-the-box.

[3](for $G[N, E]$ denoting the type of graphs whose nodes carry values of type N and whose edges carry values of type E)

[4]Note that this ‘template’ can be instantiated in many ways, with different choices of ϕ, ψ and $\oplus$ yielding popular architectures such as GCNs [Kipf and Welling, 2017] and GATs [Veličković et al., 2018]; for more background on GNNs, see e.g. [Hamilton, 2021].

Indeed, we argue that *the systematic evaluation of NAR has so far been infeasible*, due to the lack of a general extensible library creating a very high barrier-to-entry for reproductions and systematic studies.

Prior work and its limitations

The closest thing we have to a framework for training step-level NAPs is the **CLRS Algorithmic Reasoning Benchmark framework (CLRSF)** [Veličković et al., 2022]: a library for evaluating the performance of various GNN architectures on *graph-based representations of algorithms* from the CLRS textbook [Cormen et al., 2009], in order to assess their *abstract reasoning capabilities.*

But although the CLRSF is a powerful, easy-to-use tool to *declaratively train and evaluate graph-based processors on algorithmic problems*, it has a number of limitations making it unsuitable for a systematic investigation of step-level NAPs:

Its underlying ML framework is non-standard. We note that the CLRSF is written in JAX [Bradbury et al., 2018]; as PyTorch [Paszke et al., 2019] is the *de facto* framework for academic ML research, this substantially *increases the barrier-to-entry* for researchers wanting to use the framework in their own investigations.

It's not easily extensible. The CLRSF implements a particular flavour of NAR (a step-based 'input-hint-output' NAR pipeline over graphs) and offers little flexibility outside of this scope (e.g. training on algorithms that aren't in a graph representation), short of rewriting the majority of its 10,000-line codebase.

Its training pipeline isn't designed for NAR. Finally, even in the domain of graph-based NAR, the CLRSF training pipeline has several idiosyncrasies (e.g. a *scalar bottleneck* in its pipeline, as discussed in Section 3.4.2) that deviate from the 'NAR vision' as outlined in Section 2.1.1 and prevent it from being able to directly train the NAPs used in e.g. [Deac et al., 2021] and [Numeroso et al., 2023].

Towards a laboratory for graph-based neural algorithmics

As such, in order to not only facilitate our investigation, but also provide the field of NAR with a high-quality framework enabling reproducible research (in the spirit of *Stable Baselines 3* [Raffin et al., 2021] for reinforcement learning, and *PyTorch Geometric* [Fey and Lenssen, 2019] for geometric deep learning), we seek to **design, build and evaluate a laboratory for neural algorithmics**, consisting of

- a **multi-domain algorithmic reasoning framework (MDARF)** for specifying, training and evaluating NAPs in different domains, and
- a **range of benchmark problems** with which to evaluate it in a controlled setting.

We note that such a laboratory would unify many codebases estimated at thousands of lines of Python each, and would not only lower the cost for ML practitioners to leverage the power of NAR in their own applications, but also provide a platform to drive fundamental research in interpretability, transfer learning and generalisation.

§2.3 Requirements analysis

So, given that we need a laboratory for NAR, what are its requirements? While we will discuss our selection of benchmark problems in Section 4, in this section we consider how the limitations of the CLRSF motivate the high-level desiderata for our MDARF.

Designing for researchers of foundational NAR

To better understand what we want from our MDARF, let's consider its target demographic: *researchers in foundational NAR*. These researchers are typically skilled ML practitioners, who are familiar with frameworks such as PyTorch [Paszke et al., 2019], but who do not necessarily have a substantial software engineering background.

Now, the defining characteristic of this flavour of ML research is *rapid prototyping and experimentation*: for instance, in the course of investigating the foundations of NAR, researchers may wish to quickly train a range of NAPs imitating particular algorithms, deploy these NAPs in real-world problems, and then probe or modify them based on what they see. As such, the fundamental motivating factor behind the design of our framework should be *progressive disclosure*: we want a framework that, in the common case, is as easy to use as the CLRSF, but is also deeply configurable and easily adaptable to more complex pipelines.

Desiderata for the MDARF

Given this, what should our desiderata be for the MDARF? Arguably the most important factor is that our framework should **support the core functionality (and ease-of-use) of the CLRSF, but with a pipeline tailored towards NAR**. But, as a research tool, our framework should also be **flexible** – in other words, open to extension, modification and network surgery. Specifically, unlike the monolithic pipeline of the CLRSF, it should enjoy the following properties:

Compatibility: As NAR research spans a wide range of target domains, our framework should be compatible with researchers' existing codebases, and should work with familiar tools.

Extensibility: As NAR pipelines aren't limited to graphs, our framework should be easily extensible to work with new feature types and new latent spaces.

Modularity: In order to facilitate 'network surgery' on trained NAPs (e.g. extracting the processor from the algorithmic pipeline), our framework should have a modular design.

Composability: As NAR pipelines are complex and often problem-specific, we should be able to compose these modules in different ways to build custom NAR pipelines, while still retaining the type-based abstractions that make the CLRSF easy to work with.

Configurability: Finally, while (like the CLRSF) it should be easy to build 'default' NAR pipelines, all parameters of components of the framework should also be richly configurable.

§2.4 Software engineering tools and techniques

Development methodology. As this project very neatly divides into milestones, each of which iteratively add features to our core deliverable, we adopted the **spiral model** of software development [Boehm, 1988]. For each major milestone, we performed a new design iteration with its own requirements analysis, implementation and evaluation section. We structured these milestones in order to minimise risk, with the first milestones focusing on delivering the core MDARF and its benchmarks, and the higher-risk exploratory experiments being scheduled later. We present a dependency analysis for the implementation modules of this project, alongside its key milestones, in Figure 2.2.

Languages and libraries. Recall that one of our desiderata for our MDARF is *compatibility* with existing research codebases. As Python and PyTorch [Paszke et al., 2019] are the most popular language and framework used in the ML research community (with 61% of research

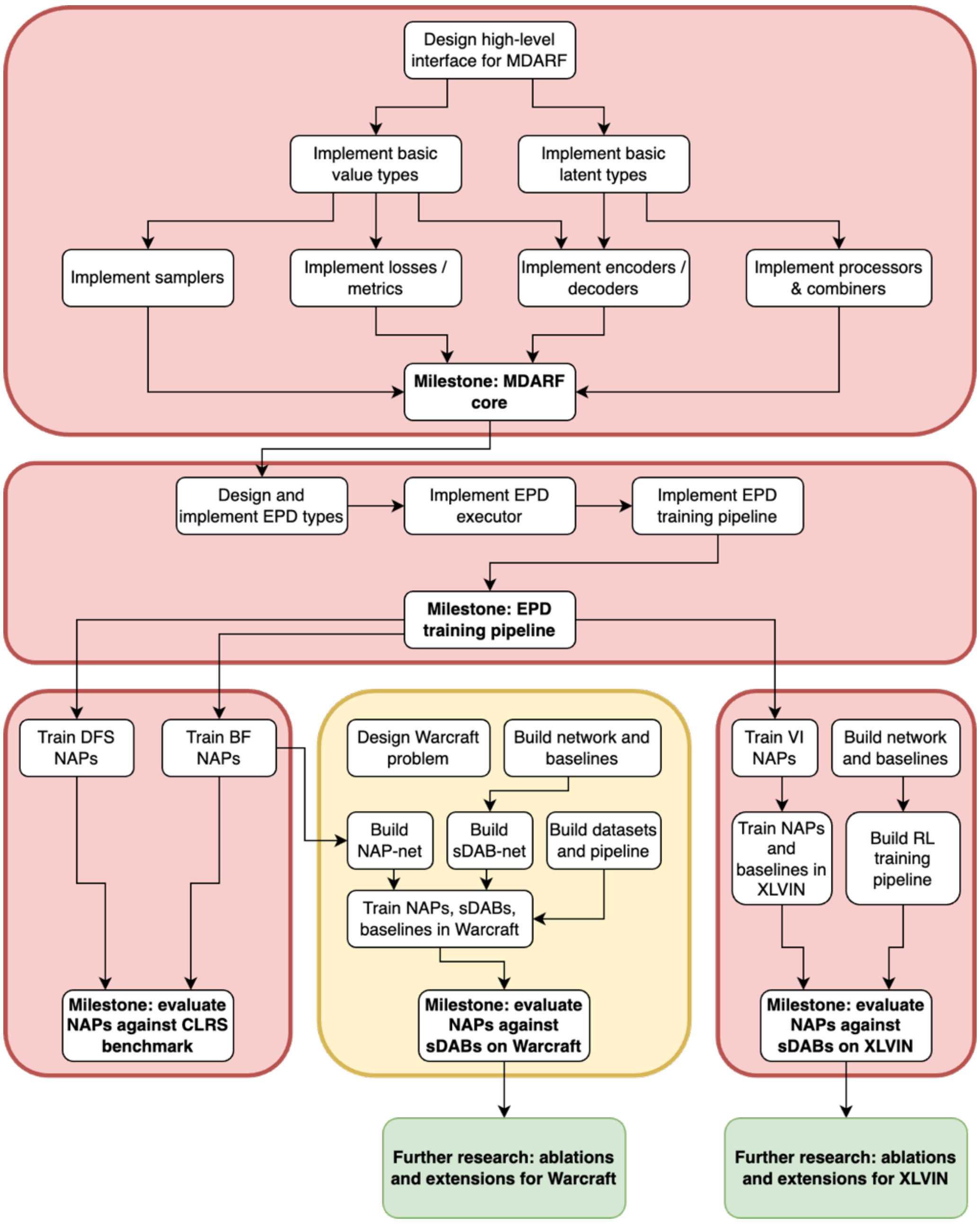

Design high-level interface for MDARF
Implement basic value types
Implement basic latent types
Implement samplers
Implement losses / metrics
Implement encoders / decoders
Implement processors & combiners
Milestone: MDARF core
Design and implement EPD types
Implement EPD executor
Implement EPD training pipeline
Milestone: EPD training pipeline
Train DFS NAPs
Train BF NAPs
Milestone: evaluate NAPs against CLRS benchmark
Design Warcraft problem
Build network and baselines
Build NAP-net
Build sDAB-net
Build datasets and pipeline
Train NAPs, sDABs, baselines in Warcraft
Milestone: evaluate NAPs against sDABs on Warcraft
Train VI NAPs
Build network and baselines
Train NAPs and baselines in XLVIN
Build RL training pipeline
Milestone: evaluate NAPs against sDABs on XLVIN
Further research: ablations and extensions for Warcraft
Further research: ablations and extensions for XLVIN

Figure 2.2: A dependency analysis of the implementation components of this dissertation. All work required for each milestone is indicated by a coloured box; red boxes indicate high-priority work, orange boxes indicate medium-priority work, and green boxes indicate low-priority work.

paper implementations hosted on Papers with Code using PyTorch), we are forced to use these in order to fulfil our desiderata.

Licensing and the open-source community. Although our MDARF and benchmarks are still proofs-of-concept and not quite ready for public use, we plan to improve the MDARF, and once it reaches feature parity with the CLRSF [Veličković et al., 2022], we intend to release it as an open-source library under the *Apache-2.0 license* as per the CLRSF. This license is a permissive license, allowing for both commercial and non-commercial use, whose main conditions are the preservation of copyright and license notices.

As discussed in Section 3.5, in the course of the project the author had to copy and extend a module from *Stable Baselines 3* [Raffin et al., 2021]; as this project uses the MIT license, we are permitted to do so, and (as per the terms of the license) we append the license and copyright notice to the top of the file.

While developing this project, the author also contributed to the open-source community, pushing a pull request to the Python library for Weights and Biases (a tool for visualising and tracking ML experiments). This PR fixed a very subtle, year-old bug in its mechanism for synchronising TensorBoard event logs: while the fix is simple, the bug was non-trivial to find, requiring very careful debugging and introspection of multithreaded code.

Testing strategy. Our development methodology included the continuous writing of over 100 per-component, system-wide and regression tests. In order to reduce the incidental complexity of writing tests, we used the **expect test** pattern [Somers, 2023], implemented through the Python `expecttest` library: instead of manually writing our expected output, we simply run the test case whose output we wish to capture, and the testing engine will automatically update the expected output of our test case with that output. Alongside significantly reducing the overhead of testing code that manipulates complex data structures (which is very prevalent in the MDARF), this pattern even enabled us to write regression tests for ML models, by setting our expected output to be the output of our model when randomly initialised and given a random input.

Code style and documentation. In order to ensure a consistent code style, we used the `black` formatter to auto-format our repository. And, while we made use of Python docstrings to document our code, we also used advanced Python type hints to make the type signatures of our functions as informative as possible. In particular, we used the `jaxtyping` library to type-annotate our tensors, letting us (for instance) write the signature of a function $\mathbb{R}^{b \times n \times h} \to \mathbb{R}^{b \times h}$ as `Float[Tensor, "b n h"] -> Float[Tensor, "b h"]`. While these tensor types can optionally be dynamically checked at runtime, they form a useful, consistent secondary notation to help us keep track of how our functions transform their input and output tensors.[5]

Hardware, version control and backup. We ran all experiments on the `ampere` cluster of the Cambridge Service for Data Driven Discovery (CSD3),[6] with 2x AMD EPYC 7763 64-Core Processor 1.8GHz (128 cores in total), 1000GiB RAM and 1x NVIDIA A100-SXM-80GB GPU. We used GitHub for version control (with a total of more than 160 commits), and backed up all experimental results on Google Drive.

Reproducibility, logging and tracking. To ensure our experiments are reproducible, for each run we saved not only the random seeds used for dataset generation and model initialisation, but also the full configuration for each model, all experimental hyperparameters, and a full dump of the entire codebase.[7] Experimental results were logged to both *TensorBoard* and *Weights & Biases*, with experiment metadata stored locally in a Google Sheet to keep track of the status of pending runs.

§2.5 Starting point

Concepts. Although the author has some prior experience with machine learning, they had no prior experience implementing neural algorithmic reasoners in PyTorch, and no background in reinforcement learning.

[5] We note, though, that we try to keep as much tensor manipulation as possible under the hood, as a large part of the aim of the MDARF is to minimise the amount of time developers need to spend thinking about raw tensors.

[6] The CSD3 is operated by the University of Cambridge Research Computing Service, provided by Dell EMC and Intel using Tier-2 funding from the Engineering and Physical Sciences Research Council (capital grant EP/T022159/1), and DiRAC funding from the Science and Technology Facilities Council.

[7] Note that, while saving the entire codebase with every run may seem wasteful, its size is insignificant compared to that of the model checkpoints saved with each run – and the author has learned from personal experience the frustration of trawling through code history a few months later to piece together the model used for a particular run.

Tools and code. While the VI-Implicit-Planner was built on top of the *Stable Baselines 3* framework for reinforcement learning [Raffin et al., 2021], the MDARF and the rest of the benchmarks were built on top of vanilla PyTorch, with some use of *PyTorch Lightning* [Falcon and The PyTorch Lightning team, 2019] wrappers in order to orchestrate training pipelines.

Prior implementations. While the author referenced the codebases of prior work (such as that of the CLRSF [Veličković et al., 2022] and the XLVIN reproduction of He [2022]) in order to confirm implementation details for the purposes of reproducibility, all frameworks and benchmarks were implemented from scratch (and indeed, often ended up shorter than the relevant official codebase).

3 Implementation

Now that we've conducted our requirements analysis, in this chapter we explore the design (Section 3.1) and implementation (Section 3.2) of our **multi-domain algorithmic reasoning framework (MDARF)**, which evolved into a framework for multi-domain, typed, declarative ML. We also discuss the construction of a CLRSF-style **encode-process-decode (EPD) pipeline** on top of our MDARF (Section 3.4), which we can use to train step-level, graph-based NAPs. Finally, we give an overview of our repository structure (Section 3.5).

§3.1 Design goal: neural algorithmic reasoners *à la carte*

Before we dive into the high-level design of our framework, we briefly reflect on the design considerations that brought us there.

§3.1.1 Perspective: the CLRSF as a restrictive framework for declarative ML

Considering the tradeoffs of the CLRSF from Section 2.3, we notice that most of its ease-of-use comes from its *declarativity* (as illustrated in Figure 3.1). By contrast, most of its limitations in Section 2.2.2, such as the *inflexibility* of its training pipeline and its *lack of extensibility* to non-graph-based algorithms, are ultimately a consequence of the *restrictiveness* of its declarative specification language.

```python
SPECS = types.MappingProxyType({
    ...
    'bellman_ford': {
        'pos': (Stage.INPUT, Location.NODE, Type.SCALAR),
        's': (Stage.INPUT, Location.NODE, Type.MASK_ONE),
        'A': (Stage.INPUT, Location.EDGE, Type.SCALAR),
        'adj': (Stage.INPUT, Location.EDGE, Type.MASK),
        'pi': (Stage.OUTPUT, Location.NODE, Type.POINTER),
        'pi_h': (Stage.HINT, Location.NODE, Type.POINTER),
        'd': (Stage.HINT, Location.NODE, Type.SCALAR),
        'msk': (Stage.HINT, Location.NODE, Type.MASK)
    },
    ...
})
```

Figure 3.1: An example algorithmic specification (for the input, hint and output types of Bellman-Ford) from the CLRSF [Veličković et al., 2022]. The first line of this specification states that each node in the input graph carries a scalar variable named `pos` (i.e. a positional encoding / 'node index'). By simply specifying the input, hint and output types of our algorithm A (alongside a mechanism for generating *algorithmic traces*), the CLRSF will automatically build a training pipeline for A-NAPs, complete with encoders and decoders mapping between these types and the space of latent graphs $G[\mathcal{N}, \mathcal{E}]$

Indeed, as the 'essence' of the CLRS pipeline – the *encoding* of a complex data type into some latent state, the repeated application of a *processor* network to this latent state, and the *decoding* of these intermediate latent states to other complex data types – is actually very general, we could implement a more flexible CLRSF by delegating the work of *building declarative encoders and decoders between complex types and latent spaces* to a more general framework for declarative ML, and implementing a CLRSF-style pipeline as a thin wrapper on top of it.

§3.1.2 Prior work: the Ludwig framework for typed, declarative ML

Now, the main prior work in the space of declarative ML is Uber's *Ludwig* [Molino et al., 2019], a general framework centred around the abstractions of *data types* and *declarative configuration files*. Specifically, Molino et al. [2019] observe that, just like in the CLRSF pipeline, many ML models can be decomposed into an *encoder* lifting input values to latent states, a *combiner* merging and transforming these latent states, and a *decoder* projecting output values from latent states. As such, their framework allows inexperienced users to build such *encode-process-decode (EPD)* pipelines without writing any code, by simply specifying a YAML configuration file (as in Figure 3.2) with the types they wish to map between and the particular network architectures they wish to use for each map.

```
{input_features: [
     {name: title, type: text, encoder: rnn},
     {name: body, type: text, encoder: stacked_cnn}],
 combiner: {type: concat, num_fc_layers: 2},
 output_features: [
     {name: class, type: category},
     {name: tags, type: set}]}
```

Figure 3.2: An example model definition for the *Ludwig* framework, inspired by examples in [Molino et al., 2019]. This definition describes a model taking two textual inputs, encoding them each to single latent states, combining these latent states by concatenating them and passing them through a 2-layer MLP, and projecting out categorical and subset outputs from the combined latent state.

§3.1.3 Designing the MDARF as a Pythonic framework for typed, declarative ML

So, if Ludwig is already a framework for typed, declarative ML, can't we just use it as our MDARF? Unfortunately, not only does it not (yet) support key *compound datatypes* like graphs, its level of abstraction is *too high* for our purpose. Specifically, as Ludwig does not target ML researchers, it optimises for *ease-of-use in the common case* at the cost of *expressivity and flexibility*: users cannot easily modify, compose or introspect on the generated pipeline in Python.

As such, the design of our MDARF combines the simplicity of declarative specifications with the flexibility of PyTorch: while the structure of our framework is heavily inspired by Ludwig's paradigm, we allow users to declaratively specify high-level, typed encoders, processors and decoders, and compose these as native PyTorch modules within Python itself (as illustrated in Figure 3.3). We then implement a CLRSF-like EPD pipeline as a thin wrapper on top of this framework.

```python
@build_enc_dec(LatentSingle)
class InputFeatures(BatchableDataclass):
    title: Text
    body: Text

encoder = build(default_encoder_config(InputFeatures,
 ↪  LatentSingleDims(hidden=128))

def model(input):
    features = encoder(input)
    ...
```

Figure 3.3: A (stylised) model definition for our MDARF. This definition describes (part of) a model that takes two textual inputs and encodes / combines them into a single latent state.

§3.2 A high-level overview of the MDARF

So, now that we've motivated the design of the MDARF, we present a high-level overview of its main features, and briefly discuss how they satisfy our desiderata from Section 2.3.

Object-oriented primitive and compound feature types. In the MDARF, tensors representing features of a given type (e.g. a batch of scalars) are **encapsulated within a class** that handles transformations relevant to that feature type, such as batching, pre-processing and computing losses. As such, the MDARF supports a range of **primitive feature types** such as scalars and categoricals, alongside a range of **compound feature types** such as records $\{l_1 : T_1; ...; l_n : T_n\}$ and graphs $G[T_N, T_E]$. **New primitive or compound types** (e.g. an 'image' type) can easily be added by implementing the relevant interface. For ease of dataset generation, all feature types in the MDARF support **random samplers**, either specified by the user (for primitive types), or automatically derived (for compound types).

Desiderata satisfied: Modularity, Extensibility

Compositional, auto-generated maps between feature and latent-space types. The MDARF supports a range of **latent-space types**, such as representations $\mathcal{R}$ and latent graphs $G[\mathcal{N}, \mathcal{E}]$. If provided with a compound feature type (e.g. a graph with categorical node features and scalar edge features) and a compatible latent type (e.g. a latent graph $G[\mathcal{N}, \mathcal{E}]$), the MDARF can **automatically derive an encoder and decoder** mapping between the datatype and the latent. Alternatively, the user can implement their own.

Desiderata satisfied: Modularity, Extensibility, Composability

Easy pipeline construction from typed processors and combiners. The MDARF also supports a range of **processors** (learnable functions between latent types) and **combiners** (transformations, like `sum` and `concat`, that let us combine latent states). We can thus build NAR pipelines (like that of the CLRSF) at a high level of abstraction, by simply **composing these encoders, processors, combiners and decoders as standard PyTorch modules**.

Desideratum satisfied: Composability

Rich defaults with deep customisability through staged configuration. While the MDARF has **rich support for defaults**, all types, encoders, processors, combiners, decoders and samplers built within it are **deeply configurable and extensible**. In particular, we represent neural modules (from processors to even entire NAR pipelines) as **hierarchical structures of composite classes**. As such, through the power of **defunctionalised configuration** (backed by Google's *Fiddle* library [Saeta and Loper, 2022]), we can **automatically generate default modules** for a particular specification of input and output types, while having the freedom to easily modify and customise any level of their structure within Python itself.

Desideratum satisfied: Configurability

A modular object-oriented framework, implemented in PyTorch. Finally, all the components described above are implemented as **statically-typed Python objects and functions**. In particular, every type of component – from types to latents to samplers – is implemented to a specific **abstract interface**: this offers great flexibility, as we can replace any component with our own implementation, so long as it satisfies the relevant interface. Note also that the framework as a whole is **built upon PyTorch** [Paszke et al., 2019], a widely-used ML library familiar to most academic researchers.

Desiderata satisfied: Compatibility, Extensibility

§3.3 Building the framework: a closer look at the MDARF

Given this high-level overview, let's take a closer look at the implementation of each of the MDARF's core components (as presented in Figure 3.4). While we can (and will) deploy the MDARF on complex algorithmic tasks, to keep things simple we'll explore our pipeline through the lens of the following toy problem:

Problem: neighbourhood statistics

Suppose we have some graph G, whose nodes have some *colour* (red, green or blue), and whose edges have both a *colour* and a *weight*. Train a graph neural network that takes graphs G as input, and computes, for every vertex $v \in G$,

- the sum of the weights of all in-edges $u \to v$ with the same colour as v,
- and the most common colour of neighbouring edges.

§3.3.1 Representing data: feature types and batching

Let's begin our exploration of the MDARF by understanding the way in which it handles the representation and preprocessing of *feature types*.

Feature types. In the MDARF, a **feature** is any datum that can in principle be *batched* and *encoded* to some arbitrary latent space – i.e. any class implementing the `Batchable` interface.

The MDARF supports a range of **primitive feature types**, such as *scalar* and *categorical* values (which encode to latent representations $\mathcal{R}$), and *graph node pointers* (which encode to latent graphs $G[\mathcal{N}, \mathcal{E}]$). The MDARF also supports a range of **compound feature types**, including

- A *graph type*, implemented as a generic class `Graph[NodeT, EdgeT, FeatureT, StructuralT]` and encoding to a *latent graph with graph-level features* $G[\mathcal{N}, \mathcal{E}] \times \mathcal{F}$. Instances of this class represent graphs carrying features `NodeT` (encoding to $\mathcal{N}$) on their nodes, features `EdgeT` (encoding to $\mathcal{E}$) on their edges, 'graph-level' features `FeatureT` (encoding to $\mathcal{F}$), and 'structural' features `StructuralT` (encoding to $G[\mathcal{N}, \mathcal{E}]$ – e.g. node pointers).[1]

[1] *Tradeoff: structural features.* As we'll see later on, ideally encoders for compound feature types should be

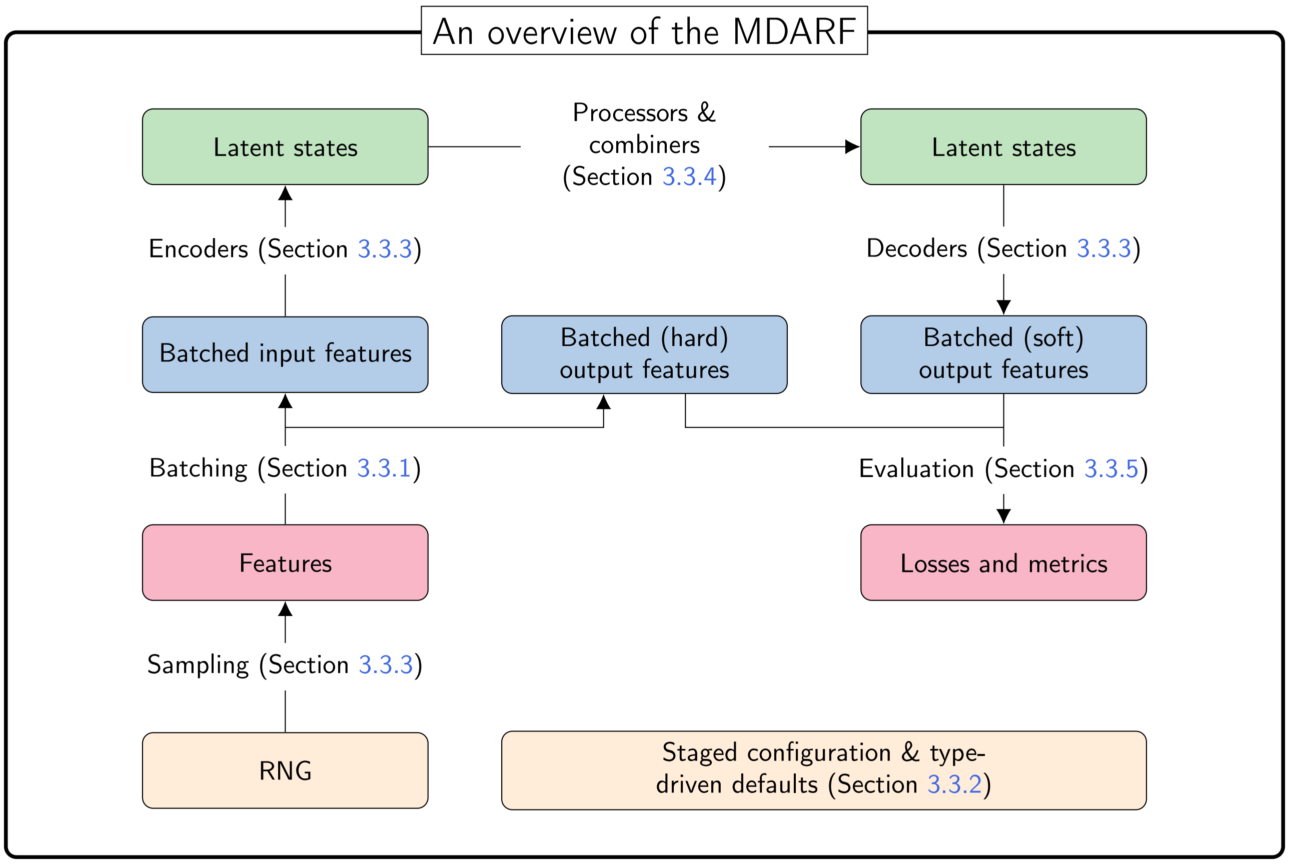


Figure 3.4: An overview of the interactions between the components of the MDARF.

- Functionality for building *batchable dataclasses* (i.e. record feature types) – dataclasses whose fields subclass `Batchable`, which inherit all the functionality of regular feature types. To build one, one simply defines a dataclass subclassing `BatchableDataclass`, as illustrated in Figure 3.5.

For ease of generating algorithmic inputs, each feature type stores a single datum of the relevant feature (as opposed to a batch). We illustrate the relevant feature types for our toy problem in Figure 3.5.

Batching and preprocessing. Now, in order to do anything interesting with our features, we must first batch and preprocess them into a form suitable for ingestion by a neural network. In a typical ML pipeline, a given batched feature will typically be pre- and post-processed through three different representations:

- **Hard:** The raw, potentially *discrete-valued representation* of our feature (typically the representation in which it is stored in the dataset).

  Example: For a categorical variable in $\{1, ..., n\}$, a batch of indices `[1,2,...]`

- **Soft input:** A pre-processed, *continuous-valued* representation of our feature that we can pass directly to our encoder.

  Example: For a categorical variable in $\{1, ..., n\}$, a batch of per-class probability distributions `[[0.9,0.1,...],[0.2,0.7,...],...]`

compositional: in other words, the encoder of some graph type $G[N, E]$ should be built from the encoders for types N and E. Unfortunately, the CLRSF contains a number of features living on graphs, such as node masks and node pointers, which, when encoded, modify both the nodes and the edges of the latent graph. As such, we make the tradeoff to treat them separately, even if they are *semantically* a node or an edge feature: while this makes the framework slightly more awkward from the user-facing perspective, it substantially reduces implementation complexity.

```python
class Colour(Enum):
    red = 0; green = 1; blue = 2

@build_enc_dec_and_sampler(LatentSingle)
@dataclass
class EdgeData(BatchableDataclass):
    colour: Categorical[Colour]
    weight: Scalar

@build_enc_dec_and_sampler(LatentSingle)
@dataclass
class OutputData(BatchableDataclass):
    most_common_colour: Categorical[Colour]
    matching_weight_sum: Scalar

InputGraph = Graph[Categorical[Colour], EdgeData, Empty, Empty]
OutputGraph = Graph[OutputData, Empty, Empty, Empty]

example_graph: InputGraph = Graph(
    nodes=[Categorical(Colour.red), ...],
    edges=[[EdgeData(colour=Categorical(Colour.red), weight=Scalar(0.6)), ...],
    ↪  ...],
    features=Empty(), pointers=Empty())
```

Figure 3.5: An illustration of the feature types for our toy problem. We define our input graph type to store a categorical variable (representing colour) on each node, and an instance of the `EdgeData` dataclass on each edge. Our output graph type stores an instance of the `OutputData` dataclass on each node. Note that both graphs have neither graph-level features nor structural features.

- **Soft output:** A *continuous-valued* representation of our feature produced by our decoder.[2]

  Example: For a categorical variable in $\{1, ..., n\}$, a batch of per-class logits `[[1.1,-1.0,...],[-0.4,0.8,...],...]`

As each of these representations should be treated differently, we opt to *distinguish between them at type-level*, with batched representations of feature types `T` subclassing either `Batched[T, Hard]`, `Batched[T, SoftInput]` or `Batched[T, SoftOutput]` as appropriate.

This lets us write interfaces specifying functions over batched types that are *polymorphic in the underlying type being batched* – for instance,

```python
def loss(
    pred: Batched[T, SoftOutput], target: Batched[T, Hard]
) -> MaskedTensorTree:
    ...
```

For each family `Batched[T, EncodingT]` of batched feature types for `T`, we provide ways to instantiate batched feature types from lists of `T`s, to convert between representations, and to compute losses and evaluation metrics between predictions `Batched[T, SoftOutput]` and ground-truth values `Batched[T, Hard]`. Finally, noting that our batched feature types are

[2]Note that, in order to allow the *factoring of latent states through feature values* as in [Ibarz et al., 2022], there should ideally exist a (differentiable) map from this representation to our soft input representation.

tensor containers – tree-based, hierarchical structures of tensors – we provide a *principled set of structural transformations* we can apply to them, such as methods to stack, split and reshape the tensors within them.[3]

§3.3.2 Building modules: staged configuration and type-driven defaults

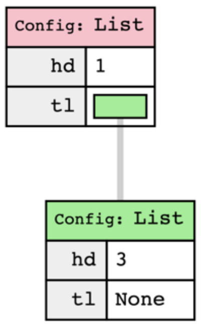
So, now we have our feature types, to make progress on our toy problem we must be able to *randomly generate* input features, and then *lift* them to latent space. To do so, we must build modules, such as samplers, encoders and decoders, for our feature types.

Type-driven defaults. Recall that, as we've introduced *compound type constructors*, our feature types may be arbitrarily large and complex, making the manual specification of these modules far too cumbersome to be practical. As such, the MDARF allows for the *automatic, type-driven generation of compositional modules for compound types*, using the *registry pattern* [Fowler et al., 2002] to allow users to register *default module factories* for a given type (or pair of types). Note that, as the default samplers / encoders / decoders for compound types are typically *compositional* (e.g. a sampler for $G[N, E]$ can be built from samplers for N and for E), so long as we register default module implementations for each primitive type, we can build principled defaults for compound types by recursion on the structure of the compound type.

Staged configuration for auto-generated modules. As these default modules have many parameters we may wish to configure (e.g. the sampling distribution used, or the particular submodules used to generate encoders for batchable dataclasses), these module factories return **staged configuration trees**, backed by Google's *Fiddle* configuration framework [Saeta and Loper, 2022]. As these configuration trees are abstract syntax trees representing nested applications of constructors (Figure 3.6), we can modify the arguments of our module (or submodule) constructors as required, before building the tree to obtain the module itself.

```
>>> from fiddle import Config, build
>>> @dataclass
... class List:
...     hd: int
...     tl: NoneType | "List"
...
>>> conf = Config(List, hd=1, tl=Config(List, hd=2, tl=None))
>>> conf.tl.hd = 3
>>> build(conf)
List(hd=1, tl=List(hd=3, tl=None))
```

(a) We illustrate how a nested structure can be staged, modified and subsequently built.



(b) Rendered config tree for `conf`

Figure 3.6: Example usage of the *Fiddle* configuration framework.

[3]*An aside on TensorDict.* When writing this report, the author discovered the *TensorDict* library [Moens, 2023]. This now-popular library provides wrappers and decorators to build dictionaries and dataclasses of tensors supporting similar structural transformations to those we implemented for batched feature types. Unfortunately, this library did not exist when the project was started in 2022; were the author to have restarted this project in 2023, they would have built the core datatypes of the MDARF around this library instead of reimplementing its functionality from scratch.

§3.3.3 Using feature types: samplers, latent spaces, encoders and decoders

So, now we've seen how to build modules from feature types, let's take a look at how we use modules to sample, encode and decode feature types.

Samplers. For the purposes of NAR, we will find it helpful to be able to randomly generate instances of the complex feature types we use in our networks. As such, the MDARF provides functionality for building **sampler modules** `Sampler[T]`, that act as wrapped generators for features of type `T`. In order to allow for deterministic generation of datasets, all randomness within each sampler is provided by a NumPy random number generator passed in as an argument.

The MDARF comes with default samplers for each primitive type (e.g. a discrete uniform sampler for categoricals), along with composite samplers for compound types: the default graph sampler for graphs $G[N, E]$, for instance, is built by sampling a graph structure from the default adjacency matrix sampler (here, a sampler for Erdős-Renyi graphs [Erdös and Rényi, 1959]) and populating its nodes and edges through the default samplers for types N and E respectively. Note that, if we desire the use of a custom sampler for nodes or edges, we can simply modify the default configuration tree to include our custom sampler (an example of the *strategy pattern* [Gamma et al., 1994]). For illustration, we demonstrate the construction of samplers for our toy problem in Figure 3.7.

Latent states. Once we've generated our input features, we're ready to lift them to **latent states**. In a similar way to feature types, these are represented as *tensor containers* – tree-based, hierarchical structures of tensors – and are equipped with appropriate methods for structural transformation. Driven by the goal of reproducing work in graph-based NAR, the MDARF currently supports latent representations $\mathcal{R}$ (as instances of `LatentSingle`) and latent graphs $G[\mathcal{N}, \mathcal{E}]$ (as instances of `LatentGraph`), and is easily extensible to a wider range of latent spaces.

Encoders and decoders. Given an arbitrary feature type T and a latent space $\mathcal{L}$, the MDARF supports the construction of learnable **encoders** $T \to \mathcal{L}$ and **decoders** $\mathcal{L} \to T$. As these encoders should *in spirit* 'do no useful work' beyond simply changing the structural representation of our data [Veličković et al., 2022], our default encoders are *linear maps*. As before, the MDARF comes with default encoders for each primitive type (e.g. a linear map lifting a scalar to a latent representation), and supports the automatic derivation of maps between compound feature types and latent spaces by recursion on the structure of the compound type. For illustration, we demonstrate the construction of encoders and decoders for our toy problem in Figure 3.8.

§3.3.4 Latent-space transformations: processors and combiners

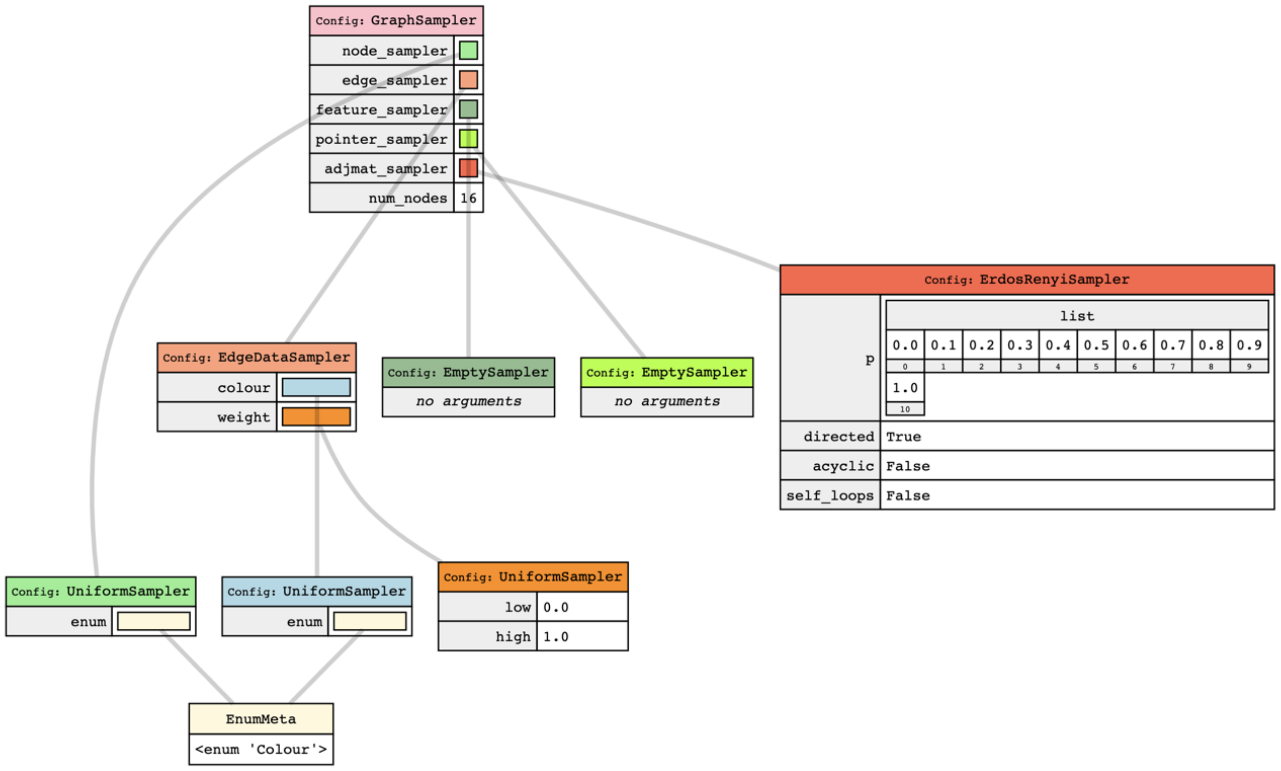
So, now that we can generate features, and map between feature types and latent spaces, let's take a brief look at the transformations we can apply to our latent states themselves.

These are roughly split into two categories: **combiners** (non-parameterised aggregation functions, such as summation or concatenation, that can merge arbitrarily many latent states of the same type into a single latent state) and **processors** (learnable functions performing computations on latent states). While combiners satisfy a light interface to allow for easy 'plug-and-play' usage in the automatic derivation of encoders, a processor is simply any PyTorch `nn.Module` mapping between appropriate latent states.

In Figure 3.9, we use one of our default processors to demonstrate how, using the encoders and decoders we built earlier, we can pass our feature inputs through an end-to-end neural pipeline.

```
>>> rng = np.random.default_rng()
>>> sampler_config = default_sampler_config(InputGraph)
>>> sampler = build(sampler_config)
>>> sampled_batch: Batched[InputGraph, Hard] = sampler.batch(rng, 10)
```

(a) To randomly sample `InputGraph`s, we simply build a default sampler.



(b) A rendering of the configuration tree for `sampler_config`. Note that the default graph generator produces directed Erdős-Renyi graphs $G(n, p)$, with $n = 16$ and p sampled uniformly at random from $[0.0, 0.1, ..., 1.0]$ as per [Ibarz et al., 2022].

Figure 3.7: Generating random samples of our `InputGraph` feature type.

§3.3.5 Evaluating performance: losses and metrics

Finally, once we've passed our input features through our pipeline and obtained our (soft) output, the only thing left to do is to compute the *loss* (and other performance metrics) between our output and the ground truth.

Now, computing metrics between complex compound types is non-trivial, as not only must we build a structure storing all the losses for each of the primitive types that make up the compound type, but we must also be able to perform *reductions* on these structures (e.g. averaging over a batch dimension, or combining all the losses from primitive types into a single value). Furthermore, as (for our EPD pipeline) we introduce a compound type representing a variable-length trajectory, we must handle the complication of *masked tensors* introduced by the batching of variable-length sequences.

We satisfy these requirements by introducing **masked tensor trees**: wrapped trees built from Python containers (e.g. dictionaries and lists), for which every leaf is a *masked tensor* (i.e. a wrapped tensor with an associated Boolean mask, together with various functions for e.g. averaging along dimensions while respecting the mask). These tensor trees support methods for reshaping, stacking and averaging, along with a generalised `reduce` function for folding the tree structure into a single value. As such, when we take the loss (or any other performance metric)

```
>>> latent_dims = LatentGraphDims(hidden_node=16, hidden_edge=16,
 ↪  hidden_feature=16)
>>> encoder = build(default_encoder_config(InputGraph, latent_dims))
>>> decoder = build(default_decoder_config(latent_dims, OutputGraph))
>>> encoder
GraphEncoder(
  (node_encoder): LinearEncoder(
    (linear): Linear(in_features=3, out_features=16, bias=True)
  )
  (edge_encoder): BatchableDataclassEncoder(
    (colour): LinearEncoder(
      (linear): Linear(in_features=3, out_features=16, bias=True)
    )
    (weight): LinearEncoder(
      (linear): Linear(in_features=1, out_features=16, bias=True)
    )
  )
  (feature_encoder): EmptyEncoder()
  (pointer_encoder): EmptyEncoder()
)
```

Figure 3.8: Building encoders and decoders between our `InputGraph` / `OutputGraph` feature types, and a `LatentGraph` of hidden dimension 16. Observe that our encoders and decoders are simply standard PyTorch modules.

```
>>> processor = build(MPNN.default_config(latent_dims, latent_dims))
>>> input_graph: Batched[InputGraph, SoftInput] =
 ↪  sampled_batch.hard_to_soft_input()
>>> latent_input: LatentGraph = encoder(input_graph)
>>> latent_output: LatentGraph = processor(latent_input)
>>> output_graph: Batched[OutputGraph, SoftOutput] = decoder(latent_output)
```

Figure 3.9: Generating a *message-passing neural network* (MPNN) [Gilmer et al., 2017] processor, mapping from a latent graph with dimensions `latent_dims` to a latent graph with dimensions `latent_dims`, and building a simple EPD pipeline. Note that type annotations are added for clarity only, and aren't required when using the library.

between two instances $y, \hat{y}$ of the same feature type with the same structure, the MDARF will recurse through these structures, taking the (elementwise) metric between each matching pair of primitive features, and returning a masked tensor tree with these metrics as its leaves (as in Figure 3.10).

§3.3.6 Example: using the MDARF beyond algorithmic reasoning

Before we conclude our discussion on the MDARF proper, we note that, although (for reasons of time and scoping) we only implemented support for constructs relevant to NAR, the MDARF is easily extensible for use beyond this context. As an example, we illustrate (in Figure 3.11) how one might use the MDARF to solve the multimodal text / image classification problem of *predicting various properties of a piece of clothing (e.g. its style and market price) from an image and a product description.*

```
>>> output_graph: Batched[OutputGraph, SoftOutput] = ... # batch of 10 graphs
>>> ground_truth: Batched[OutputGraph, Hard] = ... # batch of 10 graphs
>>> tree_losses = output_pred.loss(ground_truth_output)
>>> tree_losses.shape
MaskedTensorTree(data={
    'nodes': {'most_common_colour': Size((10,)), 'matching_weight_sum':
 ↪  Size((10,))},
    'edges': None, 'features': None, 'pointers': None
})
>>> (tree_losses.reduce() # folds over tree (averaging by default)
... .avg().data)          # averages over batch dimension
tensor(0.4515, grad_fn=<WhereBackward0>)
```

Figure 3.10: Computing the loss between predicted and ground-truth output graphs in our toy problem. Notice that, by default, our loss function returns a tree of per-batch-element, per-feature losses, respecting the structure of our `OutputGraph`.

```
class Style(Enum):
    casual = 0; formal = 1; sportswear = 2; ...

@build_enc_dec(LatentSingle)
class ProductData(BatchableDataclass):
    image: Image; description: Text

@build_enc_dec(LatentSingle)
class ProductProperties(BatchableDataclass):
    style: Categorical[Style]; price: Scalar

latent_dims = LatentSingleDims(hidden=128)
encoder = fdl.build(default_encoder_config(ProductData, latent_dims))
decoder = fdl.build(default_decoder_config(latent_dims, ProductProperties))

input_data: Batched[ProductData, SoftInput] = ...
latent: LatentSingle = encoder(input_data)
output_data: Batched[ProductProperties, SoftOutput] = decoder(latent)
```

Figure 3.11: A sample implementation of an inference pipeline for the clothing classification problem in the MDARF. We assume we have the relevant input data batched in the correct format, and stored in `input_data`. We assume our MDARF is extended with support for a `Sequence[T]` and `LatentSequence` type (with a processor `LatentSequence -> LatentSingle` such as an encoder-only transformer), a type alias `Text := Sequence[Categorical]`, and an `Image` type (with an encoder of type `Image -> LatentSingle`, such as a CNN).

§3.4 An encode-process-decode pipeline for step-level NAPs

So, up until now, everything has been very general. In fact, what we've really built is *a compositional, typed, declarative way to automatically map complex datatypes to and from latent spaces*, which can help us implement both step-level and algorithm-level NAPs. But recall from Section 2.2.1 that, for our investigation, we want to use our MDARF to build an end-to-end **encode-process-decode (EPD) pipeline** for training step-level NAPs in the style of the

CLRSF, which we can then use in our benchmark environments.

Now, while a number of recent works [Veličković et al., 2022, Deac et al., 2021, Numeroso et al., 2023] explore EPD pipelines in the context of NAR, the structure of these pipelines can vary substantially between implementations – and various preliminary experiments indicate that even slight differences in EPD pipeline structure can have substantial effects on performance. As such, in Section 3.4.1, we attempt to derive the structure of an 'ideal' EPD pipeline from first principles using the paradigm of *representations-as-types* [Olah, 2015] – and, in Section 3.4.2, we discuss how our theoretically-motivated design avoids a *scalar bottleneck* present in the EPD pipeline of the CLRSF. Finally, in Section 3.4.3, we outline the implementation of this design as a wrapper on top of our MDARF core.

§3.4.1 Deriving an EPD pipeline

A general model for algorithms. To build an EPD pipeline, we must first come up with a general way to decompose algorithms into individual steps. For maximal flexibility (e.g. to support *online algorithms* [Veličković et al., 2020a]), we consider algorithms to be functions A mapping lists of input $[x_1, ..., x_n] : List[I]$ to lists of output $[y_1, ..., y_n] : List[O]$, such that, at every time-step t, they ingest a new input x_t (which may be $\emptyset$), and return a new output y_t (which may be $\emptyset$).

Observe that these algorithms `a :: [Input] -> [Output]` can be decomposed in terms of a *program state space* `State`, an *executor* `e :: (State, Input) -> State`, an *initial program state* `s0 :: State`, and an *output projection function* `pi_y :: State -> Output`, as

```haskell
scanl :: ((a, b) -> a) -> a -> [b] -> a
scanl f s [x1, x2, ...] = [s, f(s,x1), f(f(s,x1),x2), ...]

a' :: [Input] -> [Output]
a' xs = map pi_y states
    where states = scanl e s0 xs
```

In other words, given a sequence of inputs $[x_1, ..., x_n]$, we *scan* our executor down this list of inputs, yielding a sequence $[s_1, ..., s_n]$ of per-step *program states*, from which we can project per-step outputs. We refer to this representation as the **stepwise form** of A, and illustrate its data-flow in Figure 3.12.

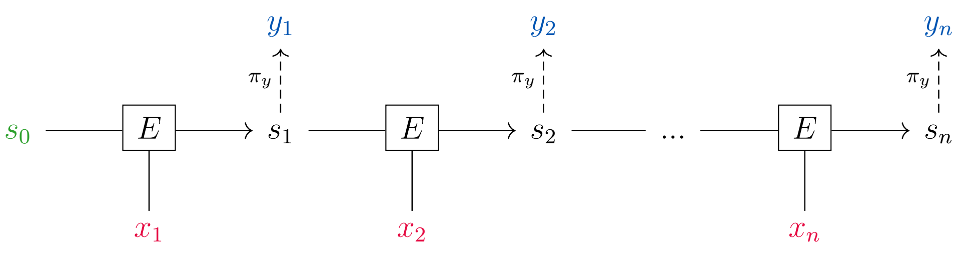


Figure 3.12: A data-flow diagram for the *scanning decomposition* of A, applied to input $[x_1, ..., x_n]$ with initial state s_0, and returning output $[y_1, ..., y_n]$.

Building our NAP. So, given a stepwise-decomposed algorithm `a :: [Input] -> [Output]`, how do we *neuralise* it (i.e. turn it into a step-level NAP)? Now, rather than neuralising `a` directly, we will instead neuralise the following function mapping our list of inputs to our list of program states:

```
a_states :: [Input] -> [State]
a_states xs = scanl e s0 xs
```

Observe that, as π_y 'does no work' (i.e. only projects the output value from each state), the NAP obtained by neuralising `a_states` is at least as powerful as that obtained by neuralising `a'`.

So, drawing on the idea of *representations-as-types* (Appendix A), the natural way to neuralise `a_states` is to simply *lift* `e` *and* `s0` *to latent space.* Specifically, given latent spaces `InputL` and `StateL`, a learnable parameter `s0_hat :: Learnable StateL`, and some learnable processor function `p :: Learnable ((StateL, InputL) -> StateL)`, we have

```
nap :: Learnable ([InputL] -> [StateL])
nap xs_hat = scanl p s0_hat xs_hat
```

Training our NAP (in principle). Now, how do we train this NAP? Recall from Section 2.1.1 that NAPs $\hat{A} : \mathcal{X} \to \mathcal{Y}$ imitating $A : X \to Y$ should be trained such that $dec_Y \circ \hat{A} \circ enc_X \approx A$ (i.e. we should be able to recover $A(x) \in Y$ from $\hat{A}(enc(x)) \in \mathcal{Y}$). Thus, the most obvious way to train `nap :: Learnable ([InputL] -> [StateL])` to imitate `a' :: [Input] -> [State]` would simply be to ensure the following diagram commutes (for suitable encoders and decoders `enc :: Input -> InputL` and `dec :: StateL -> State`):

$$\begin{tikzcd}
List[\mathcal{I}] \arrow[r, "\hat{A}"] & List[\mathcal{S}] \arrow[d, "\text{map } dec"] \\
List[I] \arrow[u, "\text{map } enc"] \arrow[r, "A"] & List[S]
\end{tikzcd}$$

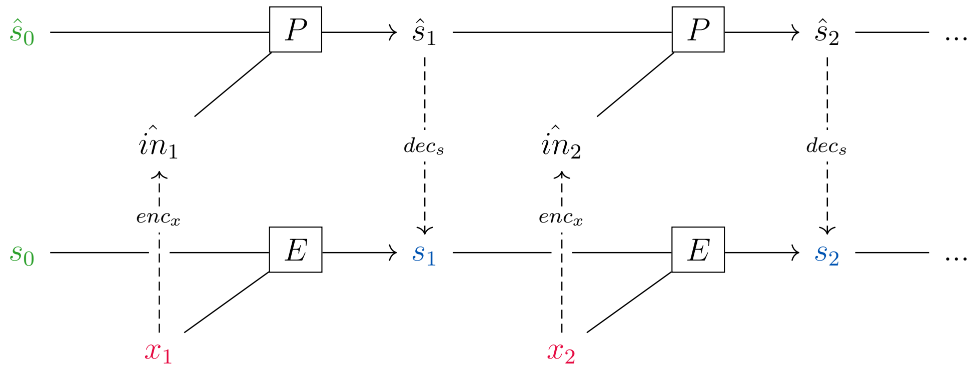
In practice, this property can be enforced as illustrated in Figure 3.13.

Training our NAP (in practice).

Now, observe that the training conditions above enforce that our complete program states s_t can be recovered in their entirety from our latent representations $\hat{s}_t$. As these states can be quite complex, in practice we enforce a slightly weaker condition: we simply ensure that we can extract some *meaningful subset of our program state* from our latent representations.

For our algorithm A in stepwise-decomposed form, we define a projection function $\pi_h : S \to H$, that for any state s_t, extracts a **per-step algorithmic hint** – i.e. some subset of s_t that we wish to be able to recover from our latent states $\hat{s}_t$. And, given these hints $h_t = \pi_h(s_t)$, we ensure that, from any latent state $\hat{s}_t$, we can recover both our per-step hint h_t and our per-step output o_t. More precisely, we ensure the following diagram commutes, for decoders $dec_h : \mathcal{S} \to H$ and $dec_y : \mathcal{S} \to O$:

$$\begin{tikzcd}
List[\mathcal{I}] \arrow[rr, "\hat{A}"] & & List[\mathcal{S}] \arrow[dl, "\text{map } dec_h"'] \arrow[dr, "\text{map } dec_y"] & \\
& List[H] & & List[O] \\
List[I] \arrow[uu, "\text{map } enc"] \arrow[rr, "A"] & & List[S] \arrow[ul, "\text{map } \pi_h"] \arrow[ur, "\text{map } \pi_y"'] &
\end{tikzcd}$$



```
wrapped_nap :: Learnable ([Input] -> [State])
wrapped_nap inputs = map dec_s (nap (map enc_in xs))

forall xs :: [Input]. algorithm inputs == wrapped_nap inputs
```

Figure 3.13: A data-flow diagram illustrating the 'ideal' training procedure for our NAP, given an input $[x_1, ..., x_n]$ and initial states $s_0 \in S, \hat{s}_0 \in \mathcal{S}$. We train our network on sequences of random inputs $[x_1, ..., x_n] \in List[I]$ such that 'all ways of computing blue variables yield equal results' – in other words, for all blue variables, we minimise (some appropriate) loss between the values on each of the incoming arrows.

So, to train our NAPs, we enforce this diagram in the obvious way, but with one small modification: in order to avoid our processor having to represent data in its latent state that are already carried by the input, we adapt dec_h and dec_y to map not from $\mathcal{S}$, but rather from $\mathcal{S} \times \mathcal{I}$. We present the full training data-flow for this modified EPD pipeline in Figure 3.14.

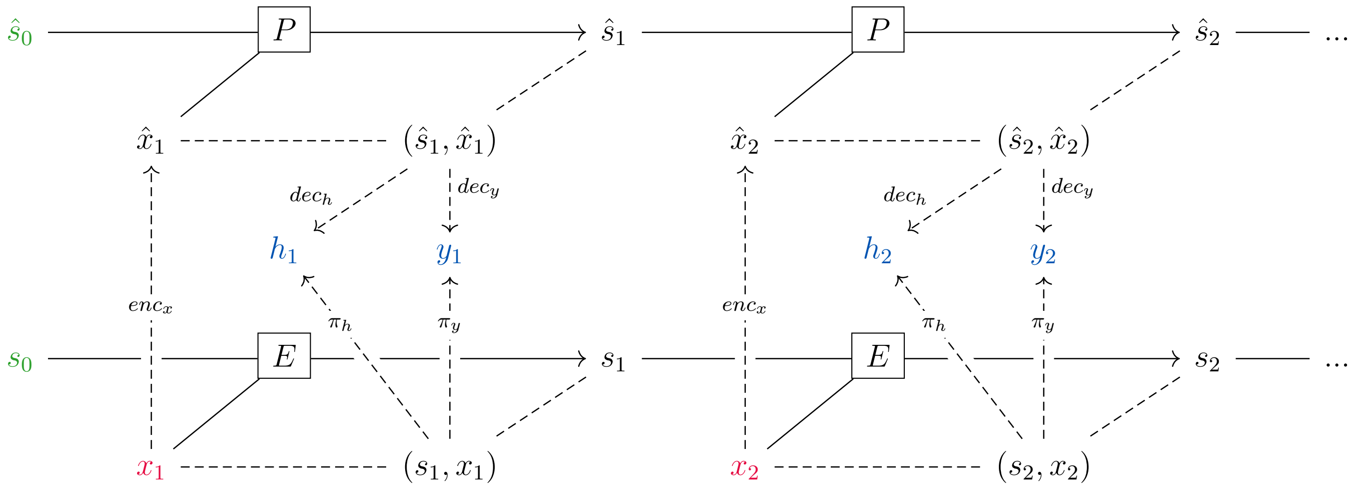
§3.4.2 Comparing our pipeline to the CLRSF: avoiding the scalar bottleneck

So, now that we've derived a principled EPD pipeline, let's see how it compares to what's used in practice: specifically, the CLRSF pipeline, whose dataflow we sketch in Figure 3.15.

Now, the main difference between these pipelines is that, unlike the EPD, the CLRSF's processor P takes at every step not only the previous latent state $\hat{s}_{t-1}$ and the current encoded input $\hat{x}_t$, but also a *re-encoded version* of the previous hint $h_{t-1} = dec_h(\hat{s}_{t-1}, \hat{x}_{t-1})$. In other words, instead of passing our latent state directly to the processor, we first *factor it through a scalar bottleneck*. While re-encoding hints in this way may provide a stronger inductive bias towards learning relevant algorithmic representations, we argue that *by not including this scalar bottleneck (SB), our EPD pipeline is not only more suitable for NAR, but is likely to perform better in the abstract domain*:

The SB defeats the (claimed) purpose of NAR. Recall from Section 2.1.2 that *the fundamental claim of NAR is that NAPs outperform sDABs by alleviating their scalar bottleneck*. As such, if this claim holds, this scalar bottleneck should reduce the maximum performance of our NAPs.[4]

[4]One may argue that this scalar bottleneck shouldn't be an issue, as we pass it in alongside a 'residual stream' of latent states. But, especially given that the CLRSF trains by supervising on hints h_i, our NAP will likely learn to depend on the re-encoded versions of these hints alongside (or even in preference to) the latent states flowing down the residual stream – and, given that (at least in recent work) NAPs are deployed in real-world problems with *frozen weights*, this effect amounts to a scalar bottleneck in the NAP.



```haskell
wrapped_algorithm :: [Input] -> ([Hint], [Output])
wrapped_algorithm xs =
    (map pi_h (zip xs states), map pi_y (zip xs states))
    where states = scanl e s0 xs

wrapped_nap :: Learnable ([Input] -> ([Hint], [Output]))
wrapped_nap xs =
    (map dec_h (zip xs_hat states_hat), map dec_y (zip xs_hat states_hat))
    where xs_hat = map enc_x xs
          states_hat = scanl p s0_hat xs_hat

forall xs :: [Input]. wrapped_algorithm xs == wrapped_nap xs
```

Figure 3.14: A data-flow diagram illustrating the training procedure for our NAP, given a (randomly sampled) input $[x_1, ..., x_n]$ and initial states $s_0 \in S, \hat{s}_0 \in \mathcal{S}$. We refer to `wrapped_algorithm` and `wrapped_nap` as the **input-hint-output (IHO) forms** of A and $\hat{A}$ respectively.

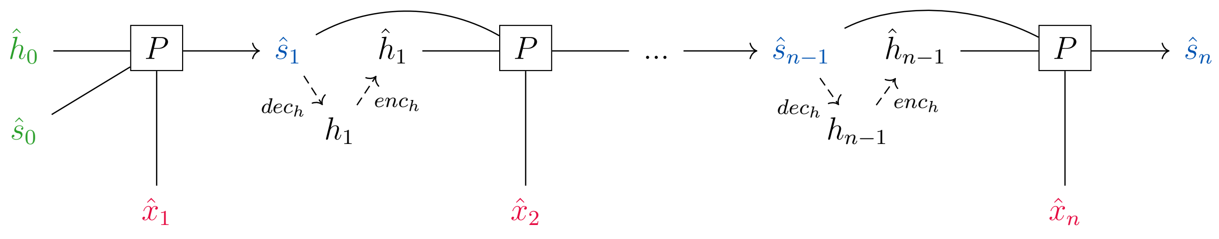


Figure 3.15: A sketch of the data-flow for the 'latent space level' of the CLRSF pipeline. Given a series of (encoded) inputs $[\hat{x}_1, ..., \hat{x}_n] \in List[\mathcal{I}]$ and initial states $\hat{s}_0 \in \mathcal{S}, \hat{h}_0 \in \mathcal{H}$, it emits a series of latents $[\hat{s}_1, ..., \hat{s}_n] \in \mathcal{S}$.

The SB leads to numerical instability during training. Furthermore, recent work [Ibarz et al., 2022] notes that one of the main performance bottlenecks of the CLRSF pipeline is the *gradient instability* brought about by factoring the latent state through a scalar space every timestep.

The SB likely harms OOD performance in the abstract domain. Finally, as per Section 2.1.2, introducing a scalar bottleneck could worsen performance out-of-distribution (in particular, over long trajectories), by amplifying small mispredictions over the course of the trajectory.

§3.4.3 Implementing the EPD pipeline

Given that we've designed our EPD pipeline, let's consider how we implement it in our MDARF.We give a high-level overview of the training process in Figure 3.16, and outline its steps in Appendix C. For illustration, in Appendix D, we demonstrate a complete training pipeline for a Bellman-Ford NAP in just 38 lines of vanilla PyTorch (excluding imports).

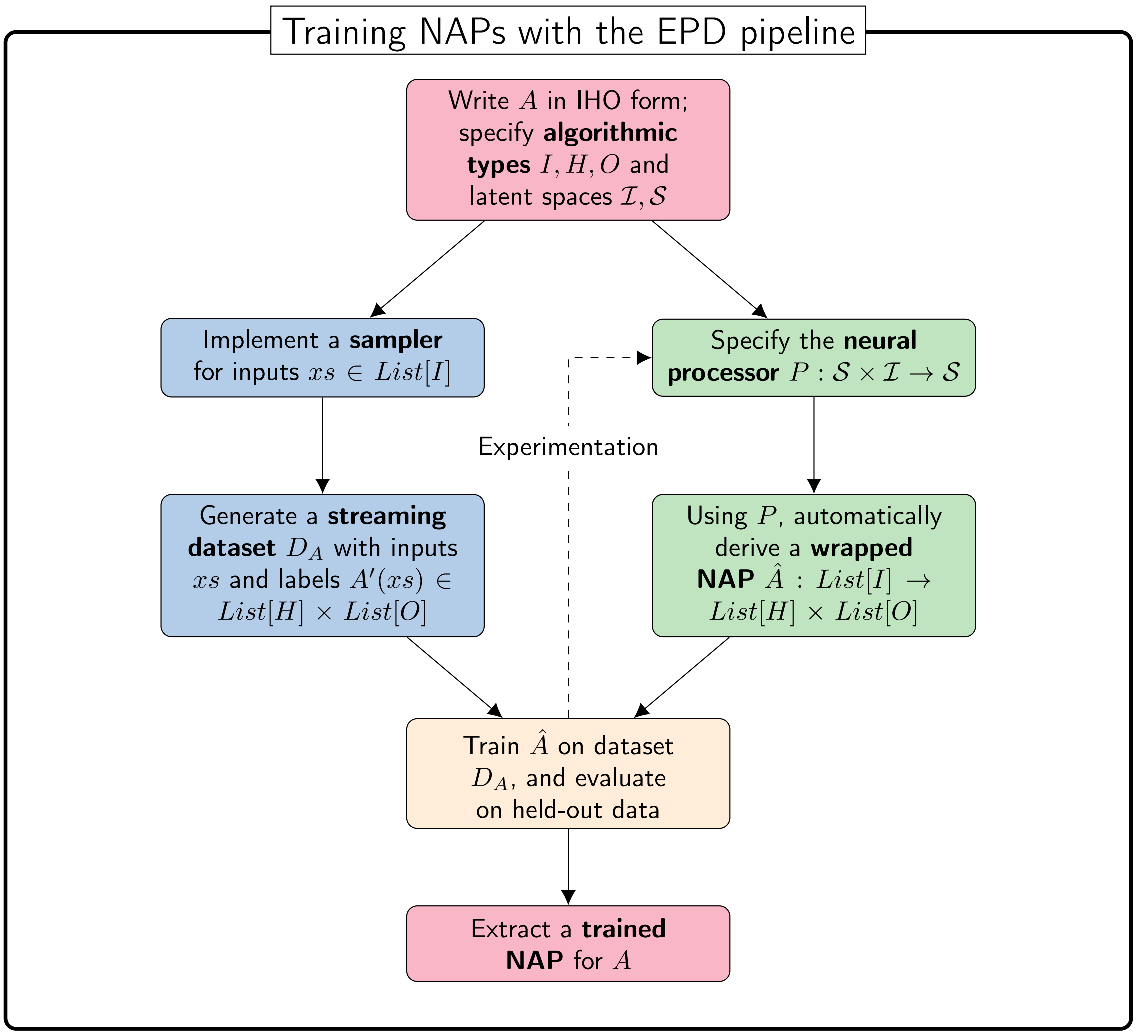

Training NAPs with the EPD pipeline
Write A in IHO form; specify **algorithmic types** I, H, O and latent spaces $\mathcal{I}, \mathcal{S}$
Implement a **sampler** for inputs $xs \in List[I]$
Specify the **neural processor** $P : \mathcal{S} \times \mathcal{I} \to \mathcal{S}$
Experimentation
Generate a **streaming dataset** D_A with inputs xs and labels $A'(xs) \in List[H] \times List[O]$
Using P, automatically derive a **wrapped NAP** $\hat{A} : List[I] \to List[H] \times List[O]$
Train $\hat{A}$ on dataset D_A, and evaluate on held-out data
Extract a **trained NAP** for A

Figure 3.16: A flowchart illustrating the process of training NAPs with the EPD pipeline to imitate some given algorithm, expressed (as in Figure 3.14) as a `wrapped_algorithm` $A : List[I] \to List[H] \times List[O]$.

§3.5 Repository overview

Besides using the libraries mentioned in Section 2.4 (and one exception[5] mentioned in Table 3.4), all source code was written from scratch. Our codebase consists of four Python packages, corresponding to the four main components of the project: the `mdarf` package (Table 3.1) containing the core MDARF and the EPD pipeline, the `training_reasoners` package (Table 3.2) containing a training wrapper for NAPs built with the EPD pipeline, and our benchmark implementations in `xlvin` (Table 3.3) and `warcraft` (Table 3.4). While the final three packages are all independent of each other, they all depend on the `mdarf`.

File / folder	Description	Lines of code
`values/`	A sub-package containing feature types, handling batching, sampling, encoders and decoders.	3281
`latents/`	A sub-package containing latent types.	343
`combiners/`	A sub-package containing combiners for latent types.	96
`processors/`	A sub-package containing processors for latent types.	729
`utils/`	A utility sub-package handling testing, pretty-printing, registries and some auxiliary tensor containers.	464
`epd/`	A sub-package implementing the EPD pipeline, containing trajectory and trace datatypes, the executor module and implementations of various algorithms to train on.	1451

Table 3.1: Top-level directory structure for the `mdarf` package (6364 lines of code)

File / folder	Description	Lines of code
`data.py`	A module containing a PyTorch Lightning wrapper for MDARF feature-type datasets.	100
`model.py`	A module containing a PyTorch Lightning wrapper for the EPD executor network.	179
`main.py`	A module handling a lightweight CLI for the PyTorch Lightning training pipeline.	68
`experiments/`	A folder for tracking experiments (logs, checkpoints, SLURM scripts).	n/a

Table 3.2: Top-level directory structure for the `training_reasoners` package (347 lines of code)

[5]In order to implement XLVIN, we needed to slightly adapt two files within the codebase of Stable Baselines 3 to make them more general. While we attempted to submit these small modifications as a pull request, this was rejected for being too niche — despite another user having submitted a pull request for a functionally equivalent feature a few months ago (which was also rejected for being too niche). As such, we simply modified the files directly, and included the modified versions in our repository.

File / folder	Description	Lines of code
`policy.py`	A module implementing the XLVIN actor-critic network.	185
`transe/`	A sub-package containing the TransE encoder, alongside a utility module for pre-training it (if desired).	205
`utils/`	Various utility modules, including some modified files from Stable Baselines 3.	335
`envs.py`	A module containing configuration parameters for various OpenAI gym environments on which to evaluate XLVIN.	27
`config.py`	A module containing a general configuration dataclass used to recursively construct our XLVIN.	19
`main.py`	A module handling a lightweight CLI for the PyTorch Lightning training pipeline.	128
`experiments/`	A folder for tracking experiments (logs, checkpoints, SLURM scripts).	n/a

Table 3.3: Top-level directory structure for the `xlvin` package (899 lines of code)

File / folder	Description	Lines of code
`data/`	A sub-package handling data processing for the Warcraft-Shortest-Path-Tree dataset.	140
`models/`	A sub-package containing models, feature extractors and algorithmic modules for building Warcraft-Net.	312
`utils/`	A module containing utilities for efficiently processing and computing metrics over sparse graphs (some of which we JIT-compile with `numba`).	603
`config.py`	A module containing a general configuration dataclass used to recursively construct our Warcraft-Net.	101
`train.py`	A module handling a lightweight CLI for the PyTorch Lightning training pipeline.	261
`experiments/`	A folder for tracking experiments (logs, checkpoints, SLURM scripts).	n/a

Table 3.4: Top-level directory structure for the `warcraft` package (1417 lines of code)

4 Evaluation

Now that we've implemented our MDARF, as per our project aims in Section 1.2, we seek both to evaluate its *correctness* by comparing it against state-of-the-art reasoners (Section 4.1), and to evaluate its *utility for reproduction and research* by using it to explore NAPs within the context of algorithmic benchmarks (Sections 4.2–4.3).

§4.1 Evaluating correctness: training NAPs on the CLRS-30 benchmark

Before we explore our benchmarks, we first **evaluate the correctness of our EPD pipeline**, by using it to train GMPNN and Triplet-GMPNN NAPs on a subset of representative algorithms from the CLRSB, and testing whether or not they match the state-of-the-art performance of the CLRSF pipeline [Ibarz et al., 2022] in and out of distribution.

§4.1.1 Experimental details

Algorithms used. As it is out-of-scope to port all 30 CLRSB algorithms to the MDARF, we chose to explore two contrasting algorithms: **Bellman-Ford (BF)**, a simple, continuous algorithm aligned with GNNs [Dudzik and Veličković, 2022], and **Depth-First Search (DFS)**, one of the most challenging algorithms in the CLRSB.

Metrics, hyperparameters and training. For both algorithms tested, we use the evaluation metric of **predecessor pointer accuracy** on out-of-distribution graphs used by the CLRSB, over 5 training runs. Full details of the model, training and evaluation hyperparameters can be found in Appendix F.1.

Hypothesis testing. On the assumption that the predecessor pointer accuracies of randomly initialised models are normally distributed, we use the one-sided **unequal-variances t-test** [Welch, 1947] on sample means to ascertain the statistical significance of any performance differences observed. Furthermore, as we will perform multiple hypothesis tests over the course of this analysis, we use the **Holm-Bonferroni method** [Holm, 1979] to ensure the **family-wise error rate (FWER)** (i.e. the risk of rejecting one or more true null hypotheses) is at most $\alpha = 0.05$.[1] Specifically, for all m null hypotheses $H_1, \ldots, H_m$ we test, we sort their p-values $P_1, \ldots, P_m$ from lowest to highest, and reject the longest prefix of null hypotheses $H_1, \ldots, H_k$ satisfying (for each $1 \leq i \leq k$) $P_i < \frac{\alpha}{m+1-i}$. As such, in the following discussion, we write, e.g., [$(1) > (3)$ OOD: $i = 1$, $P_i = 6.5 \times 10^{-11} < \frac{0.05}{3}$, ✓] to abbreviate the following: "There is sufficient evidence to suggest that architecture 1 performs better on the CLRSB test set than architecture 3 at FWER $\alpha = 0.05$.

§4.1.2 Results and discussion

Recall that we're interested in *whether our EPD pipeline can match the state-of-the-art performance of the CLRSF pipeline, both in and out of distribution.* We observe from Figure 4.1 that, while processors trained on the EPD pipeline tend to perform worse than equivalent

[1]Note that this method is uniformly more powerful than the Bonferroni correction [Holm, 1979].

Model (BF)		CLRS val pi	std	CLRS test pi	std
GMPNN	1	0.9934	0.0056	**0.9921**	0.0013
Triplet-GMPNN	2	0.9926	0.0072	**0.9896**	0.0032
[GNAL] Triplet-GMPNN	3	(not reported)		0.9739	0.0019
[CLRS] MPNN	4	**0.9948**	0.0005	0.9299	0.0034
[CLRS] PGN	5	**0.9935**	0.0005	0.9201	0.0028
[CLRS] Memnet	6	0.6875	0.0042	0.4004	0.0146

Model (DFS)		CLRS val pi	std	CLRS test pi	std
Triplet-GMPNN	7	0.9039	0.0879	**0.5955**	0.1015
[GNAL] Triplet-GMPNN	8	(not reported)		**0.4779**	0.0419
GMPNN	9	0.3773	0.0596	0.2563	0.0158
[CLRS] Memnet	10	0.4772	0.0045	0.1336	0.0161
[CLRS] PGN	11	**1.0000**	0.0000	0.0871	0.0024
[CLRS] MPNN	12	**1.0000**	0.0000	0.0654	0.0051

Figure 4.1: Results of processors trained with the EPD pipeline on algorithms from the CLRSB, in comparison with those analysed in [Ibarz et al., 2022]. Best results in bold-face, second-best in underline. Standard deviations were taken over 5 runs for EPD experiments, 3 runs for CLRS experiments [Veličković et al., 2022] and 10 runs for GNAL experiments [Ibarz et al., 2022]. Note that Ibarz et al. [2022] did not report validation results for GNAL processors.

processors trained on the CLRSF pipeline in-distribution, they perform significantly better out-of-distribution.

In the case of BF, while both EPD processors (1, 2) and the top CLRS processors (4, 5) achieve near-perfect performance (> 0.99) in-distribution, our best EPD processor (1) is the only processor to do so out-of-distribution, and indeed significantly outperforms the previous SOTA out-of-distribution $[(1) > (3)$ OOD: $i = 1$, $P_i = 6.5 \times 10^{-11} < \frac{0.05}{2}$, ✓$]$.

And in the case of DFS, we observe that the CLRSF pipeline leads to overfitting: while our best EPD processor (7) performs worse than CLRSF processors (11, 12) in-distribution, it significantly outperforms the previous SOTA out-of-distribution, albeit with a much higher p-value than for BF $[(7) > (8)$ OOD: $i = 2$, $P_i = 0.029 < \frac{0.05}{1}$, ✓$]$. Observe also that, even though our GMPNN (9) performs worse out-of-distribution than the SOTA, it significantly outperforms its closest architectural equivalent from the CLRSB $[(9) > (12)$ OOD: $i = 1$, $P_i = 6.5 \times 10^{-7} < \frac{0.05}{1}$, ✓$]$.

Summary. As **processors trained with our EPD pipeline beat the state-of-the-art on the CLRS-30 benchmark** on both BF and DFS, we can use the EPD pipeline with confidence when generating NAPs for our benchmark environments.[2]

§4.2 Evaluating utility for reproduction: exploring the VI-Implicit-Planner benchmark

So, now we've seen that our framework for building NAPs beats the state-of-the-art in the abstract domain, let's explore how our NAPs perform in the real-world domain (and *evaluate the utility of our laboratory* for reproducing existing results in NAR) by deploying them in our first benchmark environment: **VI-IMPLICIT-PLANNER**.

Benchmark overview. This benchmark is based on the observation that the problem of training deep reinforcement learning (RL) agents has an *algorithmic prior*: specifically, if our agent could learn to build a search tree representing how the state of its environment would change if it were to take various actions, then it could compute the optimal next action by running the planning algorithm of **value iteration (VI)** over this tree (see Appendix B for more details). Such agents, **eXecuted Latent Value Iteration Networks (XLVINs)** [Deac et al., 2021], explicitly generate a search tree over possible sequences of actions (whose nodes are latent representations of 'possible next states'), and use some choice of *algorithmic module* to perform

[2]Note that this result also provides weak evidence towards the claim in Section 3.4.2 that the scalar bottleneck harms OOD performance in the abstract domain. However, as the EPD pipeline has a number of other improvements over the CLRSF, there are too many confounding factors to make this claim with confidence; we leave more precise ablations for further work.

VI over this tree to compute the optimal next action. (For implementation details, including **various improvements** we made to the architecture of [Deac et al., 2021], see Appendix E.1.)

Summary of results. In the following section, we explore VI-IMPLICIT-PLANNER in the context of OpenAI's *Acrobot* environment, and demonstrate that:

Our laboratory lets us easily implement performant NAPs. Using the MDARF, we successfully trained an NAP to imitate deterministic value iteration (see Appendix F.2.1 for details), obtaining *near-perfect performance out-of-distribution.*

Our laboratory supports high-quality research. Using *robust statistical methodology*, we demonstrated that *neither our XLVIN implementation nor that of [He, 2022] substantially outperforms the PPO baseline.*

Our laboratory validates existing results. But we then saw that, if we *alleviate the compute bottleneck* in the XLVIN training regime, *our laboratory validates the performance claims of XLVIN*, with XLVIN substantially outperforming the PPO baseline on the Acrobot environment.

Our laboratory leads us to new research insights. Finally, the rich configurability of our laboratory allowed us to run many ablation tests; these revealed that, for the Acrobot environment, *the performance improvement of XLVIN over the PPO baseline is unlikely to be due to NAR.*

§4.2.1 Experimental details

Environment. In this analysis, we study the VI-IMPLICIT-PLANNER benchmark within the *Acrobot* discrete-control environment. This *sparse-reward* environment is known to be challenging for policy-gradient algorithms, and was one of the 'classical control' environments explored in [Deac et al., 2021] and [He, 2022].

Executor. To obtain a VI NAP, we trained a GMPNN processor on deterministic value iteration, which learned to predict optimal next states with near-perfect accuracy out-of-distribution (see Appendix F.2.1 for results).

Architecture, hyperparameters, training and metrics. We use the architecture and hyperparameters reported by He [2022], with minor modifications. We assessed every model over 20 training runs k, each initialised with a different random seed, reporting for each run the **per-run maximum average reward**. For full details on architecture, hyperparameters and metrics, see Appendix F.2.2.

Performance evaluation. As, in the few-run deep RL regime, point estimates of aggregate metrics are typically dominated by statistical uncertainty,[3] we follow the recommendations of the *RLiable* framework [Agarwal et al., 2021] and instead report *interval estimates* of performance. Specifically, for each set of runs, we report **95% confidence intervals (CIs)** for metrics such as mean, median and inter-quartile mean, computed via *bootstrap resampling* [Efron, 1979]. And, in order to compare models X and Y, we estimate 95% confidence intervals for the **probability of improvement (PoI)** of X over Y (in terms of per-run maximum average reward) via bootstrap resampling [Agarwal et al., 2021], by sampling from the empirical distributions of X and Y, and computing the U-statistic from the *Mann-Whitney U test* [Mann and Whitney, 1947] over these samples.

[3]Indeed, only reporting point estimates has historically led the field to erroneously conclude which methods are state-of-the-art [Lin et al., 2021, Reimers and Gurevych, 2017]

§4.2.2 Results and discussion

XLVIN is difficult to reproduce under its original conditions. We first observe that the results of XLVIN, as reported by Deac et al. [2021], are very difficult to reproduce: not only is XLVIN a *very complex architecture* with many hyperparameters and moving parts, but it is also evaluated in *extremely noisy* environments. Furthermore, claims of performance improvement are made by comparing *unreliable point estimates* of average reward.

As comparing these estimates without considering their variance can lead to misleading results [Agarwal et al., 2021], we instead try plotting 95% CIs for the probability of improvement of XLVIN over the PPO baseline (Figure 4.2), in both our codebase and the codebase of He [2022]. Observe that, while experiments suggest the PPO baseline may have a slightly higher PoI over NAP, the difference here is marginal: indeed, the 95% confidence interval includes 0.5, so **in both the MDARF and the codebase of He [2022], there is insufficient evidence that XLVIN performs better than PPO (or vice versa)** in the Acrobot environment.

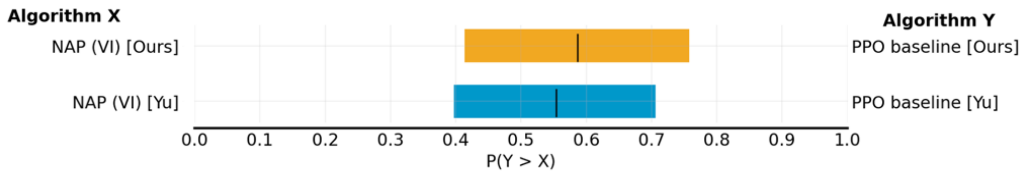


Figure 4.2: Probabilities of improvement (95% CI) of XLVIN over the PPO baseline, in both the MDARF and He [2022], trained under the 1-epoch regime.

Relaxing the compute bottleneck lets us validate the claims of XLVIN. However, in both of the standard implementations, XLVIN is trained under only one PPO epoch per minibatch. This may cause a confounding factor of a *compute bottleneck*. To alleviate it, we repeat our experiments from above on both the MDARF and the codebase of [He, 2022], this time training for 10 PPO epochs per minibatch (Figure 4.3).

Observe from Figure 4.4 that, in this regime, **our laboratory validates the claims of XLVIN**: in the MDARF we see that XLVIN substantially outperforms the PPO baseline (with a PoI of $\tilde{0}.8$, whose 95% CI does not contain 0.5).

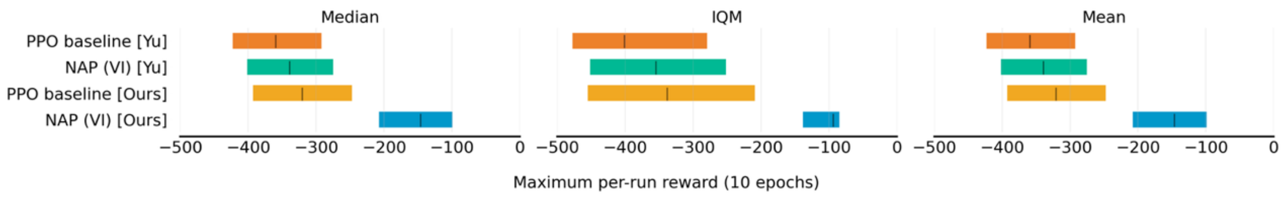


Figure 4.3: Aggregate metrics (95% CI) for the PPO baseline and XLVIN architectures across the MDARF and the codebase of [He, 2022], trained under the 10-epoch regime.

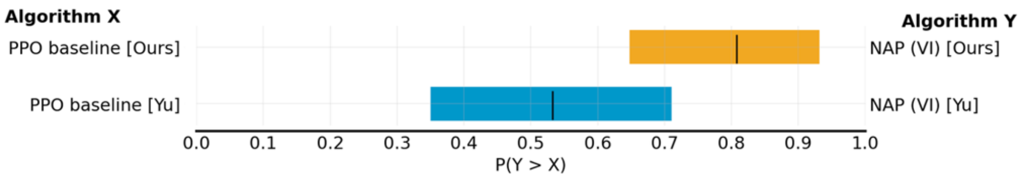


Figure 4.4: Probabilities of improvement (95% CI) of XLVIN over the PPO baseline, in both the MDARF and HF and the codebase of [He, 2022], trained under the 10-epoch regime.

Moreover, in Figure 4.5, we observe that **our laboratory achieves stronger results than [He, 2022] across the board**: while our MDARF PPO baseline is marginally more performant than that of He [2022]'s codebase, we observe that our XLVIN is substantially more performant (with a PoI of $\tilde{0}.87$, whose 95% CI does not contain 0.5).[4]

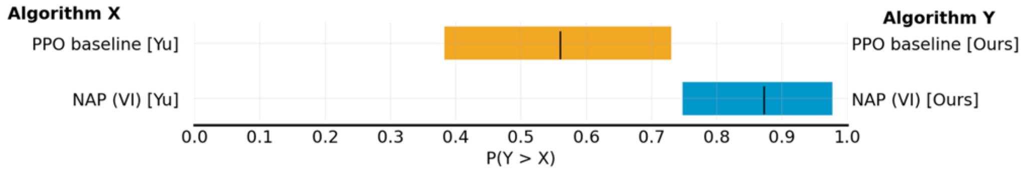


Figure 4.5: Probabilities of improvement (95% CI) of architectures from the MDARF over those from the HF, trained under the 10-epoch regime.

Ablating the NAP doesn't significantly affect the performance of XLVIN. Now, before we can use VI-Implicit-Planner as a benchmark for NAR, we must first ensure that the performance improvements we observe are actually due to its use of a NAP. On close inspection, we noticed that XLVIN has a complexity advantage over its PPO baseline: indeed, not only does it use an NAP, but it also has many more layers than the baseline. Therefore, it's unclear if the performance improvements we observe can be attributed to the algorithmic inductive bias of the NAP, or merely explained by the higher capacity of XLVIN compared to the baseline. As such, to isolate the effect of the NAP, we performed an experiment in our codebase comparing XLVIN with the 'official' PPO baseline, a version of XLVIN ablating only the NAP (the **no-NAP XLVIN**), and and the default policy network from *Stable Baselines 3*.[5]

We illustrate the results of this experiment in Figure 4.6. Observe from Figure 4.7 that, while our XLVIN outperforms the PPO baseline, **there is very little evidence to suggest that ablating the NAP reduces XLVIN performance**, with XLVIN exhibiting a probability of improvement over the ablated model of only 0.510. We also find that **the MLP baseline from Stable Baselines 3 substantially outperforms our NAP**, with a probability of improvement of 0.728 over XLVIN.

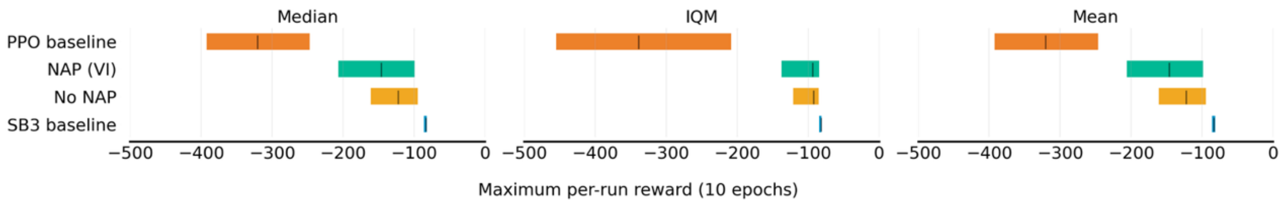


Figure 4.6: Aggregate metrics (95% CI) for the XLVIN, PPO baseline, no-NAP and Stable Baselines 3 (SB3) architectures, trained in the MDARF under the 10-epoch regime.

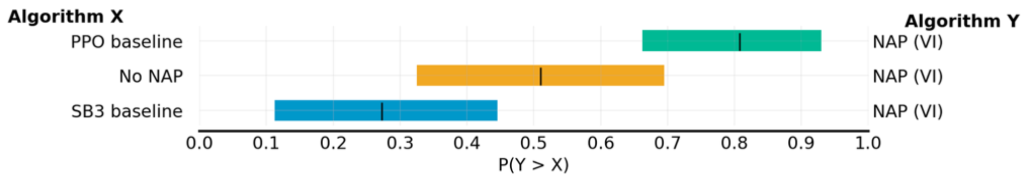


Figure 4.7: Probabilities of improvement (95% CI) of XLVIN over the PPO, no-NAP and SB3 baselines, trained in the MDARF under the 10-epoch regime.

[4]We hypothesise that this result is due to a combination of the improvements made to the XLVIN architecture in Appendix E.1, and our use of an optimised implementation of PPO backed by *Stable Baselines 3* [Hill et al., 2018].

[5]i.e. two three-layer MLPs with tanh activation, mapping from state to policy and state to value respectively

§4.2.3 Conclusions and next steps

So we've seen that, while the exact results of [Deac et al., 2021] are noisy and difficult to reproduce (with either our framework or that of [He, 2022]), a slight modification to the training procedure allowed us to validate their claims – but ablation tests then revealed that **the performance improvement of XLVIN is unlikely to be due to NAR**.

Thus, while we originally planned to use VI-IMPLICIT-PLANNER as an environment in which to compare the performance of NAPs against sDABs, **the findings of this section cast doubt on the suitability of VI-IMPLICIT-PLANNER (at least, when applied to classical control tasks[6]) as a benchmark environment for NAR**: not only is it *extremely noisy* (making it very difficult to distinguish actual performance improvements from random noise), but it is *unclear as to whether using an algorithmic module improves performance within this environment at all.*

§4.3 Evaluating utility for research: exploring the Warcraft-Shortest-Path benchmark

So, having evaluated the *correctness* of our laboratory with respect to the CLRSB, and its *utility for reproduction* in the context of XLVIN, we now evaluate its *utility for research* by using it to **probe the foundations of neural algorithmic reasoning**.

Benchmark overview. As we concluded in the previous section that VI-IMPLICIT-PLANNER is an unsuitable environment for exploring the claims of NAR, we perform this analysis in the context of our second benchmark environment, **WARCRAFT-SHORTEST-PATH-TREE**: a synthetic supervised-learning environment with a much clearer algorithmic prior.

This benchmark, inspired by one of the most popular environments for testing sDABs [Vlastelica et al., 2019], involves comparing algorithmic modules in the context of a neural network trained to find the *shortest-path tree* from a $k \times k$ Warcraft terrain map (as described in Figure 4.8). More precisely, this network uses the first five layers of ResNet-18 [He et al., 2015] to extract a $k \times k$ grid of latent features, runs a graph-based sDAB or NAP over a grid graph constructed from these features, and applies the pointer decoder from [Veličković et al., 2022] in order to extract the predecessor node for each cell. (For implementation details, see Appendix E.2.) Note that, while we can in principle use any sDAB or NAP with an algorithmic inductive bias towards a single-source shortest path algorithm, for simplicity (and following [Petersen et al., 2021]) we choose to explore sDABs and NAPs for **Bellman-Ford**.

We explore two different variants of this problem: the simpler **optimal** variant, where we train our model to predict a distribution equally weighted over all optimal shortest-path predecessors, and the more complex **tie-breaking** variant, where we train our model to break ties between optimal predecessors in a deterministic way. (For more details, see Appendix F.3.1.)

Summary of results. Through our analysis of WARCRAFT-SHORTEST-PATH-TREE, we showcase our laboratory's **utility for research**, by demonstrating that:

Our laboratory lets us systematically compare algorithmic modules.
Using our laboratory, we performed a systematic comparison of algorithmic modules, yielding *results that deviate from established wisdom in NAR*. Specifically, we find that (*contra* Veličković and Blundell [2021]) *NAPs do not substantially outperform sDABs* in any environment tested.

[6]Note that, while it may be the case that a meaningful performance improvement can be observed when the XLVIN architecture is deployed in richer environments (e.g. Atari games), we were not able to explore these due to time and resource constraints.

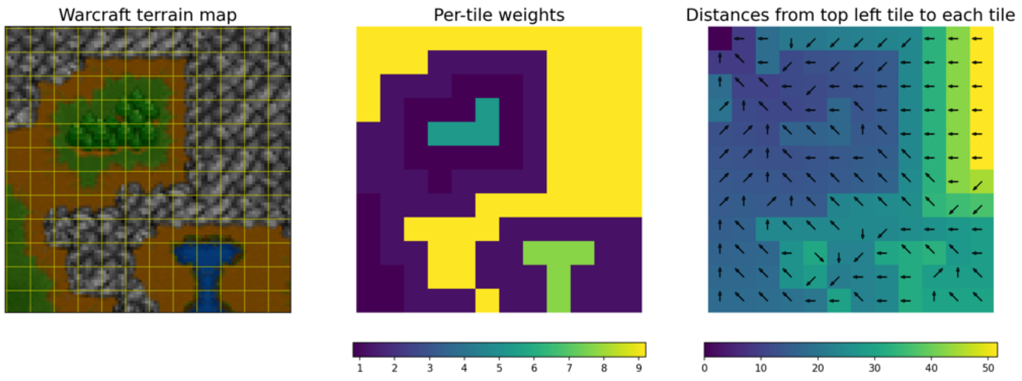


Figure 4.8: An illustration of an example *Warcraft II* terrain map [Guyomarch, 2017], alongside the cost to traverse each tile in its underlying $k \times k$ grid, and the length of the shortest path from the top left tile to each tile in the grid. Our neural network takes a terrain map as input, and must return a $k \times k$ grid of categorical variables, where each categorical variable at position (i, j) indicates the direction of the *shortest-path predecessor* for tile (i, j) (indicated by an arrow).

Our laboratory lets us perform ablation tests that challenge established wisdom in NAR. Counter to the established wisdom in NAR [Deac et al., 2021] that NAPs should be frozen to avoid catastrophic forgetting, we find that *unfreezing NAPs substantially improves their performance*. However, we find that *the performance of these unfrozen NAPs is matched or beaten* by *structurally-aligned neural networks* (SNNs) (Section 2.1.1) – in this case, a simple randomly-initialised GMPNN. These results suggest that the *scalar bottleneck* of sDABs afflicts NAPs as well, but is overcome by SNNs.

Our laboratory lets us prototype a more principled DAB, beating all other models tested and solving an open problem in the literature.
To understand why we observed these results, we explored the effect of *parallelising our sDAB*, and found that the resulting parallel DAB (pDAB) *outperformed all other models tested*. Indeed, through this very simple trick, we obtained an algorithmic module that preserves both the *empirical performance* of SNNs and the *efficiency and correctness guarantees* of sDABs, a goal previously thought to be *'very tricky' to achieve*, and marked by Cappart et al. [2021] as a *'potentially exciting area for future work'*.

Our laboratory helps us validate hypotheses about the nature of the scalar bottleneck. Based on these results, we validated the hypothesis that *the "scalar bottleneck" of sDABs is not a dimensional bottleneck, but rather an ensembling bottleneck* – in other words, that it is not simply the *high dimensionality* of SNNs that allows them to outperform sDABs, but instead their ability to *learn to perform multiple instances of the algorithm in parallel*.

§4.3.1 Experimental details

Metrics. We assessed each model on either **exact tree-accuracy** (i.e. the % of grids with all predecessors correctly predicted) or **optimal tree-accuracy** (i.e. the % of grids for which all predicted predecessor distributions in that grid have an optimal pointer as their mode) as appropriate.

Hyperparameters and training. For each algorithmic module tested, across each problem variant, we performed 5 training runs with different seeds. For each run, we trained on maps of size 12×12, and periodically evaluated them on maps of size 18×18. We reported the results of the highest-performing checkpoint of each run on the test sets of [Vlastelica et al., 2019], assessing both in-distribution (map size 12×12) and out-of-distribution (map size 18×18)

performance. For full details of hyperparameters, see Appendix F.3.2.

Performance evaluation. For this experiment, as per Section 4.2, we report model performance through bootstrapped 95% CIs for mean (exact / optimal) tree-accuracy, and we compare models through bootstrapped 95% CIs for probability-of-improvement.[7]

§4.3.2 Comparing algorithmic modules: the unreasonable effectiveness of SNNs

We first perform a systematic comparison of algorithmic modules in the Warcraft environment (comparing the performance of non-algorithmic baselines, NAPs and sDABs), and run ablation tests exploring the effect of both *unfreezing* our NAPs, and of *randomly initialising their weights.*

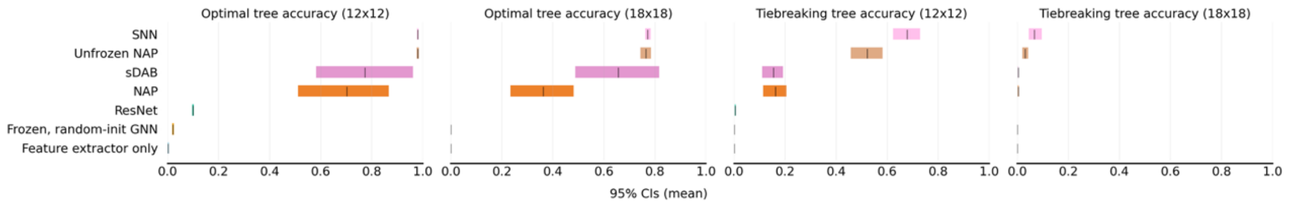


Figure 4.9: 95% CIs for mean tree accuracy, both in-distribution (12x12 grids) and out-of-distribution (18x18 grids), in the optimal and tiebreaking environments.

Algorithmic modules beat non-algorithmic baselines. To check the correctness of our environment, we first compare the performance of sDABs and NAPs against non-algorithmic baselines. We evaluate our algorithmic modules against a ResNet-18 CNN [He et al., 2015], and compare the performance of our algorithmic modules with using only the feature extractor. In order to verify that our NAPs are properly trained, we also evaluate them against frozen, randomly-initialised GNNs.

As per Figure 4.9, we see that *both sDABs and NAPs outperform all three of our baselines.* Specifically, we observe from Figure 4.10 that both sDABs and NAPs substantially outperform ResNet-18 (top left), that NAPs (i.e. frozen, pre-trained GNNs) substantially outperform frozen, randomly-initialised GNNs (top right), and that ablating the executor from WARCRAFT-NET does substantially impair its performance (bottom).

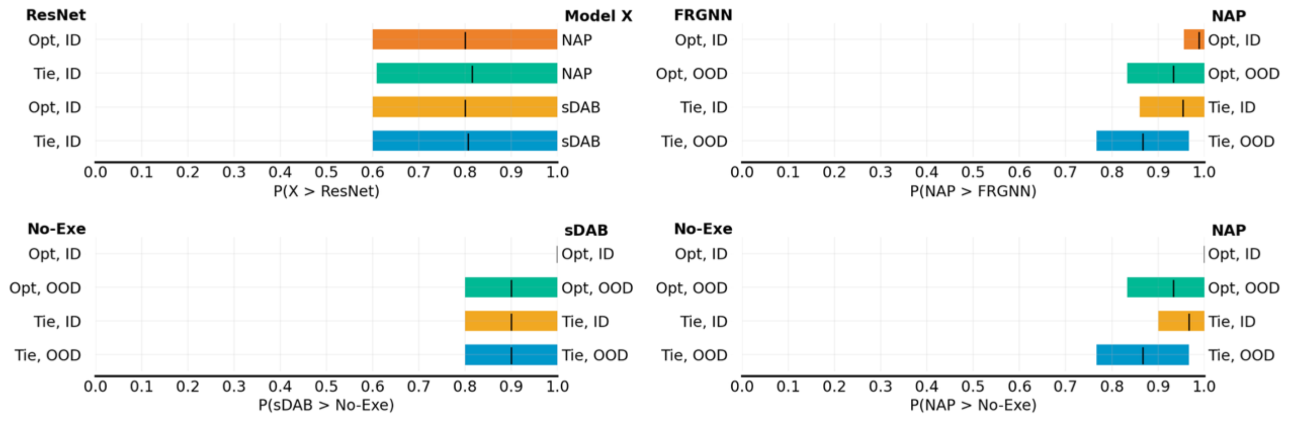


Figure 4.10: 95% CIs for probabilities of improvement in tree-accuracy for sDABs and NAPs over various non-algorithmic baselines, both in-distribution (ID) and out-of-distribution (OOD), in the optimal (Opt) and tie-breaking (Tie) environments.

[7]Observe that, as we collect $n = 15$ runs, we have $\binom{15+15-1}{15} = 7.8 \times 10^7$ possible bootstrap resamples, so we have sufficient data for bootstrap resampling to be meaningful.

NAPs do not outperform sDABs. But, although both NAPs and sDABs outperform algorithmic baselines, we observe from Figure 4.11 that *NAPs do not substantially outperform sDABs*, on either the optimal or tiebreaking environments.

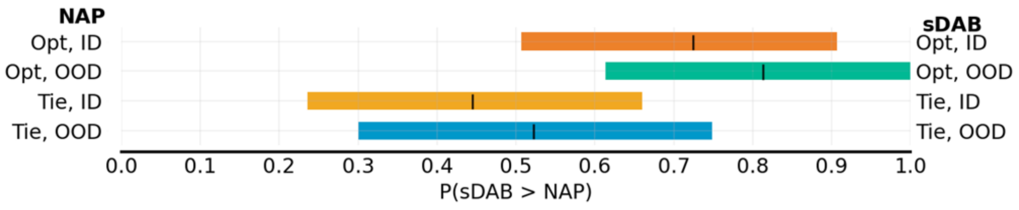


Figure 4.11: 95% CIs for probabilities of improvement in tree-accuracy for sDABs over NAPs, both in-distribution (ID) and out-of-distribution (OOD), in the optimal (Opt) and tie-breaking (Tie) environments.

Unfreezing NAPs improves their performance – but they're still no better than a randomly-initialised GNN. As NAPs are simply frozen GNNs pre-trained on abstract algorithmic tasks, we explore the effect of *unfreezing* them during training. We also contrast their performance with unfrozen, randomly-initialised GNNs: as GNNs have been shown to align with Bellman-Ford [Dudzik and Veličković, 2022], as per our taxonomy in Section 2.1.1 we consider these to be **structurally-aligned neural networks (SNNs)**.

We observe from Figure 4.12 that, contrary to established wisdom [Deac et al., 2021], *unfreezing NAPs substantially improves their performance* across all environments, with the largest performance improvements observed in the (more algorithmically-aligned) optimal environment.

But we also observe that *our SNNs match or beat the performance of unfrozen NAPs*: while SNNs perform comparably to NAPs in the optimal environment, they substantially outperform NAPs in the tiebreaking environment.

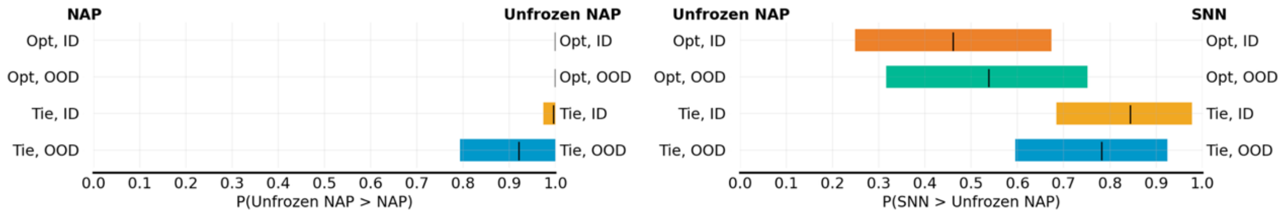


Figure 4.12: 95% CIs for probabilities of improvement in tree-accuracy for unfrozen NAPs over NAPs and for SNNs over unfrozen NAPs, both in-distribution (ID) and out-of-distribution (OOD), in the optimal (Opt) and tie-breaking (Tie) environments.

Conclusions. Contrary to several key claims of NAR, we found that *NAPs do not outperform sDABs in either environment* – and appear to suffer from the same instability issues as sDABs. Moreover, we find that *both unfrozen NAPs and SNNs have better stability and performance than either NAPs or sDABs* (even out-of-distribution). And, on the more complex tiebreaking task, it appears that *introducing a parameter-based algorithmic prior is actively harmful to performance.*

So, in the Warcraft environment, it is very likely that **NAPs trained as per [Veličković et al., 2022] do not alleviate the scalar bottleneck of sDABs**, and that **this bottleneck can instead be overcome by SNNs**.

§4.3.3 Understanding our results: developing hypotheses on the nature of the scalar bottleneck

Now, these results leave us with a compelling question: *why are sDABs outperformed by SNNs, but not by NAPs?* We list two possible hypotheses, which we seek to explore in the next section:

The ensembling-bottleneck hypothesis. SNNs outperform sDABs and frozen NAPs because they can learn to perform *many versions of the algorithm in parallel*, over *simple (possibly scalar) representations*. (Indeed, there is some evidence to suggest that neural networks solving complex problems 'in the wild' learn multiple independent modules performing the same algorithm in parallel [Wang et al., 2022].)

The expressivity-bottleneck hypothesis. SNNs outperform sDABs and frozen NAPs because they can adapt to learn *different variants of the algorithm*, over *more complex representations*, that map more closely to the exact problem at hand.

§4.3.4 Testing the ensembling hypothesis: the power of parallel DABs

One easy way to test whether the ensembling-bottleneck hypothesis holds (i.e. that *increasing algorithmic parallelism without increasing network expressivity* can improve model performance) is to simply *modify our sDAB to execute an ensemble of algorithms in parallel*, and to compare the performance of the resulting **parallel DAB (pDAB)** to that of other algorithmic modules in the optimal environment. As such, we built a pDAB for Bellman-Ford as illustrated in Algorithm 1, and tested it under all conditions explored in Section 4.3.2. We present the results of this analysis in Figure 4.13.

Algorithm 1 A pseudocode implementation of a d-dimensional Bellman-Ford pDAB, for use as an ALGORITHMIC-MODULE in WARCRAFT-NET (Algorithm 4).

```
Require: The following learnable functions:
    $enc_w : \mathcal{I} \to \mathbb{R}^d := Linear$        $dec_h : \mathbb{R}^d \to \mathcal{O}_n := Linear$
    $enc_d : \mathcal{I} \to \mathbb{R}^d := Linear$        $dec_e : \mathcal{I} \to \mathcal{O}_e := Linear$

1: function PDAB($G(\{\mathbf{h}_i\}, \{\mathbf{e}_{ij}\}) : G[\mathcal{I}, \mathcal{I}]) : G[\mathcal{O}_n, \mathcal{O}_e]$
2:     $\mathbf{w}_{ij} : \mathbb{R}^d \leftarrow enc_w(\mathbf{e}_{ij})$        ▷ Extract stack of weights from edge features
3:     $\mathbf{d}_i^{(0)} : \mathbb{R}^d \leftarrow 100 \cdot \sigma(enc_d(\mathbf{h}_i))$        ▷ Extract stack of initial distances from node features
4:     for $t \leftarrow 1, ..., (k \times k)$ do        ▷ Perform BF relaxations on stack of d grid graphs
5:         $\mathbf{d}_j^{(t)} = \min(\mathbf{d}_j^{(t-1)}, \min_{i \to j}(\mathbf{d}_i^{(t-1)} + \mathbf{w}_{ij}))$
6:     return $G(\{dec_h(\mathbf{d}_i^{(k \times k)})\}, \{dec_e(\mathbf{e}_{ij})\})$        ▷ Map final distances to latent space
```

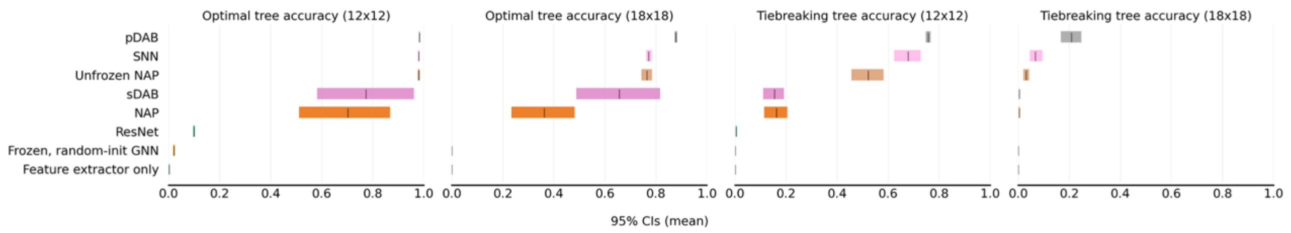


Figure 4.13: 95% CIs for mean tree accuracy, both in-distribution (12x12 grids) and out-of-distribution (18x18 grids), in the optimal and tiebreaking environments.

pDABs dominate all other algorithmic modules across all environments. Observe from Figure 4.14 that *pDABs dominate all other algorithmic modules in the optimal environment.* We see that pDABs very substantially outperform both sDABs (top left) and NAPs (top right), in and out of distribution. And, while pDABs perform on par with unfrozen NAPs (and marginally better than SNNs) in-distribution, they very substantially outperform both out-of-distribution.

Moreover, we observe that *this result holds even in the tie-breaking environment*, where Bellman-Ford alone should not be sufficient to solve the problem. This is surprising, as no positional information is passed to the pDAB.

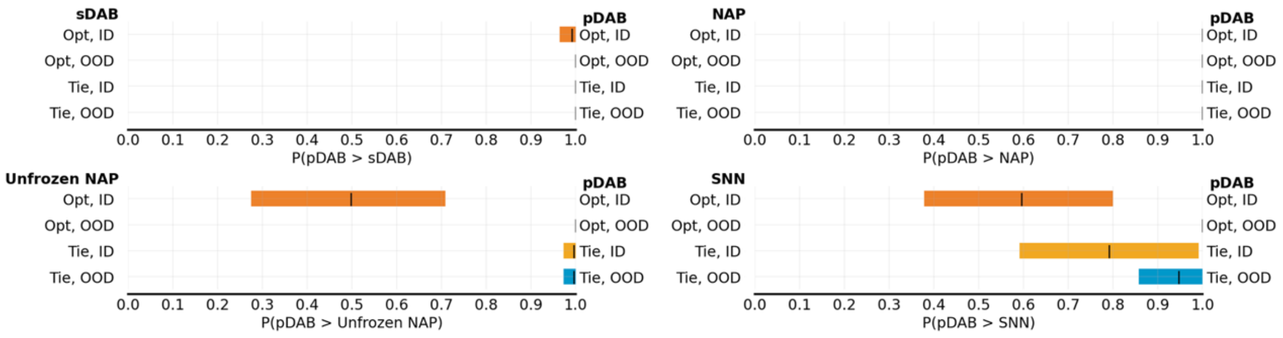


Figure 4.14: 95% CIs for probabilities of improvement in tree-accuracy for pDABs over all other modules tested, both in-distribution (ID) and out-of-distribution (OOD), in the optimal (Opt) and tie-breaking (Tie) environments.

The high dimensionality of pDABs can be leveraged in unanticipated ways to handle variant algorithms. To understand why pDABs perform so well in the tiebreaking environment, let's take a look at how exactly we learn to use them, by visualising the per-tile weight and initial distance matrices we learn to pass them as input (Figure 4.15). While some dimensions of the pDAB are used to predict the true shortest-path lengths for each tile, others appear to be used to generate vertical and horizontal gradients. As these artefacts only appear in pDABs trained on the tiebreaking problem, we hypothesise that our models have learned to use the extra dimensions of the pDAB in an unexpected way, to generate *robust positional encodings* for tiebreaking.

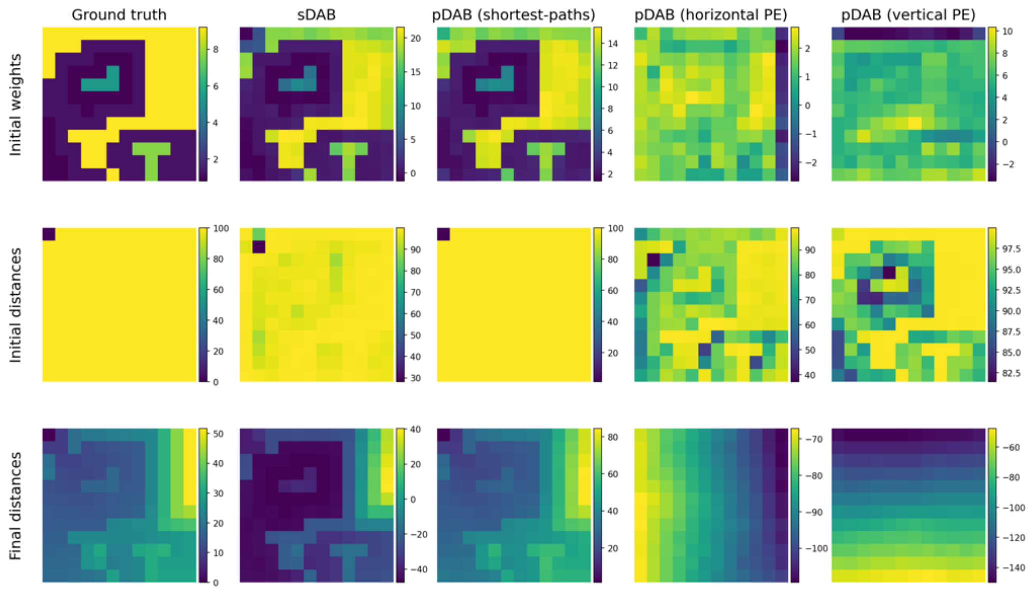


Figure 4.15: The ground-truth initial weight, initial distance and final distance matrices for a randomly-sampled Warcraft terrain map, alongside the inputs that we learn to pass to our sDAB, and representative examples of the three main classes of inputs that we learn to pass to individual Bellman-Ford instances within our pDAB. (For completeness, we present the full set of input and output matrices for each Bellman-Ford instance in our pDAB in Appendix G.)

pDABs are much more efficient to train than SNNs. Finally, we observe that, not only do pDABs not require pre-training, but they are much more efficient to train than SNNs. Indeed, training pDABs incurred an average time per epoch of 11.4 ± 1.7 seconds (across 15 runs); while slightly more compute-intensive than sDABs (9.2 ± 1.4 seconds), they are *over four times faster* than both NAPs (42.5 ± 1.6 seconds) and SNNs (51.4 ± 0.7 seconds).

Conclusions. So, as pDABs dominate all other models in the optimal environment, we have **strong evidence supporting the ensembling-bottleneck hypothesis** – specifically, that *increasing the dimensionality of sDABs* is enough to make them match the performance of SNNs in-distribution.

But beyond simply validating this hypothesis, we also observed that pDABs are much more efficient than SNNs, generalise much better than SNNs out-of-distribution, and even outperform SNNs (which should, in principle, be more flexible than pDABs) on problems whose underlying algorithm deviates slightly from our algorithmic prior. As such, through the discovery of pDABs, **we have achieved a long-standing goal of neural algorithmics** [Cappart et al., 2021]: developing a way to *deterministically distill an algorithm into a robust, high-dimensional processor network* that preserves both the efficiency and correctness guarantees of sDABs while avoiding their performance bottleneck.

5 Conclusions

This project was a success: not only did we meet all our success criteria, but we completed a range of extensions leading to new research results in the field of NAR.

§5.1 Work completed

The MDARF and EPD pipeline. In order to build a laboratory for NAR, we **introduced a new paradigm for constructing neural algorithmic reasoners *à la carte***. Synthesising ideas from DeepMind's *CLRS benchmark* [Veličković et al., 2022] and Uber's *Ludwig* [Molino et al., 2019], we designed, built and tested a novel, extensible **multi-domain algorithmic reasoning framework (MDARF)** for type-driven, declarative ML. On top of this framework, we implemented an **encode-process-decode (EPD) pipeline** for training NAPs, which we derived from first principles using the paradigm of representations-as-types [Olah, 2015] – and, in doing so, **found and alleviated a number of bottlenecks in prior work**. To verify the correctness of our framework, we used this EPD pipeline to train NAPs to imitate complex graph-based algorithms, **matching (and, in some cases, beating) state-of-the-art performance** [Ibarz et al., 2022].

The VI-Implicit-Planner benchmark. To verify the utility of our framework for reproduction, we built the **VI-Implicit-Planner benchmark** by reimplementing the XLVIN architecture [Deac et al., 2021] in our framework, **identifying and fixing a number of bugs and bottlenecks with its original formulation**. Using robust statistical methodology, we found that our implementation reproduced the relative results of [Deac et al., 2021], outperforming the reproduction of [He, 2022]. Moreover, through a series of rigorous ablation tests, we discovered that the performance improvement of XLVIN over its non-algorithmic baseline is unlikely to be due to NAR (as claimed by [Deac et al., 2021]): indeed, **removing the NAP from the XLVIN architecture does not substantially affect performance**.

The Warcraft-Shortest-Path-Tree benchmark. To verify the utility of our framework for research, we also implemented the synthetic **Warcraft-Shortest-Path-Tree benchmark**, inspired by the work of [Vlastelica et al., 2019], which we used to perform a systematic comparison of algorithmic modules. Using robust statistical methodology, while we confirmed the established result that algorithmic modules beat non-algorithmic baselines, **we found evidence to refute one of the central claims of NAR as presented by [Veličković and Blundell, 2021]**, showing that NAPs *do not* overcome the 'scalar bottleneck' of sDABs.

The ensembling bottleneck and the power of parallel DABs. Instead, based on our findings, we developed a new hypothesis: that sDABs instead suffer from an **ensembling bottleneck** of not being able to execute multiple instances of the same algorithm in parallel, and that this bottleneck is alleviated not by NAPs, but by simply using an *unfrozen, structurally-aligned neural network*. Through exploring the effects of *parallelising an sDAB* on its performance, we not only found **strong evidence in support of this hypothesis**, but also **achieved a long-standing goal of neural algorithmics** [Cappart et al., 2021]: developing a way to deterministically distill an algorithm into a robust, high-dimensional processor network that preserves both the efficiency and correctness guarantees of sDABs while avoiding their performance bottleneck.

§5.2 Lessons learned

Developing and evaluating MDARF has certainly been a massive undertaking: at over 10,000 lines of Python code, it is the largest software engineering project I have worked on to date. It also made me realise a few bitter lessons about software engineering. In particular, when developing the MDARF, I realised that there exists an important trade-off between type-safety and verbosity, and that the usability of the MDARF is highly dependent on managing this tradeoff well. Moreover, over the course of the project, I discovered that *early and rapid prototyping* is unreasonably effective, and can be more sustainable in the long run than trying to capture all of the project's requirements from the very beginning.

§5.3 Future work

Having successfully implemented and battle-tested the MDARF through this dissertation, we open the doors to a wide range of future investigations – for instance, exploring various input graph distributions at training time, evaluating our hypotheses in the low-data regime, and widening the range of NAR benchmark environments we consider. Indeed, we have already conducted preliminary investigations into the first two of these, whose results suggest that the conclusions of this dissertation are robust across sparse graph distributions, and reasonably robust in the low-data regime.

We also propose a number of probative experiments which, while out-of-scope for this project due to both time and space constraints, could substantially strengthen our conclusions:

Factoring SNNs through pDABs. While we hypothesised that SNNs break the bottleneck of sDABs through ensembling, the only evidence we have for this is the strong performance of pDABs: indeed, it could be the case that the advantage of SNNs is distinct from the advantage of pDABs. To answer this question, we can test whether or not we can factor the SNN in a trained Warcraft-Net through a pDAB without loss of performance.

Parallelising NAPs. In a similar vein, to test whether NAPs fail to break the bottleneck of sDABs due to a lack of ensembling, we could try building *parallel NAPs* by training an NAP to execute multiple instances of its relevant algorithm at once, and comparing their performance to that of a pDAB.

Bibliography

Rishabh Agarwal, Max Schwarzer, Pablo Samuel Castro, Aaron Courville, and Marc G. Bellemare. Deep Reinforcement Learning at the Edge of the Statistical Precipice. *Advances in Neural Information Processing Systems*, 35:29304–29320, 8 2021. ISSN 10495258. URL https://arxiv.org/abs/2108.13264v4.

Quentin Berthet, Mathieu Blondel, Olivier Teboul, Marco Cuturi, Jean Philippe Vert, and Francis Bach. Learning with Differentiable Perturbed Optimizers. *Advances in Neural Information Processing Systems*, 2020-December, 2 2020. ISSN 10495258. URL https://arxiv.org/abs/2002.08676v2.

Luca Beurer-Kellner, Martin Vechev, Laurent Vanbever, and Petar Veličkovi´ Veličkovi´c. Learning to Configure Computer Networks with Neural Algorithmic Reasoning. 10 2022. URL https://arxiv.org/abs/2211.01980v1.

Barry W. Boehm. A Spiral Model of Software Development and Enhancement. *Computer*, 21 (5):61–72, 1988. ISSN 00189162. doi: 10.1109/2.59.

Antoine Bordes, Nicolas Usunier, Alberto Garcia-Duran, Jason Weston, and Oksana Yakhnenko. Translating Embeddings for Modeling Multi-relational Data. *Advances in Neural Information Processing Systems*, 26(1):270–276, 2013. ISSN 2530-125X. URL https://renati.sunedu.gob.pe/handle/sunedu/3340529.

James Bradbury, Roy Frostig, Peter Hawkins, Matthew James Johnson, Chris Leary, Dougal Maclaurin, George Necula, Adam Paszke, Jake VanderPlas, Skye Wanderman-Milne, and Qiao Zhang. JAX: composable transformations of Python+NumPy programs, 2018. URL http://github.com/google/jax.

Quentin Cappart, Didier Chételat, Elias B. Khalil, Andrea Lodi, Christopher Morris, and Petar Veličković. Combinatorial Optimization and Reasoning with Graph Neural Networks. pages 4348–4355, 2021. doi: 10.24963/ijcai.2021/595.

Anne Condon. The complexity of stochastic games. *Information and Computation*, 96(2): 203–224, 2 1992. ISSN 0890-5401. doi: 10.1016/0890-5401(92)90048-K.

Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms, 3rd Edition.* MIT Press, 2009. ISBN 978-0-262-03384-8. URL http://mitpress.mit.edu/books/introduction-algorithms.

Andreea Deac, Pierre-luc Bacon, and Jian Tang. Graph neural induction of value iteration. pages 3–7, 2019.

Andreea Deac, Petar Veličković, Ognjen Milinkovíc, Pierre Luc Bacon, Jian Tang, and Mladen Nikolíc. Neural Algorithmic Reasoners are Implicit Planners. *Advances in Neural Information Processing Systems*, 19:15529–15542, 10 2021. ISSN 10495258. doi: 10.48550/arxiv.2110.05442. URL https://arxiv.org/abs/2110.05442v1.

Andrew Dudzik and Petar Veličković. Graph Neural Networks are Dynamic Programmers. pages 1–9, 2022. URL http://arxiv.org/abs/2203.15544.

B. Efron. Bootstrap Methods: Another Look at the Jackknife. *https://doi.org/10.1214/aos/1176344552*, 7(1):1–26, 1 1979. ISSN 0090-5364. doi: 10.1214/AOS/1176344552. URL https://projecteuclid.org/journals/annals-of-statistics/

volume-7/issue-1/Bootstrap-Methods-Another-Look-at-the-Jackknife/10.1214/aos/1176344552.fullhttps://projecteuclid.org/journals/annals-of-statistics/volume-7/issue-1/Bootstrap-Methods-Another-Look-at-the-Jackknife/10.1214/aos/1176344552.short.

Jeffrey L. Elman. Finding Structure in Time. *Cognitive Science*, 14(2):179–211, 3 1990. ISSN 1551-6709. doi: 10.1207/S15516709COG1402{_}1. URL https://onlinelibrary.wiley.com/doi/full/10.1207/s15516709cog1402_1https://onlinelibrary.wiley.com/doi/abs/10.1207/s15516709cog1402_1https://onlinelibrary.wiley.com/doi/10.1207/s15516709cog1402_1.

P Erdös and A Rényi. On Random Graphs I. *Publicationes Mathematicae Debrecen*, 6:290, 1959.

William Falcon and The PyTorch Lightning team. PyTorch Lightning, 2019. URL https://github.com/Lightning-AI/lightning.

Gregory Farquhar, Tim Rocktäschel, Maximilian Igl, and Shimon Whiteson. TreeQN and ATreeC: Differentiable Tree-Structured Models for Deep Reinforcement Learning. *6th International Conference on Learning Representations, ICLR 2018 - Conference Track Proceedings*, 10 2017. doi: 10.48550/arxiv.1710.11417. URL https://arxiv.org/abs/1710.11417v2.

Matthias Fey and Jan E. Lenssen. Fast graph representation learning with PyTorch Geometric. In *ICLR Workshop on Representation Learning on Graphs and Manifolds*, 2019.

Martin Fowler, David Rice, Matthew Foemmel, Robert Mee, and Randy Stafford. Patterns of Enterprise Application. *Wesley, Addison*, page 560, 2002.

Erich Gamma, Richard Helm, Ralph Johnson, and John M. Vlissides. Design Patterns: Elements of Reusable Object-Oriented Software. page 416, 11 1994.

Justin Gilmer, Samuel S. Schoenholz, Patrick F. Riley, Oriol Vinyals, and George E. Dahl. Neural Message Passing for Quantum Chemistry. *34th International Conference on Machine Learning, ICML 2017*, 3:2053–2070, 4 2017. doi: 10.48550/arxiv.1704.01212. URL https://arxiv.org/abs/1704.01212v2.

Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning - Ian Goodfellow, Yoshua Bengio, Aaron Courville - Google Books*. 2016. ISBN 9780262035613.

Jean Guyomarch. Warcraft II Open-Source Map Editor, 2017.

William L. Hamilton. *Graph Representation Learning*, volume 14. Morgan and Claypool, 2021.

Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep Residual Learning for Image Recognition. *Proceedings of the IEEE Computer Society Conference on Computer Vision and Pattern Recognition*, 2016-December:770–778, 12 2015. ISSN 10636919. doi: 10.1109/CVPR.2016.90. URL https://arxiv.org/abs/1512.03385v1.

Yu He. Combining Classical Algorithms and Deep Reinforcement Learning Agents. *Part II Dissertation, University of Cambridge*, 2022.

Yu He, Petar Veličković, Pietro Liò, and Andreea Deac. Continuous Neural Algorithmic Planners. 11 2022. URL https://arxiv.org/abs/2211.15839v1.

Ashley Hill, Antonin Raffin, Maximilian Ernestus, Adam Gleave, Anssi Kanervisto, Rene Traore, Prafulla Dhariwal, Christopher Hesse, Oleg Klimov, Alex Nichol, Matthias Plappert, Alec Radford, John Schulman, Szymon Sidor, and Yuhuai Wu. Stable Baselines, 2018. URL https://github.com/hill-a/stable-baselines.

Sture Holm. A Simple Sequentially Rejective Multiple Test Procedure. *Scandinavian Journal of Statistics*, 6(2):65–70, 1979. ISSN 03036898, 14679469. URL http://www.jstor.org/stable/4615733.

Borja Ibarz, Vitaly Kurin, George Papamakarios, Kyriacos Nikiforou, Mehdi Bennani, Róbert Csordás, Andrew Dudzik, Matko Bošnjak, Alex Vitvitskyi, Yulia Rubanova, Andreea Deac, Beatrice Bevilacqua, Yaroslav Ganin, Charles Blundell, and Petar Veličković. A Generalist Neural Algorithmic Learner. (LoG), 2022. URL http://arxiv.org/abs/2209.11142.

Thomas N Kipf and Max Welling. Semi-Supervised Classification with Graph Convolutional Networks. In *International Conference on Learning Representations*, 2017. URL https://openreview.net/forum?id=SJU4ayYgl.

Ilya Kostrikov. PyTorch Implementations of Reinforcement Learning Algorithms, 2018. URL https://github.com/ikostrikov/pytorch-a2c-ppo-acktr-gail.

Jimmy Lin, Daniel Campos, Nick Craswell, Bhaskar Mitra, and Emine Yilmaz. Significant Improvements over the State of the Art? A Case Study of the MS MARCO Document Ranking Leaderboard. *SIGIR 2021 - Proceedings of the 44th International ACM SIGIR Conference on Research and Development in Information Retrieval*, pages 2283–2287, 2 2021. doi: 10.1145/3404835.3463034. URL https://arxiv.org/abs/2102.12887v1.

H. B. Mann and D. R. Whitney. On a Test of Whether one of Two Random Variables is Stochastically Larger than the Other. *https://doi.org/10.1214/aoms/1177730491*, 18(1):50–60, 3 1947. ISSN 0003-4851. doi: 10.1214/AOMS/1177730491. URL https://projecteuclid.org/journals/annals-of-mathematical-statistics/volume-18/issue-1/On-a-Test-of-Whether-one-of-Two-Random-Variables/10.1214/aoms/1177730491.fullhttps://projecteuclid.org/journals/annals-of-mathematical-statistics/volume-18/issue-1/On-a-Test-of-Whether-one-of-Two-Random-Variables/10.1214/aoms/1177730491.short.

Vincent Moens. TensorDict: your PyTorch universal data carrier, 2023. URL https://github.com/pytorch-labs/tensordict.

Piero Molino, Yaroslav Dudin, and Sai Sumanth Miryala. Ludwig: a type-based declarative deep learning toolbox. 9 2019. doi: 10.48550/arxiv.1909.07930. URL https://arxiv.org/abs/1909.07930v1.

Minh Nguyen and Nicolas Wu. Folding over Neural Networks. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 13544 LNCS:129–150, 2022. ISSN 16113349. doi: 10.1007/978-3-031-16912-0{_}5/FIGURES/5. URL https://link.springer.com/chapter/10.1007/978-3-031-16912-0_5.

Danilo Numeroso, Davide Bacciu, and Petar Veličković. Dual Algorithmic Reasoning. 2 2023. URL https://arxiv.org/abs/2302.04496v1.

Christopher Olah. Neural Networks, Types, and Functional Programming, 2015. URL https://research.google/pubs/pub45504/.

Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Köpf, Edward Yang, Zach DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. PyTorch: An Imperative Style, High-Performance Deep Learning Library. *Advances in Neural Information Processing Systems*, 32, 12 2019. ISSN 10495258. URL https://arxiv.org/abs/1912.01703v1.

Anselm Paulus, Michal Rolínek, Vít Musil, Brandon Amos, and Georg Martius. CombOptNet: Fit the Right NP-Hard Problem by Learning Integer Programming Constraints. 5 2021. doi: 10.48550/arxiv.2105.02343. URL https://arxiv.org/abs/2105.02343v2.

Felix Petersen, Christian Borgelt, Hilde Kuehne, and Oliver Deussen. Learning with Algorithmic Supervision via Continuous Relaxations. *Advances in Neural Information Processing Systems*, 20(NeurIPS):16520–16531, 2021. ISSN 10495258.

Antonin Raffin, Ashley Hill, Adam Gleave, Anssi Kanervisto, Maximilian Ernestus, and Noah Dormann. Stable-Baselines3: Reliable Reinforcement Learning Implementations. *Journal of Machine Learning Research*, 22(268):1–8, 2021. URL http://jmlr.org/papers/v22/20-1364.html.

Nils Reimers and Iryna Gurevych. Reporting Score Distributions Makes a Difference: Performance Study of LSTM-networks for Sequence Tagging. *EMNLP 2017 - Conference on Empirical Methods in Natural Language Processing, Proceedings*, pages 338–348, 7 2017. doi: 10.18653/v1/d17-1035. URL https://arxiv.org/abs/1707.09861v1.

Brennan Saeta and Edward Loper. Fiddle: a Python-first configuration library, 2022. URL https://github.com/google/fiddle.

Subham Sekhar Sahoo, Anselm Paulus, Marin Vlastelica, Vít Musil, Volodymyr Kuleshov, and Georg Martius. Backpropagation through Combinatorial Algorithms: Identity with Projection Works. 5 2022. URL https://arxiv.org/abs/2205.15213v3.

John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. Proximal Policy Optimization Algorithms. 7 2017. URL https://arxiv.org/abs/1707.06347v2.

James Somers. What if writing tests was a joyful experience?, 2023.

Heiko Strathmann, Mohammadamin Barekatain, Charles Blundell, and Petar Veličković. Persistent Message Passing. 3 2021. URL https://arxiv.org/abs/2103.01043v2.

Hao Tang, Zhiao Huang, Jiayuan Gu, Bao Liang Lu, and Hao Su. Towards scale-invariant graph-related problem solving by iterative homogeneous graph neural networks. *Advances in Neural Information Processing Systems*, 2020-Decem(NeurIPS):1–37, 2020. ISSN 10495258.

Petar Veličković and Charles Blundell. Neural Algorithmic Reasoning. *Patterns*, 2(7):1–7, 5 2021. ISSN 26663899. doi: 10.1016/j.patter.2021.100273. URL http://arxiv.org/abs/2105.02761http://dx.doi.org/10.1016/j.patter.2021.100273.

Petar Veličković, Guillem Cucurull, Arantxa Casanova, Adriana Romero, Pietro Liò, and Yoshua Bengio. Graph Attention Networks. In *International Conference on Learning Representations*, 2018. URL https://openreview.net/forum?id=rJXMpikCZ.

Petar Veličković, Lars Buesing, Matthew C. Overlan, Razvan Pascanu, Oriol Vinyals, and Charles Blundell. Pointer graph networks. *Advances in Neural Information Processing Systems*, 2020-Decem(NeurIPS), 2020a. ISSN 10495258.

Petar Veličković, Rex Ying, Matilde Padovano, Raia Hadsell, and Charles Blundell. Neural Execution of Graph Algorithms. In *International Conference on Learning Representations*, 2020b. URL https://openreview.net/forum?id=SkgKO0EtvS.

Petar Veličković, Matko Bošnjak, Thomas Kipf, Alexander Lerchner, Raia Hadsell, Razvan Pascanu, and Charles Blundell. Reasoning-Modulated Representations. 2021. URL http://arxiv.org/abs/2107.08881.

Petar Veličković, Adrià Puigdomènech Badia, David Budden, Razvan Pascanu, Andrea Banino, Misha Dashevskiy, Raia Hadsell, and Charles Blundell. The CLRS Algorithmic Reasoning Benchmark. 5 2022. doi: 10.48550/arxiv.2205.15659. URL https://arxiv.org/abs/2205.15659v2.

Marin Vlastelica, Anselm Paulus, Vít Musil, Georg Martius, and Michal Rolínek. Differentiation of Blackbox Combinatorial Solvers. 12 2019. URL http://arxiv.org/abs/1912.02175.

Marin Vlastelica, Michal Rolínek, and Georg Martius. Neuro-algorithmic Policies enable Fast Combinatorial Generalization. 2 2021. URL https://arxiv.org/abs/2102.07456v1.

Kevin Wang, Alexandre Variengien, Arthur Conmy, Buck Shlegeris, and Jacob Steinhardt. Interpretability in the Wild: a Circuit for Indirect Object Identification in GPT-2 small. 11 2022. doi: 10.48550/arxiv.2211.00593. URL https://arxiv.org/abs/2211.00593v1.

Po Wei Wang, Priya L. Donti, Bryan Wilder, and Zico Kolter. SATNet: Bridging deep learning and logical reasoning using a differentiable satisfiability solver. *36th International Conference on Machine Learning, ICML 2019*, 2019-June:11373–11386, 5 2019. URL https://arxiv.org/abs/1905.12149v1.

B. L. Welch. The Generalization of 'Student's' Problem when Several Different Population Variances are Involved. *Biometrika*, 34(1/2):28, 1 1947. ISSN 00063444. doi: 10.2307/2332510.

Keyulu Xu, Jingling Li, Mozhi Zhang, Simon S. Du, Ken-ichi Kawarabayashi, and Stefanie Jegelka. What Can Neural Networks Reason About? 5 2019. URL https://arxiv.org/abs/1905.13211v4.

Keyulu Xu, Mozhi Zhang, Jingling Li, Simon S. Du, Ken-ichi Kawarabayashi, and Stefanie Jegelka. How Neural Networks Extrapolate: From Feedforward to Graph Neural Networks. 9 2020. URL https://arxiv.org/abs/2009.11848v5.

A Deep learning, representations and types

In this appendix, we revisit some of the fundamental ideas behind deep learning through the lens of *functional programming*, and establish a few elements of non-standard notation.

§A.1 Representations and the manifold hypothesis

One of the most popular perspectives on deep learning is the *representations narrative*: the idea that a deep neural network is a composition of *differentiable, parameterised functions*,[1] such that each successive function transforms our data into a new shape – or *representation* – that's more useful for the problem we want to solve.

More precisely, it is widely believed [Goodfellow et al., 2016] that most real-world, high-dimensional data is concentrated around a low-dimensional curved surface (or *manifold*) M capturing its geometrical structure, with input data points $\mathbf{x} : \mathbb{R}^n$ lying on the image of an embedding $M \hookrightarrow \mathbb{R}^n$. In this framing, our representations (the 'shapes' that the layers f_i of our network map between) are the images of *manifold embeddings* into $\mathbb{R}^n$, and the layers themselves are *maps between manifolds* lifted to their embeddings.

In this dissertation, we will refer to the images of (learned) manifold embeddings into $\mathbb{R}^n$ as **representations** or **latent spaces** $\mathcal{R}$, and the elements of a representation as **hidden states**, **latent states** or **latents** $\mathbf{v} \in \mathcal{R}$. (Confusingly, the term **embedding** is also sometimes used to refer to a hidden state – typically the image of a feature vector under an encoder.)

§A.2 Representations as types

Now, given this compositional perspective on deep learning, an interesting parallel begins to emerge between neural networks and functional programs [Olah, 2015, Nguyen and Wu, 2022]. Just as a representation can be seen as the embedding of a manifold in n-dimensional space, a type can be seen as the embedding of some kind of data in n bits. And just as functional programming (FP) is characterised by the composition of maps between types, deep learning (DL) is characterised by the composition of maps between representations.

So that's a cute observation. But why care? One salient implication here is that *just as functions can only be composed together if their types agree, neural networks can only be composed together if their representations agree*: when we train a network, adjacent layers will negotiate the representation they communicate in (i.e. the second will learn to consume as its input representation the output representation of the first), and after training, passing data to a network of the wrong representation will (very likely) lead to nonsensical results.

As we'll see later on, it's often helpful to make explicit these 'representational constraints' imposed on our neural modules by the structure of our network. To do so, we devise the novel formalism of *representation variables*: type variables that express the *most restrictive*

[1]For instance, recall from Part IB *Artificial Intelligence* that the simplest neural network architecture is the **multi-layer perceptron (MLP)**: the composition $f := f_L \circ ... \circ f_1$ of L learnable functions $f_i = \phi(\mathbf{W}_i\mathbf{x}+\mathbf{b}_i)$ for $\mathbf{W}_i, \mathbf{b}_i$ learnable parameters and ϕ a elementwise non-linearity.

constraints on the input and output representations of a learnable function that we know at compile-time (i.e. before training).

Specifically, we introduce **representation variables** $\mathcal{A}, \mathcal{B}, \mathcal{C}, \ldots$ to denote arbitrary representations that a trained neural network learns to map between.

We introduce typing judgements $f : \mathcal{A} \to \mathcal{B}$ for functions $f : \mathbb{R}^n \to \mathbb{R}^m$, and $e : \mathcal{C}$ for vector-valued expressions $e : \mathbb{R}^n$, where $\mathcal{A}, \mathcal{B}, \mathcal{C}, \ldots$ are the most general solution to the constraints emitted by e.g.

$$\frac{f : \mathcal{A} \to \mathcal{B} \qquad e : \mathcal{C}}{f(e) : \mathcal{D}} \; \boxed{\mathcal{A} = \mathcal{C} \wedge \mathcal{B} = \mathcal{D}} \qquad \frac{e_1 : \mathcal{A} \qquad e_2 : \mathcal{B}}{e_1 \,\|\, e_2 : \mathcal{C}} \; \boxed{\mathcal{C} = \mathcal{A} \times \mathcal{B}}$$

Observe that, given a neural network, we can easily perform *representation variable inference* through a constraint-based analysis, walking the network and unifying representation variables where necessary.

§A.3 Neural networks as functional programs

One other implication of this is that we can view the construction of neural networks as the construction of *differentiable, parameterised pure functional programs* (an idea embodied by DL frameworks like JAX [Bradbury et al., 2018]). And by do so, we can take powerful ideas for structuring functional programs, pass them chunks of neural network, and get out powerful, principled neural architectures.

Consider, for instance, the problem of *reducing* a list of data of type `a` to a single value of type `b`. In FP, the canonical way to do so is to *fold* over the list with an appropriate aggregation function and initial element:

```
fold :: ((a, b) -> b) -> b -> [a] -> b
fold f z [] = z
fold f z (x:xs) = f (x, fold f z xs)
```

But if we parameterise our fold by a learnable accumulator $f : \mathcal{R} \times \mathcal{S} \to \mathcal{S}$ and a learnable initialisation element $z : \mathcal{S}$, we obtain a *recurrent neural network* [Elman, 1990], a popular family of architectures for reducing a list of latents $\mathbf{x}_i \in \mathcal{R}$ to some $\mathbf{y} \in \mathcal{S}$:

```
rnnCell :: Rep r, s => Learnable ((r, s) -> s)
initialState :: Rep s => Learnable s

rnn :: Rep r, s => Learnable ([r] -> s)
rnn = fold rnnCell initialState
```

Indeed, throughout this dissertation, we'll see that thinking about neural networks as *differentiable, pure functional programs* often gives us a natural way to frame the design of various algorithmic modules.

B An introduction to reinforcement learning and value iteration

In this appendix, we provide an overview of reinforcement learning and the value iteration algorithm.

§B.1 A stylised overview of reinforcement learning

At a very high level, the field of RL is concerned with building **agents** that learn to act within some **environment** through trial and error. More precisely, suppose we have an agent in some environment (e.g. some ML model playing a video game), that we can reward or punish based on the actions it takes. We're interested in answering the question: *how can we build an agent that, based on the feedback we give it, can learn to get as much reward as possible?*

In order to answer this question, let's try building a mathematical model of *agents acting in a stochastic environment*, and see if we can solve our problem there. For the kinds of environments we want to train agents to act in, the way in which the environment evolves over time is partly random and partly due to the actions of the agent. The classical way to model decision-making under these conditions is through a **Markov decision process (MDP)**, a structure $M := (S, A, P, R, s_0)$ not entirely dissimilar to that of a Markov chain:

- We model our world as a (possibly infinite) set S of **states**, such that our agent is always in one particular state at any given time.
- We model our agent's interaction with the world as taking place over a series of **time-steps** $t = 0, 1, 2, ...$
- Our agent starts in some **initial state** $s_0 \in S$.
- At every time-step t, our agent (in state s_t) gets to choose some **action** a_t from a fixed set A. (Note that, for simplicity, we will restrict ourselves to **discrete action spaces** – in other words, we assume A is finite.) Based on their chosen action and their current state, they will obtain some (possibly negative) **reward** $R(s_t, a_t) \in \mathbb{R}$, and will be sent to some **next state** s' with probability $P(s' \mid s_t, a_t)$.
- We call the sequence of actions and states $\tau = (s_0, a_0, s_1, a_1, ...)$ observed over the course of our agent's interaction with its environment a **trajectory**.

So, given this model, we want to build an agent that tries to maximise its 'cumulative reward' $R(\tau)$ (known as its **return**) over the course of a trajectory. Note that there are a number of ways to formalise this notion of 'return'; for illustration, one popular definition is the **infinite-horizon discounted return** $R(\tau) = \sum_{t=0}^{\infty} \gamma^t R(s_t, a_t)$, which values future rewards less than present ones according to an exponential *discount factor* $\gamma \in [0, 1]$.

As such, our goal is to design a **stochastic policy** π for our agent (i.e. a strategy that, given a current state s_t, gives us a distribution $a_t \sim \pi(\cdot \mid s_t)$ over actions it should take) such that sampling actions according to π maximises its **expected return** $\mathbb{E}_{\tau\sim\pi}[R(\tau)]$ over trajectories τ. Indeed, **finding the optimal stochastic policy $\pi^* = \arg\max_\pi \mathbb{E}_{\tau\sim\pi}[R(\tau)]$ for a given environment is the central optimisation problem of RL**.

§B.2 Value iteration: finding the optimal policy given a known MDP

So, suppose we can model our environment perfectly with some MDP M, for which S, A, P, R, s_0 are all known. (We call such an MDP a **tabular MDP**.) Given this, can we find the optimal policy π^*?

To motivate this discussion, notice that, if we had what's known as an **optimal value function** $V^*(s) = \mathbb{E}_{\tau\sim\pi^*}[R(\tau) \mid s_0 = s]$ – an oracle that, for every state s, could tell us our expected return if we acted optimally starting from s – we could use this to find the expected return $\mathbb{E}_{\tau\sim\pi^*}[R(\tau) \mid s_0 = s, a_0 = a]$ for any a, and our optimal policy would simply be to take the action maximising this expected return.

Now, for an MDP under infinite-horizon discounted return, expanding the equation above yields the following recurrence:

$$V^*(s) = \max_a \mathop{\mathbb{E}}_{s'\sim P(\cdot|s,a)}[R(s,a) + \gamma V^*(s')] \tag{B.1}$$

As such, we can solve for V^* in a process known as **value iteration (VI)**, by randomly initialising $V_0^*(s)$ for all $s \in S$ and iteratively applying the following equation until convergence:

$$V_{t+1}^*(s) = \max_a \left(R(s,a) + \gamma \sum_{s'} P(s' \mid s,a) V_t^*(s') \right) \tag{B.2}$$

(Note that, while each iteration can be computed in $O(|A||S|^2)$ time, the number of iterations required until convergence may grow exponentially with the discount factor γ [Condon, 1992].)

And, as mentioned earlier, once we have such a value function V^*, our optimal policy in this environment simply becomes to (deterministically) choose the action a that maximises our expected return $\mathbb{E}_{\tau\sim\pi^*}[R(\tau) \mid s_0 = s, a_0 = a]$:

$$\pi^*(s) = \arg\max_a \left(R(s,a) + \gamma \sum_{s'} P(s' \mid s,a) V^*(s') \right) \tag{B.3}$$

Stepping back a little, we can interpret VI as a *planning algorithm*: we explore all possible sequences of actions to find the ones that will lead to the maximal reward in expectation, using the property of *semiring distributivity* (à la Bellman-Ford) to prevent the exponential blow-up in space complexity that this would otherwise entail.

§B.3 Reinforcement learning in practice

So, in the world of mathematics, we've built a framework for modelling environments as MDPs, and we've derived an algorithm to find the optimal policy for a tabular MDP. But can we translate this back to the real world, to actually build agents that learn?

Now, while any real-world problem can be modelled as a (sufficiently large) MDP, we run into a number of issues if we naïvely try to build an optimal agent for this MDP through VI. Indeed, in almost all cases, MDPs modelling environments of interest are intractably large (and therefore infeasible to tabulate); moreover, we often don't even know the exact state our agent is currently in (e.g. the RAM state of a game console), and can only obtain a partial **observation** of this state (e.g. what's currently displayed on the screen).

Instead, the typical approach to solving these problems is to make some component of the MDP *learnable*, and to try to arrive at an approximate solution through gradient descent. While there exist many ways to do this, in this dissertation we'll focus on the method of **model-free policy optimisation**:

- We define our policy as a learnable **policy network** $a_t \sim \pi_\theta(\cdot \mid o_t)$ – i.e. a neural network mapping observations o_t to a distribution over possible next actions a_t.
- We then train our policy by gradient ascent to directly maximise its expected return $J(\pi_\theta) := \mathbb{E}_{\tau \sim \pi_\theta}[R(\tau)]$, by periodically sampling trajectories $\mathcal{D} = \{\tau_i\}_{i=1,\ldots,n}$ from policy π_θ, and using these to estimate the **policy gradient** $\nabla_\theta J(\pi_\theta)$.

Now, as the naïve estimator for policy gradient

$$\nabla_\theta J(\pi_\theta) = \underset{\tau \sim \pi_\theta}{E}\left[\sum_{t=0}^{T} \nabla_\theta \log \pi_\theta(a_t \mid s_t) R(\tau)\right] \tag{B.4}$$

(obtained by expanding $J(\pi_\theta) := \mathbb{E}_{\tau \sim \pi_\theta}[R(\tau)]$) has very high variance, in practice we use a lot of tricks to reduce the variance of our estimate. One of the most popular such bundles of tricks (and the one we'll use for this problem) is **proximal policy optimisation (PPO)** [Schulman et al., 2017].

A notable distinction of PPO is that it introduces a learnable approximator $V_\phi(o_t)$ of our **on-policy value function** $V^\pi(s_t) := \mathbb{E}_{\tau \sim \pi_\theta}[R(\tau) \mid o_0 = o]$ that we train alongside our policy network π_θ. We call this pair of networks (which typically share some modules) an **actor-critic network**; here, the policy network π_θ is the 'actor', and the value network V_ϕ assessing the long-term reward of its behaviour is the 'critic'.

C Implementation details for the EPD pipeline

We explain the stages of Figure 3.16 in detail:

Specifying algorithmic types. To get started, we need batchable type representations for $List[I]$, $List[H]$ and $List[O]$. As per Section 3.3.1, we can easily use *primitive and compound feature types* to build type representations `Input, Hint, Output <: Batchable` for I, H and O – and, in order to represent lists in the MDARF, we introduce a new compound type `Trajectory[BatchableT]`, representing sequences of (optional) elements of type `BatchableT`. We also need to define type representations for $\mathcal{I}$ and $\mathcal{S}$: for simplicity, we assume these are *latent graphs*.

Implementing the sampler and generating the streaming dataset. Once we've defined our `Input` type, as per Section 3.3.2 we can easily build (and customise) a default sampler for our trajectory type `Trajectory[Input]`. Then, given our input sampler, we can use $A : List[I] \rightarrow List[H] \times List[O]$ to implement a generator for **algorithmic traces** (i.e. tuples $List[I] \times List[H] \times List[O]$), encoded for batching purposes as a new compound type `Trace[InputT, HintT, OutputT]`.

To facilitate this process, users can obtain generators for new algorithms by subclassing the interface `AlgorithmSampler[InputT, HintT, OutputT] <: Sampler[Trace[InputT, HintT, OutputT]]` – specifying the relevant types, an appropriate input sampler, and an execution function generating traces from algorithmic inputs.

These generators can be used to build either *fixed-size* or *streaming* (i.e. 'infinite') datasets of batched algorithmic traces:

```
sampler_config = ALGORITHM.default_config(GRAPH_SIZE)
sampler = fdl.build(sampler_config)
dataset = SamplerIterableDataset(sampler, rng, BATCH_SIZE)
```

Specifying the neural processor and deriving the wrapped NAP. Now, given we have our data and our algorithmic types, all we need to build our NAP is a processor $P : G[\mathcal{S}_N, \mathcal{S}_E] \times G[\mathcal{I}_N, \mathcal{I}_E] \rightarrow G[\mathcal{S}_N, \mathcal{S}_E]$ (i.e. a **recurrent GNN cell**). In this dissertation, following prior work [Veličković et al., 2022, Ibarz et al., 2022], we use the state-of-the-art **GMPNN** and **Triplet-GMPNN** processors; while we explore the Triplet-GMPNN processor to evaluate our framework against the CLRSB, we use the GMPNN processor when training NAPs for our benchmark environments.

For convenience, we provide an `EPDExecutor[InputT, HintT, OutputT]` module that, given an algorithm and processor, generates the relevant encoders and decoders between feature and latent types, and applies the processor to algorithmic input according to the EPD pipeline:

```
executor_config = EPDExecutor.default_config(
    algorithm=ALGORITHM, processor=ClrsMPNN,
    latent_input_dims=LATENT_DIMS, latent_hint_dims=LATENT_DIMS)
executor = fdl.build(executor_config).to(DEVICE)
```

Training and evaluating the NAP. Finally, given an executor and an algorithmic dataset, this wrapped NAP can be trained in PyTorch using one's method of choice. For illustration, in Appendix D, we demonstrate a complete training pipeline for a Bellman-Ford NAP in just 38 lines of vanilla PyTorch (excluding imports). For convenience, we also provide a fully-featured experimental pipeline for training NAPs, built with *PyTorch Lightning* [Falcon and The PyTorch Lightning team, 2019].

D Example code for training NAPs with the MDARF

We present below a working code snippet (excluding imports) that trains an NAP to imitate Bellman-Ford.

```python
ALGORITHM = BellmanFordSampler; GRAPH_SIZE = 16
LATENT_DIMS = LatentGraphDims(
    hidden_node=128, hidden_edge=128, hidden_feature=128)
BATCH_SIZE = 32; LR = 0.001; DEVICE = 'cuda:0'
rng = np.random.default_rng()

# Building the executor
executor_config = EPDExecutor.default_config(
    algorithm=ALGORITHM, processor=ClrsMPNN,
    latent_input_dims=LATENT_DIMS, latent_hint_dims=LATENT_DIMS)
executor = fdl.build(executor_config).to(DEVICE)

# Building the dataset
sampler_config = ALGORITHM.default_config(GRAPH_SIZE)
sampler = fdl.build(sampler_config)
dataset = SamplerIterableDataset(sampler, rng, BATCH_SIZE)
dataloader = DataLoader(dataset, batch_size=None)

# Training loop
optimizer = torch.optim.Adam(executor.parameters(), lr=LR)
for i, (batch_hard, batch_soft) in enumerate(dataloader):
    optimizer.zero_grad()
    batch_hard = batch_hard.to(DEVICE); batch_soft = batch_soft.to(DEVICE)
    pred_hints, pred_outputs = executor.execute(
        batch_soft.inputs, batch_soft.hints)

    output_loss = pred_outputs.summary_loss(batch_hard.outputs).reduce().data
    hint_loss = pred_hints.summary_loss(batch_hard.hints).reduce().data
    loss = output_loss + hint_loss
    loss.backward()
    optimizer.step()

    output_metric = (
        pred_outputs.soft_output_to_hard()
        .summary_evaluate_against(batch_hard.outputs)
        .map(lambda x: x.data.item()).data)
    print(f'[Batch {i + 1:2d}] loss: {loss.item():.3f}, '
          f'output pi: {output_metric["pointers"]["pi"]:.3f}')
```

E Implementation details for the benchmarks

§E.1 The VI-Implicit-Planner benchmark

Our first benchmark (VI-Implicit-Planner) allows for the comparison of algorithmic modules in the context of providing an *implicit-planning policy network* with an inductive bias towards value iteration. So, in order to build this benchmark, we must first design such a policy network that, when given an appropriate algorithmic module, can use it to execute value iteration over a latent MDP.

We base our policy network on that of the **eXecuted Latent Value Iteration Network (XLVIN)** [Deac et al., 2021], the current state-of-the-art for implicit planning and one of the flagship results of NAR. As such, in this section, we give a rough outline of our VI-Implicit-Planner benchmark, by briefly reviewing the XLVIN architecture and discussing a few improvements we make to its design.

A note on RL environments. Recall from Section B that, for a given RL environment, its action space can be either *continuous* or *discrete*, and its underlying MDP can be either *deterministic* or *non-deterministic*. While some work [He et al., 2022] has been conducted extending XLVIN to continuous action spaces, for the purposes of this dissertation we only explore environments with discrete action spaces. Furthermore, as the environments we intend to explore with VI-Implicit-Planner are deterministic, we explore XLVIN within the context of environments with underlying deterministic MDPs.

A note on implementation. While the rest of this section mostly discusses XLVIN (and its modifications) at a high level, we briefly mention some implementation details. Notably, in contrast to the works of [Deac et al., 2021] and [He, 2022] (which build custom training wrappers for XLVIN by modifying the library of Kostrikov [2018]), we instead implement XLVIN as a policy within *Stable Baselines 3* [Raffin et al., 2021], one of the most popular frameworks for deep RL, in order to ensure reliable, reproducible results.

§E.1.1 A review of the XLVIN architecture

XLVIN is an *actor-critic policy network*: a network $(\pi_\theta(a \mid o), V_\phi(o))$ that, given some observation $o \in O$ of the current state, predicts both a distribution π_θ over next actions from the current state, and the expected long-term reward V_ϕ of acting according to π_θ from the current state.

As we may wish to deploy our network in environments with a range of observation spaces (e.g. images of an Atari gameboard, or scalars representing the position and velocity of a cartpole), we define the XLVIN architecture as a function $\textsc{Xlvin} : \mathcal{O} \to \mathcal{P} \times \mathcal{V}$ mapping latent observations to latent policy and value representations, from which we can (linearly) project out policy logits and value predictions. This function can then be precomposed with an appropriate encoder $enc_o : O \to \mathcal{O}$, and postcomposed with appropriate decoders $dec_\pi : \mathcal{P} \to Categorical[A]$ and $dec_v : \mathcal{V} \to \mathbb{R}$, to yield a policy network operating over observation space O.

We present a pseudocode implementation of the XLVIN architecture in Algorithm 2.

A high-level summary of XLVIN. The key idea behind XLVIN is that we introduce an algorithmic inductive bias towards implicit planning by *inferring a latent state* from our observation, *inferring a tree of latent next states* from this latent state, *performing value iteration* over this tree with an algorithmic module, and *projecting latent policy and value representations* from the results of this process.

Inferring latent states from observations. To lift latent observations $\mathbf{o} \in \mathcal{O}$ to latent states $\mathbf{s} \in \mathcal{S}$, XLVIN uses an MLP encoder $z : \mathcal{O} \to \mathcal{S}$.

Generating the tree of latent next states. In order to generate our tree of latent next states, we must be able to generate the latent next state $\mathbf{s}'$ to which we transition after taking some action a from latent state $\mathbf{s}$. To do so, XLVIN introduces an MLP transition function $T : \mathcal{S} \times A \to \mathcal{S}$, such that taking action a in latent state $\mathbf{s}$ sends us to $\mathbf{s} + T(\mathbf{s}, a)$. We then use this transition function to roll out a tree of depth k (where k is a hyperparameter to be chosen), representing all possible length-k trajectories $\mathbf{s}_0 \xrightarrow{a_1} ... \xrightarrow{a_k} \mathbf{s}_k$ starting from our initial latent state $\mathbf{s}_0$.

Regularising towards sensible latent state embeddings. To ensure we learn sensible latent state embeddings, we apply the *TransE* loss [Bordes et al., 2013] as a regularisation term: for all observed transitions (o, a, o'), we ensure that our latent state embedding of observation o' is as close as possible to our predicted next state embedding of observation o after applying action a in latent space, and as far away as possible from latent state embeddings of other observations $\tilde{o}$. In other words, we ensure the following diagram commutes:

$$\begin{tikzcd}
\mathcal{S} \arrow[r, "\lambda \mathbf{s}.\mathbf{s}+T(\mathbf{s},a)"] & \mathcal{S} \\
O \arrow[u, "z \circ enc_O"] \arrow[r, "\text{take action } a"'] & O \arrow[u, "z \circ enc_O"']
\end{tikzcd}$$

Performing value iteration over the tree. After generating the tree, we linearly map its nodes into algorithmic latent space, and use an algorithmic module of our choice to perform value iteration over the tree; for the original XLVIN implementation, this is a (pre-trained, frozen) NAP returning a result embedding $\mathbf{v}'_i$ for each node i of the tree.

Projecting latent policy and value representations. Finally, in the original XLVIN implementation, we take the result embedding $\mathbf{v}'_0 \in \mathcal{S}$ for the root node of the tree, and project policy and value representations out from the pair $(\mathbf{v}'_0, \mathbf{s}_0)$.

§E.1.2 Modifications made to the XLVIN architecture

Now, upon analysing this architecture, we noticed a number of potential issues with the way in which the value iteration NAP was trained and used in XLVIN. As such, we outline some of these issues, and discuss the ways in which we fixed them.

Analysing the XLVIN NAP

Now, although the latent MDP constructed by the XLVIN is deterministic, Deac et al. [2021] follow Deac et al. [2019] and instead train the XLVIN NAP to, at every timestep, imitate the *delta* of a single step of value iteration

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

Algorithm 2 A pseudocode implementation of XLVIN [Deac et al., 2021], with modifications we made marked in red.

```
Require: For $FC := Linear \triangleright ReLU$, the following learnable functions:
    $z : \mathcal{O} \to \mathcal{S} := FC^3$                                   $dec_n : \mathcal{V}_n^{(out)} \to \mathcal{S}' := Linear$
    $enc_n : \mathcal{S} \to \mathcal{V}_n^{(in)} := Linear$                     $\pi : \mathcal{S} \times \mathcal{S}' \times \mathcal{S}'^{|A|} \to \mathcal{P} := Linear$
    $enc_e : A \times \mathbb{R} \to \mathcal{V}_e^{(in)} := Linear$             $V : \mathcal{S} \times \mathcal{S}' \times \mathcal{S}'^{|A|} \to \mathcal{V} := Linear$
    $T : \mathcal{S} \times A \to \mathcal{S} := FC^2 \triangleright LayerNorm \triangleright FC$

1: function EXPAND-TREE($\mathbf{s}_0 : \mathcal{S}$, $k : \mathbb{N}$) : $G[\mathcal{S}, \mathcal{V}_e^{(in)}]$      ▷ Expand tree to depth k
2:     $g \leftarrow$ newTree($root = \mathbf{s}_0$)
3:     for $i \leftarrow 1, ..., k$ do
4:         for $\mathbf{s}_u \leftarrow$ getLeaves($g$) do      ▷ Expand each leaf state with all possible actions.
5:             $g$.addEdges($\{\mathbf{s}_u \xrightarrow{enc_e(a,\gamma)} T(\mathbf{s}_u, a) \mid a \in A\}$)

6: function VALUE-ITERATION($tree : G[\mathcal{V}_n^{(in)}, \mathcal{V}_e^{(in)}]$, $k : \mathbb{N}$) : $G[\mathcal{V}_n^{(out)}, \mathcal{V}_e^{(out)}]$
7:     ▷ Executes algorithmic module (NAP or sDAB) over depth-k tree.
8:     ▷ Example: for NAP $P : G[\mathcal{V}_n^{(out)}, \mathcal{V}_e^{(out)}] \times G[\mathcal{V}_n, \mathcal{V}_e] \to G[\mathcal{V}_n^{(out)}, \mathcal{V}_e^{(out)}]$:
9:     $state \leftarrow G(\{\mathbf{0}\}, \{\mathbf{0}\})$
10:    for $i \leftarrow 1, ..., k$ do $state \leftarrow P(state, tree)$
11:    return $state$

12: function XLVIN($\mathbf{obs} : \mathcal{O}$, $k : \mathbb{N}$) : $\mathcal{P} \times \mathcal{V}$      ▷ XLVIN policy-value network with tree depth k
13:    ▷ Map observation to latent state
14:    $\mathbf{s}_0 \leftarrow z(\mathbf{obs})$

15:    ▷ Generate latent-state tree (rooted at $\mathbf{s}_0$)
16:    $G(\{\mathbf{s}_i\}, \{\mathbf{e}_{ij}\}) \leftarrow$ EXPAND-TREE($\mathbf{h}_0$)

17:    ▷ Map to algorithmic latent space and execute algorithmic module
18:    $\mathbf{v}_i \leftarrow enc_n(\mathbf{s}_i)$
19:    $G(\{\mathbf{v}'_i\}, \{\mathbf{e}'_{ij}\}) \leftarrow$ VALUE-ITERATION($G(\{\mathbf{v}_i\}, \{\mathbf{e}_{ij}\}), k$)
20:    $\mathbf{s}'_i \leftarrow dec_n(\mathbf{v}'_i)$

21:    ▷ Project policy / value from original state, result and neighbours
22:    $S'_{\mathcal{N}} \leftarrow \{\mathbf{s}'_i \mid \mathbf{s}'_0 \xrightarrow{a} \mathbf{s}_i, a \in A\}$
23:    return $\pi(\mathbf{s}_0, \mathbf{s}'_0, S'_{\mathcal{N}}), V(\mathbf{s}_0, \mathbf{s}'_0, S'_{\mathcal{N}})$
```

over *non-deterministic* MDPs represented as stacks of graphs. This NAP is then deployed in XLVIN as detailed in Algorithm 3.

Algorithm 3 An illustration of the usage of the pre-trained VI NAP in XLVIN, for f a pre-trained linear layer.

```
1: function VALUE-ITERATION($tree : G[\mathcal{V}_n^{(in)}, \mathcal{V}_e]$, $k : \mathbb{N}$) : $G[\mathcal{V}_n^{(out)}, \mathcal{V}_e]$
2:     $G(\{\mathbf{h}_s^{(in)}\}, \{\mathbf{e}_{s's}^{(in)}\}) \leftarrow tree$
3:     $\mathbf{h}_s^{(0)} \leftarrow \mathbf{0}$
4:     for $t \leftarrow 0, ..., k-1$ do
5:         $\mathbf{h}_s \leftarrow \mathbf{h}_s^{(in)} + \mathbf{h}_s^{(t)}$
6:         $\mathbf{h}_s^{(t+1)} := \max_{s' \xrightarrow{a} s} \left(\mathbf{h}_s + f(\mathbf{h}_{s'}, \mathbf{h}_s, \mathbf{e}_{s's}^{(in)})\right)$
7:     return $G(\{\mathbf{h}_s^{(k)}\}, \{\mathbf{e}_{s's}^{(in)}\})$
```

Now, recall that, as we *freeze* our NAP before deployment, strictly speaking, the only guarantees we have on its behaviour are those enforced by the way in which we pre-trained it (see Section 3.4 for more details).

While the derivation is rather messy (and omitted for brevity), considering the way in which this NAP was pre-trained and subsequently deployed, we obtain the following guarantee for the behaviour of the VALUE-ITERATION function in Algorithm 3:

There exist maps enc_n, enc_e, dec such that, for all timesteps t:

- If $\mathbf{h}_s = enc_n([V_t(s), R(s,a)])\ \forall a$ and $\mathbf{e}_{s's} = enc_e([1, \gamma])$ for all states s, s',
- then $V_{t+1}(s) - V_t(s) = dec(\mathbf{h}_s^{(t+1)})$.

Given this, we observe the following issues:

- For the specification to be met, for all states s, our learned latent representation of reward (within $\mathbf{h}_s$) must be the same for all actions a out of s. In other words, **we can only model deterministic MDPs with per-state rewards** – i.e. MDPs with rewards $R(s)$ as opposed to $R(s,a)$ – restricting expressivity in a (likely) unintended way.
- While the NAP's processor is only ever trained on single steps of value iteration, when deployed, its output latent state is passed in as part of the input latent state for the next time-step. So, in order for us to use the NAP in this way and still satisfy the specification, we must learn representations $\mathbf{h}_s^{(in)}$ such that, for some $\mathbf{h}_s^{(t)}$ satisfying $V_t(s) - V_{t-1}(s) = dec(\mathbf{h}_s^{(t)})$, we have $\mathbf{h}_s^{(in)} + \mathbf{h}_s^{(t)} = enc_n([V_t(s), R(s,a)])\ \forall a$. Now, while our NAP may happen to generalise beyond its specification, in general there is no guarantee that it will be possible to learn a $\mathbf{h}_s^{(in)}$ satisfying this property – not least because there's no guarantee $\mathbf{h}_s^{(t)}$ even stores data about $V_t(s)$, as opposed to just the difference $V_t(s) - V_{t-1}(s)$. In short, **we use our pre-trained NAP in a way that 'doesn't type-check', and as such, the behaviour of our NAP may be no more meaningful than that of a randomly-initialised, frozen GNN**.
- Recall that, after the execution of VALUE-ITERATION, we attempt to recover the predicted policy and value from state $\mathbf{s}_0'$, which contains at most as much information as is present in state $\mathbf{h}_0^{(k)}$. Observe that, as we only guarantee that $V_{t+1}(s) - V_t(s)$ is recoverable from this state, **our decoder has insufficient information to predict the optimal next action** (i.e. to transition to the neighbouring state s' with the highest $V^*(s')$).

```
VIInput = Graph[{reward: Scalar}, {gamma: Scalar}, Empty, Empty]
VIHint = Graph[{value: Scalar}, Empty, Empty, Empty]
VIOutput = Graph[Empty, Empty, Empty, NodePointer]

def run(
    x: VIInput,
) -> EPDTrace[VIInput, VIHint, VIOutput]:
    h: VIHint = Graph(nodes=[{value=0.0} for _ in x.nodes])
    trace = EPDTraceBuilder(input=x, initial_hint=h)

    iter_diff = inf
    while iter_diff > 1e-4:
        new_values = [0.0 for _ in x.nodes]
        for s in x.nodes:
            # V_{t+1}(t) = max_{t -> s} (R(s) + gamma V_t(t))
            new_values[s], output.pi[s] = max(
                (x[s].reward + e.gamma * h[t].value, t)
                for (t, e, s) in in_edges_of(s))

        iter_diff = max(abs(h[i].value - new_values[i]) for i in range(n))
        for i in range(n): h[i].value = new_values[i]
        trace.step(hint)

    return trace.finalize_using_last_hint(output=Graph(pointers={pi: ptrs}))
```

Figure E.1: A (simplified) presentation of algorithmic trace generation for value iteration.

Our improved NAP

Given these issues with the existing training pipeline, we instead train a (type-safe) NAP as per Section 3.4 to directly imitate value iteration over *deterministic MDPs with per-state rewards*:

$$\begin{aligned} V_0(s) &= 0 \\ V_{t+1}(s) &= R(s) + \gamma \max_{s' \to s} V_t(s') \end{aligned}$$

Specifically, we represent our MDP inputs as directed graphs (V, E), whose nodes correspond to states and whose edges $u \xrightarrow{a} v$ correspond to actions $a \in A$ taken from state v, with a per-state reward $R(s)$ stored on each node and the MDP discount factor γ stored on every edge. Given an input MDP, we train our NAP (supervising on algorithmic traces generated as in Figure E.1) to output the optimal value predictions $V^*(s)$ for all nodes s, alongside per-node pointers to the optimal state $s \xrightarrow{\pi^*(s)} t$.

Neighbourhood decoding

We resolve the problem of the decoder having insufficient information to predict the next action by performing *neighbourhood decoding* – in other words, allowing the decoder to project out predicted logits and value functions from both $\mathbf{s}'_0$ and its set of neighbour states $\{\mathbf{s}'_i \mid 0 \to i\}$.

§E.2 The Warcraft-Shortest-Path benchmark

We now turn to the design of our second benchmark: WARCRAFT-SHORTEST-PATH. This benchmark, inspired by the work of Vlastelica et al. [2019], allows for the comparison of algorithmic modules in the context of *finding the shortest-path tree for a Warcraft terrain map*; as such, in this section, we outline both the adaptations we made to the problem setting of Vlastelica et al. [2019], and the architecture we used to compare algorithmic modules on this problem.

§E.2.1 Adaptations made to prior work

As a general theme, as we're trying to explore the null hypothesis that NAPs perform better than sDABs, in order to minimise the risk of false positives, we typically choose design decisions that are favourable towards the NAP.

The original problem setting. Recall that we based the design of this problem on WARCRAFT-SHORTEST-PATH, a popular problem for benchmarking sDABs [Vlastelica et al., 2019, Berthet et al., 2020, Petersen et al., 2021]. This problem involved training a network (with an algorithmic inductive bias) to take as input an image of a $k \times k$ Warcraft terrain map, and return as output a $k \times k$ matrix indicating the cells of the terrain map involved in the *optimal shortest path* from the top left to the bottom right corner.

Exploring the problem in the context of Bellman-Ford. Note that, to solve this problem, we can use algorithmic modules with an AIB towards any single-source shortest path algorithm (e.g. Dijkstra [Vlastelica et al., 2019] or Bellman-Ford [Petersen et al., 2021]). As we wish to benchmark step-level NAPs against natively-differentiable sDABs, we choose to explore algorithmic modules with an AIB towards Bellman-Ford, a simple, natively-differentiable algorithm that aligns well with graph-based NAPs.

From shortest-path to shortest-path-tree. Observe, however, that to actually solve this problem, we need algorithmic modules that not only compute the minimum distance from the source to every other node, but also walk the resulting predecessor tree in $O(V)$ time in order to recover the actual shortest path from the top left to the bottom right cell. Now, while it is easy to add this postprocessing step to an sDAB, adding another $O(V)$ steps to a Bellman-Ford NAP could substantially impact performance. As such, to minimise the trajectory length of our NAP, and to align more closely with the version of Bellman-Ford used to train NAPs in the literature [Veličković et al., 2022], we avoid this postprocessing overhead by instead supervising on the *shortest path tree* of per-node predecessors rooted in the top-left grid cell.

Removing the inductive bias of algorithmic supervision. Now, in its original form, WARCRAFT-SHORTEST-PATH is a problem of *algorithmic supervision* [Petersen et al., 2021]: given an sDAB mapping a grid with weights to an indicator matrix representing the shortest path across it, we precompose this sDAB with a feature extractor (which should learn to predict a cost for each type of tile) and supervise directly on the shortest-path output of our sDAB. We note, however, that in most real-world problems with AIBs (which are ultimately where we want to apply NAR), we can't directly supervise on algorithmic outputs – instead, we must typically learn to *postprocess* (or project out relevant information from) the output of our algorithmic module. As such, we adapt our architecture by both pre-composing and post-composing our algorithmic module with learnable layers, and ensuring the outputs of our algorithmic modules require some mild post-processing in order to extract the final outputs.

§E.2.2 The Warcraft-Net architecture

So, given our adapted problem of WARCRAFT-SHORTEST-PATH-TREE, we now outline the architecture we built to solve it. We define this architecture as a function WARCRAFT-NET : $Grid[(8k, 8k), 3] \rightarrow Grid[(8k, 8k), Categorical[8]]$, mapping $8k \times 8k \times 3$ images of Warcraft terrain maps to grids of categorical variables indicating the predecessor cell for each cell in the grid.

We present a pseudocode implementation of the WARCRAFT-NET architecture in Algorithm 4.

A high-level summary of Warcraft-Net. In a similar vein to the architecture of XLVIN, our WARCRAFT-NET has four main components: *extracting a $k \times k$ grid of features* from the original image, *generating a latent graph* from these features, *applying an algorithmic module* to this latent graph, and *projecting out predecessor pointers* from this latent graph.

Extracting the grid of features. As per [Vlastelica et al., 2019], we extract features from our input image using the first five layers of *ResNet-18*. But while Vlastelica et al. [2019] use this feature extractor to directly predict a $k \times k$ grid of cell costs which they pass to their sDAB, as we wish to use our architecture with either sDABs or NAPs, we instead return a $k \times k$ grid $\mathbf{f} : Grid[(k,k), \mathcal{I}]$ of latent per-cell features.

Generating the grid graph. Given such a grid $\mathbf{f}$, in order to compute its shortest-path tree, we must first generate its underlying *latent grid graph*. This graph has a node (i,j) for every cell $\mathbf{f}_{ij}$ in the grid, and each node (i,j) has an out-edge to each neighbouring cell (i',j') (including those diagonally adjacent to (i,j)). Now, to decide how to populate nodes and edges with features, based on the type signature of the Bellman-Ford algorithm we want our nodes to carry information about whether or not their corresponding cell is the source node, and our edges to carry information about their cost of traversal. As such, we populate nodes (i,j) with node features $\mathbf{f}_{ij}$, and following Vlastelica et al. [2019], who synthesise their grid graph such that the weight of an edge $(i,j) \to (i',j')$ is the cost of the terrain type of cell (i',j'), we populate edges $(i,j) \to (i',j')$ with edge features $\mathbf{f}_{i'j'}$.

Executing the algorithmic module. Now, once we have a latent grid graph $g : G[\mathcal{I}, \mathcal{I}]$, we can apply a wrapped algorithmic module in order to execute Bellman-Ford over it. For this benchmark, we provide two algorithmic modules for comparison:

- **NAP:** A frozen, graph-based EPD NAP (pre-trained as described in Section 4.1), wrapped with linear encoders and decoders (Algorithm 5).
- **sDAB:** A scalar implementation of Bellman-Ford, wrapped with skip-connected linear encoders and decoders (Algorithm 6). Note that, when extracting initial distances from node features, we apply a scaled sigmoid to introduce an inductive bias towards either predicting $d_i^{(0)} = 0$ or $d_i^{(0)} = \infty$ (where $\infty \approx 100$ to avoid issues with numerical instability).

Predicting predecessor pointers. Finally, once we have our output graph $g : G[\mathcal{O}_n, \mathcal{O}_e]$, for each node in our graph, we attend to its in-edges (using node pointer decoding as in [Veličković et al., 2022]) to obtain a distribution over its 8 possible predecessors. We output these per-node predecessor distributions as a grid of per-cell categorical variables indicating the cardinal direction (e.g. North, North-East, East etc) of the predecessor cell.

Algorithm 4 A pseudocode implementation of WARCRAFT-NET for a $k \times k$ terrain map.

```
Require: The following learnable functions:
    $enc : \mathbb{R}^3 \to \mathcal{I} := Linear$
    $f_1, f_2 : \mathcal{O}_n \to \mathcal{P} := Linear$
    $f_e : \mathcal{O}_e \to \mathcal{P} := Linear$
    (alongside the first five layers of ResNet-18)

function RESNET-FEATURE-EXTRACTOR($\mathbf{img} : Grid[(8k, 8k), 3]) : Grid[(8k, 8k), \mathcal{I}]$
    $\mathbf{h} : Grid[(2k, 2k), 3] \leftarrow$ ResNet-18.(conv1 ▷ bn1 ▷ ReLU ▷ MaxPool ▷ layer1)($\mathbf{img}$)
    $\mathbf{h}' : Grid[(2k, 2k), \mathcal{I}] \leftarrow \text{map}(enc, \mathbf{h})$
    return $\text{maxPool}(\mathbf{h}', (2k, 2k) \to (k, k))$

function BUILD-GRAPH($\mathbf{h} : Grid[(k, k), \mathcal{I}]) : G[\mathcal{I}, \mathcal{I}]$
    $g \leftarrow \text{newGraph}(\text{nodes} = \{(y, x) \mid y, x \in [1..k]\})$
    for $(y, x) \in [1..k] \times [1..k]$ do
        $g.\text{nodes}[(y, x)] \leftarrow \mathbf{h}_{yx}$
        $g.\text{addEdges}(\{(y', x') \xrightarrow{\mathbf{h}_{yx}} (y, x) \mid (y', x') \in \text{getAdjacent8Cells}(y, x)\})$
    return $g$

function PREDICT-POINTERS($G(\{\mathbf{h}_i\}, \{\mathbf{e}_{ij}\}) : G[\mathcal{O}_n, \mathcal{O}_e]) : Grid[(k, k), Categorical[8]]$
    ▷ For each node, compute a weighted distribution over its in-edges.
    $\pi_{i \to j} : \mathbb{R} \leftarrow f_m(\max[f_1(\mathbf{h}_j), f_2(\mathbf{h}_i) + f_e(\mathbf{e}_{ij})])$

    ▷ Convert these per-node weighted distributions to a grid of categoricals
      representing the direction $\{N, NE, E, SE, S, ...\}$ of the pointed-to in-edge
    $preds_{yx} \leftarrow \text{softmax}([\pi_{(y', x') \to (y, x)} \mid (y', x') \in \text{getAdjacent8Cells}(i, j)])$
    return $preds$

function ALGORITHMIC-MODULE($g : G[\mathcal{I}, \mathcal{I}]) : G[\mathcal{O}_n, \mathcal{O}_e]$
    ▷ A NAP or sDAB, wrapped in appropriate encoders and decoders.

function WARCRAFT-NET($\mathbf{img} : Grid[(8k, 8k), 3]) : Grid[(k, k), Categorical[8]]$
    ▷ Extract a $k \times k$ grid of features from the terrain map.
    $\mathbf{h} \leftarrow$ RESNET-FEATURE-EXTRACTOR($\mathbf{obs}$)

    ▷ Generate latent grid graph from grid
    $g^{(in)} \leftarrow$ BUILD-GRAPH($\mathbf{h}_0$)

    ▷ Execute algorithmic module on grid graph
    $g^{(out)} \leftarrow$ ALGORITHMIC-MODULE($g^{(in)}$)

    ▷ For each node, identify its optimal predecessor(s) by attending to its in-edges.
    return PREDICT-POINTERS($g^{(out)}$)
```

Algorithm 5 A pseudocode implementation of a Bellman-Ford NAP (i.e. a wrapper around an EPD-NAP pre-trained as in Section 4.1), for use as an ALGORITHMIC-MODULE in WARCRAFT-NET (Algorithm 4).

```
Require: The following learnable functions:
    $enc_n : \mathcal{I} \rightarrow \mathcal{V}_n^{(in)} := Linear$        $dec_h : \mathcal{V}_n^{(out)} \rightarrow \mathcal{O}_n := Linear$
    $enc_e : \mathcal{I} \rightarrow \mathcal{V}_e^{(in)} := Linear$        $dec_e : \mathcal{V}_e^{(out)} \rightarrow \mathcal{O}_e := Linear$

1: function NAP($G(\{\mathbf{h}_i\}, \{\mathbf{e}_{ij}\}) : G[\mathcal{I}, \mathcal{I}]) : G[\mathcal{O}_n, \mathcal{O}_e]$
2:     ▷ Map into algorithmic latent space
3:     $\mathbf{h}_i^{(in)}, \mathbf{e}_{ij}^{(in)} \leftarrow enc_h(\mathbf{h}_i), enc_e(\mathbf{e}_{ij})$

4:     ▷ Execute EPD-trained NAP
5:     $G(\{\mathbf{h}_i^{(out)}\}, \{\mathbf{e}_{ij}^{(out)}\}) \leftarrow$ EPD-NAP($G(\{\mathbf{h}_i^{(in)}, \mathbf{e}_{ij}^{(in)}\})$)

6:     ▷ Map out of algorithmic latent space
7:     $\mathbf{h}_i^{(ret)}, \mathbf{e}_{ij}^{(ret)} \leftarrow dec_h(\mathbf{h}_i^{(in)}, \mathbf{h}_i^{(out)}), dec_e(\mathbf{e}_{ij}^{(in)}, \mathbf{e}_{ij}^{(out)})$

8:     ▷ Return graph
9:     return $G(\{\mathbf{h}_i^{(ret)}\}, \{\mathbf{e}_{ij}^{(ret)}\})$
```

Algorithm 6 A pseudocode implementation of a Bellman-Ford sDAB, for use as an ALGORITHMIC-MODULE in WARCRAFT-NET (Algorithm 4).

```
Require: The following learnable functions:
    $enc_w : \mathcal{I} \rightarrow \mathbb{R} := Linear$        $dec_h : \mathbb{R} \rightarrow \mathcal{O}_n := Linear$
    $enc_d : \mathcal{I} \rightarrow \mathbb{R} := Linear$        $dec_e : \mathcal{I} \rightarrow \mathcal{O}_e := Linear$

1: function sDAB($G(\{\mathbf{h}_i\}, \{\mathbf{e}_{ij}\}) : G[\mathcal{I}, \mathcal{I}]) : G[\mathcal{O}_n, \mathcal{O}_e]$
2:     ▷ Extract weights from edge features
3:     $w_{ij} : \mathbb{R} \leftarrow enc_w(\mathbf{e}_{ij})$

4:     ▷ Extract initial distances from node features
5:     $d_i^{(0)} : \mathbb{R} \leftarrow 100 \cdot \sigma(enc_d(\mathbf{h}_i))$

6:     ▷ Perform Bellman-Ford relaxations on grid graph
7:     for $t \leftarrow 1, ..., (k \times k)$ do
8:         $d_j^{(t)} = \min(d_j^{(t-1)}, \min_{i \rightarrow j}(d_i^{(t-1)} + w_{ij}))$

9:     ▷ Map final distances to latent space and return latent graph
10:    return $G(\{dec_h(d_i^{(k \times k)})\}, \{dec_e(\mathbf{e}_{ij})\})$
```

F Hyperparameters and experimental details

In this appendix, we present hyperparameters and experimental details for each of the experiments we ran.

§F.1 Evaluating correctness: training NAPs on the CLRS-30 benchmark

Evaluation metric. For both algorithms tested (BF and DFS), we use the evaluation metric of **predecessor pointer accuracy** from the CLRSB. Observe that we can frame the algorithms of BF and DFS as generating *spanning trees (or forests)* of their input: BF generates a *shortest path tree*, where an edge $u \to v$ is included in the tree iff it is part of some shortest path $s \to^* u \to v$, and DFS generates a *predecessor forest*, where an edge $u \to v$ is included in the tree iff it is explored by the DFS algorithm. So, for ease of evaluation, the CLRSB defines the output of BF and DFS to be the parent arrays of their respective predecessor subgraphs (i.e. for every node v, we predict a pointer to its parent node u, or to itself if it has none) and scores BF and DFS performance based on the (per-node) accuracy of these predicted predecessor pointers.

Hyperparameters and training. We adopted the hyperparameters and input data distribution of [Ibarz et al., 2022], with the exception of fixing our graph size to $n = 16$ as opposed to sampling this from a distribution.[1]

For each processor and each algorithm tested, we performed 5 training runs with different seeds. For each run, we trained for 10,000 steps on (continuously-sampled) batches of 32 trajectories with graph size 16, using a validation set of 32 trajectories with graph size 64 to choose the highest-performing checkpoint for evaluation. We then evaluated the highest-performing checkpoint of each run on the official CLRSB validation and test datasets (which we converted to work with the MDARF), assessing in-distribution (graph size 16) and out-of-distribution (graph size 64) performance respectively.

§F.2 Evaluating utility for reproduction: exploring the VI-Implicit-Planner benchmark

§F.2.1 Training NAPs for value iteration

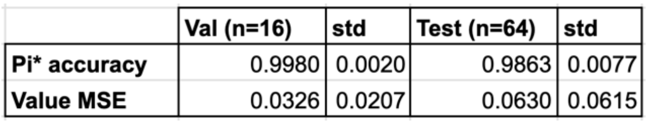
Input distribution. Our algorithmic inputs for this problem consist of deterministic MDPs with per-state rewards, represented as directed graphs (V, E) (whose nodes correspond to states and whose edges $u \xrightarrow{a} v$ correspond to actions $a \in A$ taken from state v), with a per-state reward $R(s)$ on each node, and the MDP discount factor γ on every edge. We randomly generate MDPs of size n by sampling $|A| \sim \mathit{Unif}[1, n-1]$, $\gamma \sim \mathit{Unif}[0.1, 0.9]$, $R(s) \sim \mathcal{N}(0, 1)$, and for each node u, choosing a random subset of $V \setminus \{u\}$ of size $|A|$ as the next states for actions out of u.

[1] While we also tested the graph size distribution used in [Ibarz et al., 2022] (i.e. cycling through sizes $[4, 7, 11, 13, 16]$), we found this did not confer the performance improvements observed by Ibarz et al. [2022]. This is likely due to the fact that the GNAL uses fully-connected graphs, and is hence at risk of overfitting to nodes with a particular in-degree if trained on fixed-size graphs – but because we don't use fully-connected graphs, we're not susceptible to this issue.

Architecture and hyperparameters. We follow the standard EPD pipeline, as detailed in Section 3.4. Note that, for our executor, we use a slightly older version of our MPNN: we implemented the Triplet-GMPNN later on in the course of the project, but due to resource constraints, we were unable to rerun all experiments in this section to adapt to this change. Note also that, due to resource limitations, we were only able to conduct 3 training runs for this experiment. All other hyperparameters and training conditions are as specified in Section 4.1.1.

Metrics and evaluation. Recall that our algorithmic outputs for this problem consist of, for each node s, an *optimal value* prediction $V^*(s)$, and a pointer to the next state t given by $s \xrightarrow{\pi^*(s)} t$. We score VI performance based on mean-squared error between predicted and ground-truth values averaged across every time-step of VI, and pointer accuracy between predicted and ground-truth optimal next states, aggregating these metrics as described in Section 4.1.

Results and discussion. Observe from Figure F.1 that our trained value-iteration NAP learns to predict the optimal next state with near-perfect accuracy both in and out of distribution. As such, for our subsequent experiments, we chose to use the trained NAP with the best out-of-distribution accuracy for $\pi^*(s)$.



	Val (n=16)	std	Test (n=64)	std
Pi* accuracy	0.9980	0.0020	0.9863	0.0077
Value MSE	0.0326	0.0207	0.0630	0.0615

Figure F.1: A table of aggregate performance metrics for our trained value-iteration NAPs, evaluated on both in-distribution (graph size 16) and out-of-distribution (graph size 64) data. The checkpoint we selected for XLVIN had in-distribution accuracy 0.9961 / MSE 0.0115, and out-of-distribution accuracy 0.9917 / MSE 0.0174.

§F.2.2 Training XLVIN

Architecture and hyperparameters. Besides the changes described in Appendix E.1, unless otherwise stated, we use the architecture and hyperparameters reported by the reproduction of He [2022].[2]

Training and metrics. Recall from Appendix B that, in (policy-based) RL, the training process consists of repeatedly using our current policy to sample a batch of trajectories, using these to estimate the policy gradient, and updating the model's weights based on this estimate. For each model we explored, we assessed it over 20 training runs k, each initialised with a different random seed; to ensure a low-data regime, each run took place over 20 batches of 5 trajectories each. As per [Deac et al., 2021] and [He, 2022], after each batch i of training run k, we sampled 100 test trajectories and computed the average reward $r_{k,i}$ obtained across all trajectories. We then report $r_k := \max_i r_{k,i}$ as the **per-run maximum average reward** obtained across run k.

§F.3 Evaluating utility for research: exploring the Warcraft-Shortest-Path benchmark

§F.3.1 Details of the *optimal* and *tiebreaking* dataset variants

For comparability, we base our dataset on that of [Vlastelica et al., 2019], consisting of 10,000 training, 1,000 validation and 1,000 test images of randomly-generated terrain maps from the

[2]Note that, in keeping with [Deac et al., 2021] we do not pre-train the TransE encoder and transition model, but, in keeping with [He, 2022], we apply the TransE loss to our PPO baseline.

Warcraft II tileset [Guyomarch, 2017].

Now, we observe that these terrain maps do not always have unique shortest paths (let alone shortest path trees). Indeed, while Vlastelica et al. [2019] simply ignore this issue,[3] we handle it by adapting our training and evaluation metrics accordingly.

Indeed, we explore two different ways of dealing with non-unique predecessors:

Optimal variant. For each grid cell, we train via cross-entropy loss to predict a distribution over its predecessors, with uniform weight over all optimal predecessors, and zero weight everywhere else. We evaluate our models based on both **optimal accuracy** (i.e. the percentage of predicted predecessor distributions which have an optimal pointer as their mode), and **optimal tree-accuracy** (i.e. the percentage of grids for which all predicted predecessor distributions in that grid have an optimal pointer as their mode).

Tie-breaking variant. For each grid cell, we train via cross-entropy loss to predict the optimal predecessor, breaking ties by priority (with the highest priority predecessor being the eastern cell, and priority decreasing clockwise). We evaluate our models based on **exact accuracy** (i.e. the percentage of correctly-predicted predecessors), and **exact tree-accuracy** (i.e. the percentage of trees with correctly-predicted predecessors).

As neither our sDAB nor our NAP are designed to break ties in the manner described above[4], while the optimal problem variant is easily solvable with only information about per-node distances, to solve the tiebreaking variant, our NAPs and sDABs must need to learn to distinguish between different nodes with the same shortest path length. As such, comparing performance across these two environments lets us explore the claim that *NAR confers a particular advantage in the case where the underlying algorithm of our problem differs slightly from our algorithmic prior.*

§F.3.2 Experimental details

Executors. Recall that the problem of WARCRAFT-SHORTEST-PATH-TREE has an algorithmic prior towards Bellman-Ford. As such, in the following experiments, we compare the relative performance of WARCRAFT-NET (Appendix E.2) when equipped with various executors with an algorithmic inductive bias towards Bellman-Ford – specifically, the Bellman-Ford sDAB (as described in Section E.2), and a Bellman-Ford NAP (as trained in Section 4.1). For our experiments, we chose to use the NAP with the highest test pointer accuracy; this NAP achieved a pointer accuracy of 0.9941 in-distribution (on graphs of size 16), and a pointer accuracy of 0.9561 out-of-distribution (on graphs of size 64).

Hyperparameters and training. For each model, across each of our two environments, we performed 5 training runs with different seeds. For each run, we adopted the hyperparameters and training regime of Vlastelica et al. [2019], training for 50 epochs with batch size 70 and learning rate 5×10^{-4}, and evaluating on a validation set of size 1,000 after every epoch to choose the highest-performing checkpoint for evaluation. We then evaluated the highest-performing checkpoint of each run on the test sets of [Vlastelica et al., 2019], assessing both in-distribution

[3]Note that, during training, Vlastelica et al. [2019] select a particular shortest path to supervise on for each map, and during evaluation, they score models based on whether or not their predicted shortest path is optimal. This can potentially lead to performance issues if the shortest path on which we supervise is selected in a deterministic way: in particular, the model may try to learn the algorithm for choosing the exact shortest path in specific cases, at the cost of decreasing its overall performance at choosing an optimal path.

[4]Indeed, our sDAB only outputs per-node shortest path lengths, and while the NAP we use (i.e. the Bellman-Ford NAP from Section 4.1) was trained with supervision on both per-node shortest path lengths and predecessor pointers, as all edges had weights uniformly randomly sampled from $[0, 1]$, it never learned to perform tie-breaking.

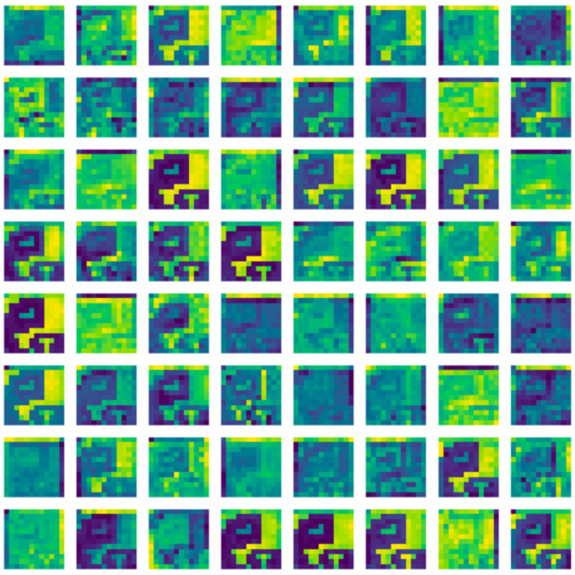
(tree size 12×12) and out-of-distribution (tree size 18×18) performance in terms of either tree-accuracy or optimal tree-accuracy as appropriate.[5]

Note that, while in principle we should run our executors for $|V| = 144$ steps, in order for Bellman-Ford to converge, we need only apply our executors for n steps, where n is the maximum number of edges in any shortest path from the root node. As such, due to compute limitations, we only apply our executors for 45 steps, the maximum number of edges in any shortest path from the root node across all our training and test data.

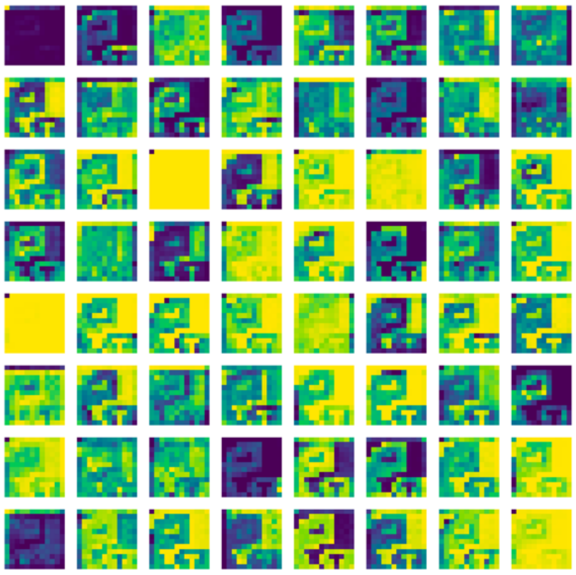
Performance evaluation. For this experiment, as in Section 4.2, we report model performance through bootstrapped 95% CIs for mean tree-accuracy (or optimal tree-accuracy), and we compare models through bootstrapped 95% CIs for probability-of-improvement. Observe that, as we collect $n = 15$ runs, we have $\binom{15+15-1}{15} = 7.8 \times 10^7$ possible bootstrap resamples, so we have sufficient data for bootstrap resampling to be meaningful.

[5]Note that, due to limitations of the ResNet-18 architecture, as per [Vlastelica et al., 2019], we are unable to evaluate its out-of-distribution performance.

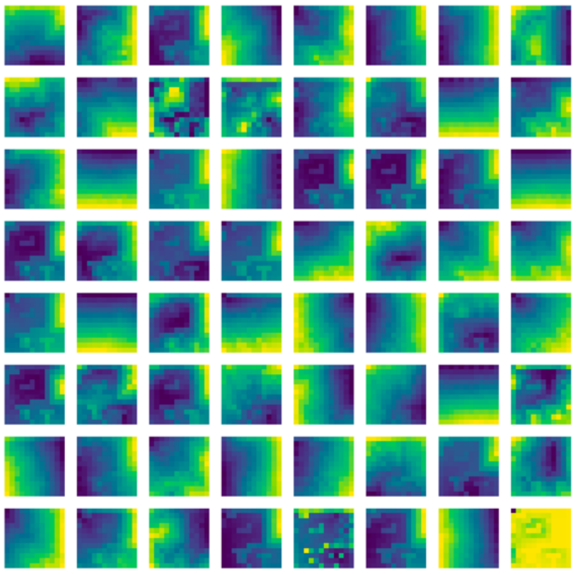
G An illustration of weight matrices from a learned pDAB-Warcraft-Net



(a) Initial weight matrices (w_{ij})



(b) Initial distance matrices ($d_{ij}^{(0)}$)



(c) Final distance matrices ($d_{ij}^{(k\times k)}$)

Figure G.1: The inputs and outputs to each of the 64 Bellman-Ford instances within our pDAB, when run on the Warcraft terrain map analysed in Section 4.3.4, presented as (individually-scaled) heatmaps. Each cell (i, j) corresponds to the learned input / output for the $8i + j$th Bellman-Ford instance in our pDAB.