Machine Learning
https://doi.org/10.1007/s10994-024-06538-7

Learning explanatory logical rules in non-linear domains: a neuro-symbolic approach

Andreas Bueff[1] · **Vaishak Belle**[1]

Received: 23 May 2022 / Revised: 4 February 2024 / Accepted: 5 March 2024

Abstract
Deep neural networks, despite their capabilities, are constrained by the need for large-scale training data, and often fall short in generalisation and interpretability. Inductive logic programming (ILP) presents an intriguing solution with its data-efficient learning of first-order logic rules. However, ILP grapples with challenges, notably the handling of non-linearity in continuous domains. With the ascent of neuro-symbolic ILP, there's a drive to mitigate these challenges, synergising deep learning with relational ILP models to enhance interpretability and create logical decision boundaries. In this research, we introduce a neuro-symbolic ILP framework, grounded on differentiable Neural Logic networks, tailored for non-linear rule extraction in mixed discrete-continuous spaces. Our methodology consists of a neuro-symbolic approach, emphasising the extraction of non-linear functions from mixed domain data. Our preliminary findings showcase our architecture's capability to identify non-linear functions from continuous data, offering a new perspective in neural-symbolic research and underlining the adaptability of ILP-based frameworks for regression challenges in continuous scenarios.

Keywords Inductive logic programming · Neuro-symbolic artificial intelligence · Knowledge representation and reasoning · Non-linear modelling · Deep learning

1 Introduction

State-of-the-art deep neural networks, while powerful, face inherent limitations like the need for vast training examples and a deficiency in generalisation and interpretability (Chollet, 2019). Inductive logic programming (ILP) offers a promising alternative with over three decades of history, emphasising data-efficient methods to learn first-order logic

Editor: Andrea Passerini.

✉ Andreas Bueff
andreas.bueff@ed.ac.uk

Vaishak Belle
vbelle@ed.ac.uk

[1] School of Informatics, The University of Edinburgh, 11 Crichton Street, Edinburgh EH8 9LE, UK

Published online: 08 April 2024

rules, which has the potential for greater generalisability (d'Avila Garcez et al., 2019; Muggleton & De Raedt, 1994). However, the ILP approach is not without its challenges, such as handling non-linearity and continuous properties, the computational intensity of predicate rule generation, and dependence on expert-defined background knowledge (Cropper & Dumančic, 2020).

The recent surge in neuro-symbolic ILP has aimed to establish hybrid frameworks that address these challenges, enabling features like noise handling, recursion-based generalisation, and supporting predicate invention. Such innovations have paved the way for research combining deep learning with relational ILP models, which not only enhances interpretability but also positions decision boundaries using logical rules (Evans & Grefenstette, 2017; Payani & Fekri, 2019; Yang & Song, 2019). However, the intricacy of addressing non-linearity in continuous domains remains an underexplored area.

Our work embarks on this challenge, aiming to design a neuro-symbolic ILP framework tailored for modelling non-linearity and continuous properties. Using the differentiable Neural Logic (dNL) networks as a foundation (Payani & Fekri, 2019), we focus on extracting non-linear rules in mixed discrete-continuous spaces, or more concretely, *Our aim is to design an ILP-based framework specifically for learning in a mixed discrete-continuous space for the purpose of non-linear function extraction*. This task necessitates adapting current systems to efficiently derive non-linear rules from data reflecting non-linear functions specified in advance.

Understanding the significance of our proposed framework becomes clearer when considering its application to real-world problems. Take, for instance, the domain of physics: if we were to gather data from a device measuring the relationship $E = m \times c^2$, the iconic energy-mass equation. The extended capabilities of ILP, with its focus on non-linear continuous predicates, could intuitively model such a relationship. This is accomplished by teaching the system to recognise operations like squaring a variable (c^2) and multiplying variables ($m \times c^2$). With this, we are not just adding to the toolset of data analysis; we are providing a path for data-efficient neuro-symbolic methods that can further scientific discoveries by extracting and understanding non-linear equations.

We introduce a three-step methodology, commencing with continuous data discretisation and the learning of variable transformations using differentiable Neural Logic Non-Linear (dNL-NL) networks, repurposed from the standard dNL. The second phase involves utilising a separate (dNL-NL) network to represent operational predicates like addition or multiplication. Each dNL-NL network aims to define transformations, using the provided non-linear predicates. A subsequent dNL-NL module extracts operational predicates between individual features, utilising an augmented dataset transformed by the initial dNL-NL module's non-linear predicates. The first and the second modules are end-to-end differentiable but the same cannot be said about the overall pipeline. To ensure accurate non-linear function extraction, we calculate the true loss on the continuous target using the compiled non-linear function from the extracted rules. This approach centres on extracting non-linear functions from mixed domain data.

While the current study focuses on supervised learning, future endeavours might extend this proposal to dynamic contexts, such as discerning state dynamics within reinforcement learning environments. Our findings showcase our architecture's ability to retrieve non-linear functions from continuous data where clausal rules describe the function. This venture unveils a unique direction in neural-symbolic research, suggesting that ILP-based frameworks are adaptable to continuous data, addressing regression challenges. To our knowledge, this proposed work is the first to focus on the differential logical learning of non-linear functions.

2 Related work

Logic programs, by their very nature, offer a high degree of interpretability. As models become increasingly complex, there is a growing need in many domains for models that can be understood, validated, and trusted by humans (Barredo Arrieta et al., 2020). Logic programs allow for the symbolic representation of knowledge, which can be useful in scenarios where domain knowledge needs to be incorporated or where explanations are required in symbolic form (Hitzler & Sarker, 2022). While neural networks and kernel machines are powerful function approximators, they often act as black-box models. The trade-off between accuracy and interpretability is a well-known challenge (Barredo Arrieta et al., 2020). Our approach seeks to bridge this gap by providing a mechanism to learn non-linear relationships in data while retaining the interpretability of symbolic methods.

The foundational principles of ILP, especially as detailed in Muggleton and de Raedt (1994), played a significant role in guiding this research. Classic ILP systems excel at learning logic-based rules and first-order logical predicates from structured, discrete, and relational data. However, they struggle when faced with non-linear predicates in continuous domains. Several classical ILP systems, including FOIL (Quinlan, 1990), Progol (Muggleton, 1995), Claudien (De Raedt & Dehaspe, 1997), Aleph (Srinivasan, 2001), XHAIL (Ray, 2009), and Atom (Ahlgren & Yuen, 2013), have showcased their proficiency in handling noisy data and addressing problems within infinite domains. While they are adept at processing certain recursive rules, they do not consistently deliver optimal results and lack support for predicate invention. To illustrate a specific challenge, Aleph's heuristic search, with its greedy nature, runs the perpetual risk of entrapment in local optima. Notably, these systems harness the set covering algorithm, learning hypotheses one clause at a time.

Another approach to note is Metagol (Muggleton et al., 2018), which, in contrast to its predecessors, boasts capabilities in predicate invention, recursion handling, and producing optimal programs across infinite domains. One disadvantage of Metagol is its inability to manage noise. Newer systems, such as Popper (Cropper & Morel, 2021a) and Poppi (Cropper & Morel, 2021b)-which builds upon Popper to enable automatic predicate invention-fill this gap. They adeptly handle noise, predicate invention, recursion, and can craft optimal programs within infinite domains (Cropper & Morel, 2021a, b).

ILP systems are inherently designed for symbolic representations, specifically for relational data in discrete domains. They are built to identify patterns using logical predicates. This design makes them unsuitable for continuous domains, which rely on numerical values and mathematical operations, differing greatly from ILP's logic-based foundation. Tackling non-linear predicates in these domains is challenging due to the complex mathematical computations and the vast function space it requires to search, which is computationally intensive.

Early ILP research was primarily focused on reasoning within discrete spaces, with notable contributions from Chavira and Darwiche (2008); Kersting et al. (2000); Richardson and Domingos (2006); Kimmig et al. (2012). A shift towards exploring mixed discrete-continuous spaces was evident in works such as Speichert and Belle (2018), where piecewise polynomials were harnessed to model continuous distributions, subsequently informing target predicate definitions. In a parallel vein, Nitti et al. (2016) proposed a probabilistic approach, leveraging Gaussian base atoms to derive rules.

Non-linear predicates present clear challenges in traditional ILP systems. This has led to growing interest in neuro-symbolic methods for better modelling in continuous domains. Our proposed method introduces a new ILP technique, learning from magic values with

lazy evaluation. A magic value in a program refers to a constant symbol vital for the program's proper execution, even if its selection lacks a clear rationale (Hocquette & Cropper, 2023). In our system, the lower and upper bound weights (detailed in Sect. 4.1) for the non-linear and operation predicates serve as these magic values. While these bounds ultimately become constant symbols in predicate definitions post-training, they start as trainable parameters. Lazy evaluation, as adopted by Aleph for constant refinement, is also utilised in our method (Srinivasan & Camacho, 1999). Aleph refines bottom clauses by seeking variable substitutions and executing a partial hypothesis on both positive and negative examples. Essentially, rather than exploring all constant symbols, lazy evaluation focuses solely on symbols derived from the examples. Similarly, we evaluate our rules using evolving lower/upper bounded weights and predicate membership weights, making our approach aligned with the principles of lazy evaluation. Our work expands ILP's learning capabilities to cover large and unbounded domains, and our tests show its advantages over previous systems.

Recent advancements in the ILP field have embraced a neuro-symbolic approach, as highlighted by Evans and Grefenstette (2017). This study introduced dILP, a novel neural ILP solver with a differentiable architecture for deduction. The emergence of such neuro-symbolic ILP systems has spurred a trend in benchmarking ILP based on various features, including noise resilience, compatibility with infinite domains, recursion, and predicate invention. As this field matures, richer feature sets will be introduced, setting the stage for more nuanced evaluations and progress benchmarking. Neuro-symbolic ILP methods, while presenting numerous enhancements, have not prioritized learning non-linear predicates in continuous domains. In contributing to this narrative, our work introduces the modelling of non-linearity in continuous or mixed domains, broadening the comparative landscape.

Neuro-symbolic ILP has seen various advancements, with several noteworthy contributions pushing the boundaries of the field. Shindo et al. employ differentiable logic modules that softly compose logic programs. Instead of utilizing MLPs, they manage multiple clauses with function symbols to enhance interpretability. Additionally, they incorporate predicate operations such as negation and preservation, which bolster flexibility (Shindo et al., 2021). This method's differentiable aspects, such as tensor encoding and inference, function on discrete logic symbols and their respective truth values. Sen et al. expanded upon logical neural networks to derive rules in first-order logic. They demonstrate the joint learning of rules and logical connectives (Sen et al., 2021). The flexibility of their learning algorithm accommodates various linear inequality and equality constraints. Owing to adaptable parametrisation, their approach outperforms others on multiple benchmarks. Krishnan et al. build upon the differentiable ILP framework introduced by Evans et al. Their primary objective is to learn recursive programs and conduct predicate invention with stratified and safe negation (Krishnan et al., 2021).

Addressing the learning challenge in our work, we acknowledge parallels in other research that prioritise distinct component learning for data modelling, such as the approach detailed in Duvenaud et al. (2013). They innovatively redefine kernel learning as a structure discovery challenge, automating kernel form choices and composing kernel structures using base components. This approach offers an expressive modelling language, capturing widely-used kernel construction methods. Their emphasis on Gaussian process regression with the kernel as a covariance function leverages the Bayesian framework for streamlined structure discovery, using marginal likelihood for evaluation.

In this initial study, the focus has been on establishing the dNL-NL model as a foundational approach for extracting non-linear equations from continuous data within a

regression framework. This work serves as a proof of concept, laying the groundwork for future research which will aim to scale the model and undertake comprehensive comparative evaluations with other neuro-symbolic approaches to further validate its applicability and effectiveness. Given that, we note other models which could be considered in evaluating future iterations of the dNL-NL architecture in program synthesis with continuous domains. TerpreT, a probabilistic programming language, integrates neural networks with traditional search techniques in Inductive Program Synthesis (IPS) (Gaunt et al., 2016). It uses models inspired by compiler representations, which are trainable via gradient descent, enabling the handling of complex control flows and external storage interactions. TerpreT's unique architecture supports defining execution models like Turing Machines, using parameterized programs and interpreters. It accommodates a variety of back-end inference algorithms, facilitating the synthesis of interpretable source code with intricate control structures. This setup not only aids in learning complex programs but also permits comparisons across different inference techniques and representations.

Building on this foundation, DeepCoder emerges as an innovative IPS method, harnessing neural networks to decode patterns in problem descriptions for guiding search-based synthesis (Balog et al., 2017). DeepCoder redefines IPS as a big data challenge, training extensively on IPS problems. Its framework establishes a versatile programming language, easily predictable from input–output examples, and devises models to link these examples to program attributes. This leads to considerable speed enhancements in program synthesis, particularly for the complex problems typical in competitive programming. DeepCoder's use of machine learning improves the efficiency and effectiveness of program synthesis by predicting program attributes and influencing the synthesis process with neural network insights.

Other approaches to program synthesis using elements of non-linear bias include DreamCoder (Ellis et al., 2020). DreamCoder is an innovative program learning system that specializes in program induction across multiple domains, utilizing self-supervised learning, bootstrapping, and domain-specific languages. It employs a unique "Wake/Sleep" architecture for program induction, combining generative models and neural networks to efficiently synthesize and refactor programs. DreamCoder stands out for its ability to discover specialized abstractions, enabling the expression of complex solutions to tasks at hand and achieving significant advancements in the field of program learning.

3 Background

3.1 ILP

Inductive logic programming (ILP) is a method of symbolic computing which can automatically construct logic programs provided a background knowledge base (KB) (Muggleton & de Raedt, 1994). An ILP problem is represented as a tuple $(\mathcal{B}, \mathcal{P}, \mathcal{N})$ of ground atoms, with $\mathcal{B}$ defining the background assumptions, $\mathcal{P}$ is a set of positive instances which help in defining the target predicate to be learned, and $\mathcal{N}$ defines the set of negative instances of the target predicate. The aim of ILP is to eventually construct a logic program that explains all provided positive sets and rejects the negative ones. Given an ILP problem $(\mathcal{B}, \mathcal{P}, \mathcal{N})$, the aim is to identify a set of hypotheses (clauses) $\mathcal{H}$ such that (Muggleton & de Raedt, 1994):

- $\mathcal{B} \wedge \mathcal{H} \models \gamma$ for all $\gamma \in \mathcal{P}$
- $\mathcal{B} \wedge \mathcal{H} \not\models \gamma$ for all $\gamma \in \mathcal{N}$

Where $\models$ denotes logical entailment. Thus stating, that the conjunction of the background knowledge and hypothesis should entail all positive instances and the same should not entail any negative instances. We assume for example a KB with provided constants $\{bob, carol, volvo, jacket, pants, skirt, \cdots\}$, where the task is to learn the predicate $\texttt{Passenger}(X)$. Then the ILP problem is defined as:

- $\mathcal{B} = \{\texttt{Car}(ford), \texttt{Clothing}(jacket), \texttt{On}(jacket, bob), \texttt{Inside}(carol, volvo), \cdots\}$
- $\mathcal{P} = \{\texttt{Passenger}(bob), \texttt{Passenger}(carol), \cdots\}$
- $\mathcal{N} = \{\texttt{Passenger}(volvo), \texttt{Passenger}(jacket), \cdots\}$

The outcome of the induction performed is a hypothesis of the form:

$$\texttt{Passenger}(X) \leftarrow \texttt{Inside}(X, Y_1) \wedge \texttt{Car}(Y_1) \wedge \texttt{On}(Y_2, X) \wedge \texttt{Clothing}(Y_2)$$

The learned first-order logic rule from the KB states *"if an object is inside the car with clothing on it, then it is a passenger"*.

The ILP problem may also contain a language frame $\mathcal{L}$ and program template Π (Evans & Grefenstette, 2017). The language frame is a tuple which contains information on the target predicate, the set of extensional predicates, arity of each predicate, and a set of constants, while the program template describes the range of programs that can be generated.

The placement of ILP in the context of non-linearity is that predicate rules can be equated with a non-linear function. For example, consider the equation for the mass-energy equivalence $E = m \times c^2$, which takes in as input mass (m) and multiplies it by the square of the constant for the speed of light (c). For the sake of the example, we treat both m and c as random variables within the range [0, 1). As non-linear equations produce continuous output, we can discretise the output into specified ranges. Discretising the continuous output transforms the non-linear regression problem into a classification problem, aligning with the discrete reasoning of ILP. By discretising the output of a function, such as $E = m \times c^2$ into distinct intervals, we can also interpret the distinct intervals as level sets on the equation. These level sets, which can be arbitrarily shaped and even disconnected, correspond to the range within specific bins and serve as target predicates for ILP to learn. Associating each FOL rule with specific level sets enriches our understanding of the function's representation, clarifying the relationship between logic rules and the function's behaviour.

The ILP problem would then focus on learning target predicates which represent an output range, here $\texttt{Class}_1(m, c)$ maps to the output range $(0 \leq E \leq 3.07 \times 10^{-1})$. As before, the tuple of ground atoms $(\mathcal{B}, \mathcal{P}, \mathcal{N})$ would contain background assumptions but in the non-linear context, the predicates in the KB would be associated with inequalities (`LessThan`), inequalities on transformed input (`SquareLessThan`), and inequalities for operations between variables such as taking the product between two variables (`ProdLessThan`). Assuming a KB with well-defined predicates consisting of operations and transformations on continuous data, a hypothesis for the first class can be induced:

$$\texttt{Class}_1(m, c)_{(0 \leq E \leq 3.07 \times 10^{-1})} \leftarrow \texttt{LessThan}(m, 0.6) \wedge \texttt{SquareLessThan}(c, 0.5) \wedge \texttt{ProdLessThan}(m, c^2, 0.3)$$

Here, the logic rule for the classification can be interpreted as *"if m is less than 0.6 and the square of c is less than 0.5 and the product between m and c^2 is less than 0.3, then the*

value of E will be greater than or equal to 0 *and less than or equal to* 3.07×10^{-1}". The interpretation, while distant from the original non-linear function, can be parsed to derive the true equation. The inequalities on the continuous values assist in defining the discrete range associated with our $\texttt{Class}_1$ predicate, but the associated transformations (c^2) and operations ($m \times c^2$) lend themselves to defining the non-linear equation for E. We note that the reliability of the non-linear and operation predicate rules we derive hinges on the granularity of the target's discretisation. A sufficiently fine-grained discretisation ensures the robustness of the learned rules, as a low discretisation scheme risks overgeneralising the non-linear output.

3.2 dNL

Initially, ILP was constrained to program induction on non-noisy symbolic data. While data-efficient, traditional ILP was limited in its applications. In Evans and Grefenstette (2017), ILP was bridged with neural networks, to create an end-to-end differentiable architecture which could learn on noisy data. The core ideas have been used in subsequent proposals, including the approach presented here. Payani et al. proposed a further extension of ILP called a differentiable neural logic (dNL) network which utilises differentiable neural logic layers to learn Boolean functions (Payani & Fekri, 2019), building upon ideas proposed by Evans et al. The concept is to define Boolean functions that can be combined in a similar cascading architecture akin to neural networks. This gives deep learning an explicit symbolic representation that is interpretable. It also redefines ILP as an optimisation problem. The dNL architecture uses membership weights and conjunctive and disjunctive layers with forward chaining to remove the need for the rule template to solve ILP problems.

In the construction of the logical framework, Boolean values ($true = 1, false = 0$) are mapped to real value ranges [0, 1]. Payani et al. define fuzzy unary and dual Boolean functions of variables x and y as follows:

- $\overline{x} = 1 - x$
- $x \wedge y = xy$
- $x \vee y = 1 - (1 - x)(1 - y)$

The core component of the dNL network is their use of differentiable neural logic layers to learn Boolean functions (Payani & Fekri, 2019). The dNL architecture uses membership weights and conjunctive and disjunctive layers to learn a target predicate or Boolean function. Learning a target predicate p requires the construction of Boolean function $\mathcal{F}_p$ which passes in a Boolean vector $\mathbf{x}$ of size N with elements $x^{(i)}$, into a neural conjunction function f_{conj} (see Eq. 1a) which is defined by a conjunction Boolean function F_{conj} (see Eq. 1a).

$$f_{conj}(\mathbf{x}) = \prod_{i=1}^{N} F_{conj}(x^{(i)}, m^{(i)}) \tag{1a}$$

$$F_{conj}(x^{(i)}, m^{(i)}) = \overline{\overline{x^{(i)}} m^{(i)}} = 1 - m^{(i)}(1 - x^{(i)}) \tag{1b}$$

A predicate defined by Boolean function in this matter is extracted by parsing the architecture for membership weights ($w^{(i)}$) above a given threshold, where membership weights are converted to Boolean weights via a sigmoid $m^{(i)} = \sigma(cw^{(i)})$ with constant $c \geq 1$. The

constant c effectively acts as a "scaling factor" for the sigmoid's argument. As the constant is greater than 1, it makes the sigmoid curve steeper. This means that the transition from the lower asymptote to the upper asymptote of the sigmoid function occurs over a narrower range of input values. Membership weights are paired with continuous lower and upper bound predicate functions (discussed in Sect. 3.4) which are eventually interpreted as atoms in the body of the predicate being learned. These same Boolean predicate functions are used to transform non-Boolean data into a Boolean format for the logic layers.

Similarly, a neural disjunction function f_{disj} (see Eq. 2a) can be constructed using the disjunction Boolean function F_{disj} (see Eq. 2a).

$$f_{disj}(\mathbf{x}) = 1 - \prod_{i=1}^{N}(1 - F_{disj}(x^{(i)}, m^{(i)})) \tag{2a}$$

$$F_{disj}(x^{(i)}, m^{(i)}) = x^{(i)}m^{(i)} \tag{2b}$$

By combining different neural Boolean functions, a multi-layered structure can be created. For example, cascading a conjunction function with a neural disjunction function (see Eq. 2a) creates a layer in disjunctive normal form (DNF), so-called dNL-DNF. Alternatively, cascading a disjunctive function with a neural conjunction function reinterprets the architecture in conjunctive normal form (CNF), forming a dNL-CNF.

Each rule to be learned corresponds to a dNL-DNF (or dNL) function, a differentiable symbolic Boolean function with parameterized membership weights in its conjunction and disjunction layers. A rule's body is represented by a Boolean dNL function, determined by membership weights in its neural conjunction and disjunction functions, given by $\mathcal{F}_p \leftarrow f_{disj}(f_{conj}(\mathbf{x}))$. The conjunction layer's membership weights, $m^{(i)}$, relate to specific Boolean inputs $x^{(i)}$ from vector $\mathbf{x}$, signifying the presence or absence of a Boolean feature. Meanwhile, the disjunction layer's membership weights, which are separate weights from the conjunction layer, map to the conjunction layer's rows, offering multiple definitions for rule p.

The optimization of the model is performed by evaluating the extracted rules' membership weights, where evaluation is done by applying the extracted rules to the background knowledge so that negative examples are rejected and positive examples are entailed. The weights are optimized based on the loss functions detailed in Sect. 3.5. As we extend to the continuous case where inputs are continuous and the target is discrete, each class in a tabular dataset is associated with a target predicate Boolean function $\mathcal{F}_p$, which is defined by bounded continuous Boolean predicates derived from each continuous feature, as explained in Sect. 3.4.

3.3 Notation

In this proposal, mathematical equations play a vital role in detailing our approach. To ensure clarity, we provide tables that list and define the crucial hyperparameters and notations. Please refer to Table 2 for hyperparameters and Table 1 for the specific notations employed throughout this proposal.

We also clarify, that for a given instance i, we define a discrete output label y_i, and our set of continuous feature values $\mathbf{x}_i \leftarrow \{x_i^{(1)}, x_i^{(2)}, \cdots, x_i^{(m)}\}$ representing the data entry for our label y_i. We note that the $x^{(i)}$ notation is useful in understanding the input with respect to Eqs. (1a and 2a), however, we will use the following notation to refer to a continuous

Table 1 Notation and definitions for implemented equations

Notation	Definition
X_i	Random variable
$\{X_1, \cdots, X_m\}_i$	Data instance i with m features
$D_{c,c}$	Continuous data set with subscripts c signifying both target and inputs are continuous
$D_{d,c}$	Continuous data set with subscript d signifying the target is discrete while inputs are continuous
Y_d	Vector for the discrete target with d distinct classes
y_i	Discrete output label for an instance i
$\mathbb{P}$	Set of target predicates
p	A given target predicate
C_p	Set of continuous predicates associated with each variable X_i
$\mathcal{F}_p$	Target predicate Boolean function for a predicate p
gt^i_x, lt^i_x	Boolean upper/lower boundary predicate for variable x and interval i
l_{xi}, u_{xi}	Lower/upper boundary values for feature x and interval i
$\mathcal{F}_{gt^i_x}$, $\mathcal{F}_{lt^i_x}$	Continuous bounded Boolean predicate functions
$gt^i_{f(x)}$, $lt^i_{f(x)}$	Non-linear transformation upper/lower boundary predicates for $f(x) \in \{sin(x), exp(x), \cdots\}$
$\mathcal{F}_{gt^i_{f(x)}}$, $\mathcal{F}_{lt^i_{f(x)}}$	Non-linear transformation bounded Boolean predicate functions
$\mathcal{F}_{gt^i_{f(x,y)}}$, $\mathcal{F}_{lt^i_{f(x,y)}}$	Operation bounded Boolean predicate functions between features x and y
KB_T, KB_O	Transformation and operation knowledge base
$\mathbf{I}$	Input matrix
$\mathcal{X}_p$	Evaluated predicate p rule
$\mathbb{X}$	Set of optimised target predicates
F, F^*	Non-linear generator function and non-linear function approximation

Table 2 Listed user defined hyperparameters and definitions

Hyperparameter	Definition
c	A constant applied to sigmoid function inputs to ensure output is Boolean
$\mathbf{b}$	Batch size
k	Number of bin intervals for continuous variables
d	Number of discrete classes on the target
N_p	Row space of the disjunction layer
N_e	Column space of the conjunction layer

random variable X_i, and for a given instance i in the data, $\{X_1, X_2, \cdots, X_m\}_i$, as this uppercase format reflects the final logical rule syntax given by the architecture (seen in the results section).

3.4 Continuous predicates

The original dNL architecture made use of continuous predicates but the investigation was limited. To handle continuous inputs, Boolean predicate functions are applied to the continuous variables as a series of lower and upper bound predicates (Payani & Fekri, 2019; Speichert

& Belle, 2018; Belle et al., 2016; Bueff et al., 2021). The resulting continuous Boolean predicates are taken in as input continuous values and return either true or false based on whether the input meets the condition. In this interpretation, each continuous variable x is defined by k upper and lower boundary predicates, where we have a Boolean upper boundary predicate $gt^i_x(x, l_{xi})$, which states whether "*x is greater than* l_{xi}" is true, and the lower boundary predicate $lt^i_x(x, u_{xi})$ which states whether "*x is less than* u_{xi}" is true, where $i \in 1, 2, \cdots, k$. The lower and upper boundary values l_{xi} and u_{xi} respectively are also treated as trainable weights allowing for the following definition for the predicates:

$$\mathcal{F}_{gt^i_x} = \sigma(c(x - u_{xi})), \quad \mathcal{F}_{lt^i_x} = \sigma(-c(x - l_{xi})) \tag{3}$$

In Eq. (3), the sigmoid (σ) and constant $c \geq 1$ (set to $c = 20$) are applied to ensure that the output is Boolean. While the sigmoid ensures a Boolean output, the constant acts on the steepness of the sigmoid curve. A larger constant value increases the steepness, resulting in greater confidence when the input is in fact greater/less than the corresponding boundary value.

3.5 dNL loss function

The following loss functions are used in the dNL network to ensure that training accounts for loss on the predictive output and heuristic loss related to the interpretability of the learned clauses. The primary loss measure is the average cross-entropy between predictions on $\mathcal{X}_p[e]$, where e is a ground truth from the set of negative and positive class examples $\mathcal{N}_p$, $\mathcal{P}_p$, and the true classification of e. The loss function takes into account all predicates p in the set of target predicates $\mathbb{P}$ (see Eq. 4):

$$\begin{aligned} \mathcal{L} = -\mathbb{E}_{p\in\mathbb{P}}\mathbb{E}_{(e,\lambda_p)\in\Lambda_p}\Big\{ \lambda_p log\mathcal{X}_p[e] + (1-\lambda_p) log(1-\mathcal{X}_p[e]) \Big\} \\ \text{where, } \Lambda_p = \Big\{ (\gamma, 1) | \gamma \in \mathcal{P}_p \Big\} \bigcup \Big\{ (\gamma, 0) | \gamma \in \mathcal{N}_p \Big\} \end{aligned} \tag{4}$$

Additional loss metrics include a measure of interpretability (see Eq. 5), where consideration is given to the fact that membership weights may not converge to 0 or 1 in the final network. Often such cases occur when no formal Boolean rule definition can model the discrete target classification in the background knowledge. To increase interpretability, the dNL model is optimised such that membership weights converge to 0 and 1. The membership weights m reflect the weights from both the disjunction and conjunction layers used to define $\mathcal{F}_p$.

$$\mathcal{L}_{int} = \mathbb{E}_{p\in\mathbb{P}}\mathbb{E}_{m\in\mathcal{F}_p} m(1-m) \tag{5}$$

The final loss term is designed to reduce the number of terms in each clause via pruning (see Eq. 6). During training, it is possible to encounter cases of redundancy, so for each predicate function $\mathcal{F}_p$, the dNL model prunes the definitions by subtracting from the sum of membership weights, the term N_{max}, which denotes the number of allowed terms in the definition.

$$\mathcal{L}_{prn} = \sum_{p\in\mathbb{P}} relu\Big(\sum_{m_i\in\mathcal{F}_p} m_i - N_{max} \Big) \tag{6}$$

In Eq. 7, all loss terms are combined into a final aggregated loss metric. Note the included hyperparameters on the pruning loss (λ_{prn}) and on the interpretability loss (λ_{int}).

$$\mathcal{L}_{agg} = \mathcal{L} + \lambda_{prn}\mathcal{L}_{prn} + \lambda_{int}\mathcal{L}_{int} \tag{7}$$

4 Contributions

The following section discusses the extensions applied to baseline dNL networks to derive dNL-NL networks, as well as coverage of the transformation and operation layers which will be used in the overall pipeline to extract non-linear functions from continuous data. We first provide a general overview of the various components prior to discussing them more in-depth with regard to their algorithmic and mathematical implementation

Non-linear and Operation predicates: Our dNL-NL model efficiently handles non-linear transformations of continuous inputs. We apply standard non-linear transformations like power, exponential, and sine functions. Additionally, the model uses operation predicates, symbolizing basic mathematical operations, enhancing its computational capabilities.

Handling of Non-linear Continuous data: To manage the non-linear continuous data, we redefined it within the classification framework. This involved basing our target predicate on continuous Boolean predicates. For each classification within a dataset with discrete targets and continuous variables, we efficiently separated positive and negative examples.

Input Matrix and dNL architecture: Our model's architecture prominently features an input matrix. This matrix converts continuous inputs into a Boolean interpretation suitable for dNL-NL processing. The activations from the input matrix are fed through a conjunctive and disjunctive layer each with respective membership weights.

Loss function: The model employs a loss function designed for rule consistency and reduced complexity. Additionally, the 'true loss' metric measures the difference between the extracted non-linear function and the original continuous target.

Subsequent sections delve into *transformation* and *operation* layers and the *rule extraction pipeline*, integrating the aforementioned techniques into a unified framework.

4.1 Non-linear and operation predicates

Consider the initial example for the equation $E = m \times c^2$. Our objective is to learn logical rules that define a particular region of its output. Using these rules, we can reconstruct a non-linear equation with the given predicates. In our framework, m is addressed by predicates from Eq. 3, while c necessitates the transformation c^2. Additionally, predicates capturing the multiplication operation $m \times c^2$ are paramount. Though inequalities define the range $0 \leq E \leq 3.07 \times 10^{-1}$ for the associated discrete class predicate, our primary focus is on variable transformations and operations. Subsequent continuous predicates are derived from this need.

To handle continuous inputs, we employ non-linear transformations, specifically focusing on functions like the power ($f(x) = x^2$), exponential ($f(x) = exp(x)$), and sine ($f(x) = sin(x)$) function. We introduce k upper and lower boundary transformation predicates for each (*non-linear transformation predicates*):

$$\text{square: } \mathcal{F}_{gt^i_{Sqr(x)}} = \sigma(c(x^2 - u_{xi})),\ \mathcal{F}_{lt^i_{Sqr(x)}} = \sigma(-c(x^2 - l_{xi}))$$
$$\text{exponential: } \mathcal{F}_{gt^i_{Exp(x)}} = \sigma(c(exp(x) - u_{xi})),\ \mathcal{F}_{lt^i_{Exp(x)}} = \sigma(-c(exp(x) - l_{xi}))$$
$$\text{sine: } \mathcal{F}_{gt^i_{Sin(x)}} = \sigma(c(sin(x) - u_{xi})),\ \mathcal{F}_{lt^i_{Sin(x)}} = \sigma(-c(sin(x) - l_{xi}))$$

Here, $i \in 1, ..., k$, σ denotes the sigmoid function, c is a constant, and u_{xi} and l_{xi} represent upper and lower bounds for the i^{th} predicate. The bounded weights are trainable parameters specific to each transformation predicate. Given that, the bounded weights are shared among the learned target predicates in $\mathbb{P}$ to ensure learned literals are consistent among the rule definitions. During training, we split transformed continuous features into k bins of equal width.

Furthermore, we define operation predicates, capturing arithmetic operations between variables. For variables x and y, we present k upper and lower boundary predicates (*operation predicates*):

$$\begin{aligned}
\text{addition: } \mathcal{F}_{gt^i_{Add(x,y)}} &= \sigma(c((x + y) - u_{(x,y)i})),\\
\mathcal{F}_{lt^i_{Add(x,y)}} &= \sigma(-c((x + y) - l_{(x,y)i}))\\
\text{subtraction: } \mathcal{F}_{gt^i_{Sub(x,y)}} &= \sigma(c((x - y) - u_{(x,y)i})),\\
\mathcal{F}_{lt^i_{Sub(x,y)}} &= \sigma(-c((x - y) - l_{(x,y)i}))\\
\text{multiplication: } \mathcal{F}_{gt^i_{Prod(x,y)}} &= \sigma(c((x \times y) - u_{(x,y)i})),\\
\mathcal{F}_{lt^i_{Prod(x,y)}} &= \sigma(-c((x \times y) - l_{(x,y)i}))
\end{aligned}$$

These boundaries stem from the resulting outputs of variable operations. Combined, our dNL model, equipped with both transformation and operation predicates, can adeptly infer non-linear relationships from data.

4.2 Non-linear continuous data

In the dNL framework and similar ILP-inspired architectures, a Knowledge Base (KB) is formed with positive instances $\mathcal{P}$ and negative instances $\mathcal{N}$, supported by general background assumptions $\mathcal{B}$. For dNL's continuous input handling, continuous attributes undergo a transformation into Boolean predicate functions via discretisation. We recast continuous data $D_{c,c}$ to fit a classification problem, resulting in discrete targets, where subscripts c and d indicate continuous and discrete data, respectively. Our target predicate's body consists of continuous Boolean predicates, with its head representing a discrete class, mirroring a non-linear function's output range, $(a \leq \mathsf{F}(x) \leq b)$. Given a dataset $D_{d,c}$, with a discrete target Y_d and continuous variables X_i (where i ranges from 1 to m), instances belonging to a specific classification are viewed as positive examples, and the rest are considered negative.

Using the dNL-NL model, we craft background knowledge, as illustrated in Fig. 1. Inputs from an example dataset are transformed according to our transformation KB KB_T, resulting in our background knowledge $\mathcal{B}$. This transformation process is further elaborated in Algorithm 1 where background knowledge is crafted for each predicate p in the set of target predicates $\mathbb{P}$. Depending on our model's emphasis (either operation or transformation), background knowledge incorporates either operations from KB_O or transformations from KB_T, but not both. In Algorithm 1, the 'OR' reflects this option and is not an

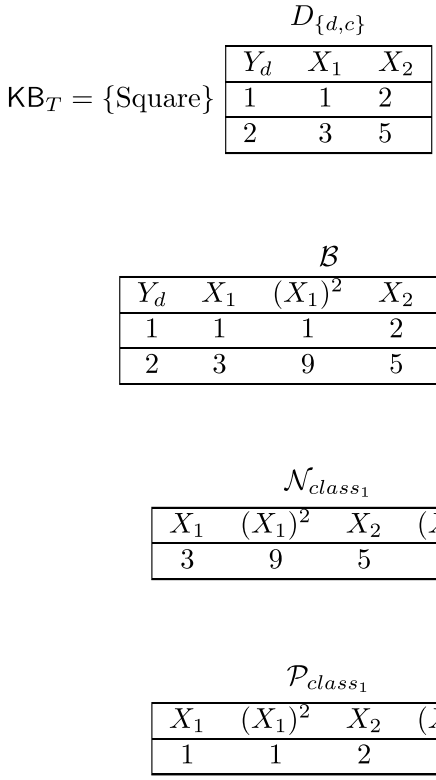

Fig. 1 Example depiction of creating background knowledge from original data and the representation of positive and negative examples for continuous data with respect to target predicate $class_1$

executable operation. Training leverages $\mathcal{B}$ to determine positive and negative instances corresponding to a class. As an example, for the predicate linked to $class_1$, we derive $\mathcal{N}_{class_1}$ and $\mathcal{P}_{class_1}$ from $\mathcal{B}$ - a process depicted in Fig. 1. During training iterations, each data batch provides positive and negative instances for every target class.

Algorithm 1 building Background Knowledge

Input: $\mathbb{P}$-set of target predicates, $D_{d,c}$-data with discretized target, KB_T-transformation KB, KB_O-operation KB
Output: $\mathcal{B}$- background knowledge
for $(p \in \mathbb{P})$ {
  for (each $(y_i, \{X_1, X_2, \cdots, X_m\}_i) \in D_{(d,c)}$) {
    if (y_i is an instance of p) **then** {
      $\mathcal{B} \cup \{p \leftarrow \bigwedge_{i=1}^{m} f(X_i) | f \in \mathsf{KB}_T\}$
      OR
      $\mathcal{B} \cup \{p \leftarrow \bigwedge_{i=1}^{m} g(X_i, \{\cdots, X_{i-1}, X_{i+1}, \cdots\}_{\setminus X_i}) | g \in \mathsf{KB}_O\}$
    }
  }
}

4.3 Input matrix and architecture structure

The Boolean target predicate function, $\mathcal{F}_p$, is constructed of alternating conjunction and disjunction layers, arranged in DNF form, as described by Eqs. () and (). These layers process Boolean predicate functions derived from continuous variables. Specifically, within $\mathcal{F}_p$, the conjunction layer's membership weight columns, N_e, are defined as $N_e = 2 \times k \times |C_p|$, where the factor of 2 corresponds to separate lower and upper bound predicates, and C_p indicates continuous predicates derived from each variable X_i and its affiliated knowledge base KB_i. The number of stacked conjunction neurons at the disjunction layer is defined by the hyperparameter N_p.

The input matrix, $\mathbf{I}$, defined in Eq. 8, consolidates continuous lower and upper bound predicate functions. The column space is defined by N_e and the row space is defined by the batch size. During training, batch size, $\mathbf{b}$, is left as a hyperparameter. This matrix facilitates the transformation of continuous data into a Boolean form suitable for the dNL-NL network's reasoning.

$$\mathbf{I} = \left[\bigvee_{e \in C_p} \bigvee_{i=1}^{k} \left(\mathcal{F}_{gt_e^i} \vee \mathcal{F}_{lt_e^i} \right) \right]_{\mathbf{b}} \tag{8}$$

$\mathcal{F}_p$ is defined by $\mathbf{I}$ and encompasses continuous Boolean predicates, membership weights, disjunction layer F_{disj}, and conjunction layer F_{conj}, as characterized in Eq. 9.

$$\mathcal{F}_p|_{\mathbf{I},N_e,N_p} = F_{disj}(N_p, F_{conj}(N_e, \mathbf{I})) \tag{9}$$

The neurons in the primary dNL function initialize the membership weights close to zero using random Gaussian distributions to prevent gradients from becoming exceedingly small. Equations (10a) and (10b) detail how conjunction and disjunction functions interact with the input matrix and membership weights.

$$\begin{aligned} F_{conj} &= \sum_{i=1}^{N_e} [1 - m_i^{conj}(1 - \mathbf{I}_i)] \\ \text{where, } \mathbf{m}^{conj} &= \sigma(c\mathbf{W}^{conj}),\ \mathbf{W}^{conj}_{N_p,N_e} \sim \mathcal{N} \end{aligned} \tag{10a}$$

$$\begin{aligned} F_{disj} &= 1 - \sum_{i=1}^{N_p} [1 - m_i^{disj} F_{conj}^i] \\ \text{where, } \mathbf{m}^{disj} &= \sigma(c\mathbf{W}^{disj}),\ \mathbf{W}^{disj}_{1,N_p} \sim \mathcal{N} \end{aligned} \tag{10b}$$

Note that the Boolean membership weights $\mathbf{m}$ are derived from a sigmoid transformation of weights $\mathbf{W}$, and $\mathbf{W}^{conj}$ is a matrix with dimensions $(N_p \times N_e)$ while $\mathbf{W}^{disj}$ is a vector of size N_p.

Algorithm 2 encapsulates the dNL-NL model's single-step design, merging Eqs. (8), (10a), and (10b). In this model, each grounding e updates in one step based on the background knowledge. Following a single step on an input batch, we ground the weighted matrix $\mathcal{X}_p$ for predicate p. This method integrates the logical entailments from our input matrix. Upon training all predicate functions, a set $\mathbb{X}$ of evaluated target predicate functions is returned.

Algorithm 2 Non-linear Single Step Forward Chain Model/ dNLNL model

Input: C_p-Set of continuous predicates, $\mathbb{P}$-set of target predicates
Output: $\mathbb{X}$-set of evaluated target predicates
for $(p \in \mathbb{P})$ {
  $\mathbf{I}$=[]
  for $(e \in C_p)$ {
    for ($i \in \{1, 2, \cdots, k\}$) {
      $\mathbf{I} \cup (\mathcal{F}_{gt_e^i} \bigvee \mathcal{F}_{lt_e^i})$
    }
  }
  $\mathcal{X}_p = \mathcal{F}_p|_{\mathbf{I}, N_e, N_p}$
  $\mathbb{X} \cup (\mathcal{X}_p)$
}

4.4 Loss function

The loss function design ensures that rules for discrete classes are consistent and that the complexity is minimized (refer to Eq. 7). The true loss, $Loss_{true}$, while not employed during dNL-NL network training, acts as a measure of how closely the non-linear function approximation, F^*, derived from the network, matches the original continuous target. $Loss_{true}$ represents the average absolute disparity between our approximated function and the continuous output Y_c, see Eq. 11.

$$Loss_{true} = \frac{1}{N} \sum_{i=1}^{N} \left| \left(Y_c[i] - \mathsf{F}^*(\{X_1, X_2, \cdots, X_m\}_i) \right| \right. \tag{11}$$

4.5 Transformation layer

The objective of the transformation layer is to learn the mathematical transformations applied to individual variables. The layer takes as input, discretized data associated with a variable X_i, denoted $D_{d.c}^{(i)}$, and the knowledge base on transformations, KB_T. Then outputs the transformation layer accuracy $acc_{X_i}^T$ and a set of optimised target predicates $\mathbb{X}_{X_i}^T$.

The dNL-NL architecture extracts rules that capture the non-linear relationship between continuous variables using discrete classification of the target. The rules, however, might sometimes deviate from the original non-linear function F which derived the data, especially when access to all features and transformations is granted. This deviation might result in either the inclusion of non-existent non-linear transformations or the omission of certain variables. To overcome these challenges, it's preferable to adopt a more constrained approach. In this context, the transformation layer applies the dNL-NL model to individual

variables to determine their mathematical transformations. This is done in a manner where the knowledge base can be updated or specific transformations removed if required.

Seen in Algorithm 3, for input variable X_i, a list of continuous predicates C_p is created using the available transformations from KB_T. Subsequently, the knowledge base is instantiated using the dataset, target predicates, and potential continuous predicates. The dNL-NL model $\mathbb{X}$ is then constructed and trained for a number of iterations (I_{max}) using batches ($\mathcal{B}^{(I)}$). The optimization focuses on the average cross-entropy between the ground truth and the individual class predictions but also uses the loss function $\mathcal{L}_{agg}$ (see Eq. 7). The learning rate starts at 0.001 and is adjusted for faster convergence.

Algorithm 3 Transformation Layer

Input: $D_{d,c}^{(i)}$ data with discretized target, KB_T-transformation KB, X_i-variable
Output: $acc_{X_i}^T$-transformation layer accuracy, $\mathbb{X}_{X_i}^T$-set of optimised target predicates
$\mathbb{P}$, $C_p \leftarrow$ CREATEPREDICATE(X_i, KB_T)
$\mathcal{B} \leftarrow$ BUILDBACKGROUND$(C_p, \mathbb{P}, D_{d,c}^{(i)})$
$\mathbb{X} \leftarrow$ dNLNLmodel$(C_p, \mathbb{P}, \mathcal{B})$
for $(I \in \{1, 2, \cdots, I_{max}\})$ {
    $\mathcal{B}^{(I)} = []$
    for $(p \in \mathbb{P})$ {
        $\mathcal{B}^{(I)} \cup [\mathcal{P}_p \leftarrow \{\gamma \models p | \gamma \in \mathcal{B}\} \bigcup \mathcal{N}_p \leftarrow \{\gamma \not\models p | \gamma \in \mathcal{B}\}]_{\mathbf{b}}$
    }
    $acc_{X_i}^T, \mathbb{X}_{X_i}^T = \text{AdamOptimizer}(\mathcal{L}_{agg}, \mathbb{X}, \mathcal{B}^{(I)})$
}

4.6 Operation layer

The objective of the operation is to learn rules reflecting mathematical operations between variables after their transformations have been learned. The layer takes as input a dataset with discretized target $D_{d,c}$, the operation knowledge base KB_O, and optimised transformation layer models for all variables, $\{\mathbb{X}_{X_1}^T, ..., \mathbb{X}_{X_m}^T\}$. Then the layer outputs the operation layer accuracy acc^O and a set of optimised target predicates $\mathbb{X}^O$.

Following the transformation layers, the operation layer comes into play. It differs from the transformation layer primarily in its knowledge base and dataset: it uses an operation-associated knowledge base and includes all variables in the dataset. After the transformations have been learned for each feature, they are used to inform the operation layer's learning process.

$$\begin{aligned} &\text{TRANSFORMDATA}(D_{d,c}, \{\mathbb{X}_{X_1}^T, \cdots, \mathbb{X}_{X_m}^T\}) = e_{X_i}(D_{d,c}^{(i)}) \\ &\quad \text{where, } e_{X_i} \leftarrow \text{MAXPREDICATE}(\mathbb{X}_{X_i}^T),\ \forall \mathbb{X}_{X_i}^T \in \{\mathbb{X}_{X_1}^T, \cdots, \mathbb{X}_{X_m}^T\} \end{aligned} \tag{12}$$

$$\textsc{maxPredicate}(\mathbb{X}^T_{X_i}) = *arg\,max_e \left[\sum_{\mathcal{X}_j \in \mathbb{X}^T_{X_i}} \sum_{i=1}^{2\times k} \mathbb{1}\left(m^e_{\mathcal{X}_j,i} > \epsilon \right) \middle| \forall e \in \mathcal{X}_j \right] \tag{13}$$

As seen in Algorithm 4, the dataset is transformed based on the predicates from the transformation layer with the highest confidence. The confidence of a predicate is gauged by its associated weights. The function transformDataset (see Eq. 12) takes the original dataset and the transformation layer's results to derive a transformed dataset $D_{d,T}$. For maxPredicate (see Eq. 12) we iterate over all associated transformation predicates, ignoring the upper/lower bound distinction, and count the number of transformation predicates with membership weights greater than hyperparameter ϵ to determine which transformation to apply on the variable. Like the transformation layer, the operation layer builds its knowledge base and subsequently trains the dNL-NL model. The final optimised architecture, $\mathbb{X}^O$, captures operations between transformed variables and is pivotal in deducing the true non-linear function.

Algorithm 4 Operation Layer

Input: $D_{d,c}$-data with discretized target, KB_O-operation KB, $\{\mathbb{X}^T_{X_1}, \cdots, \mathbb{X}^T_{X_m}\}$-set of optimised variable predicates
Output: acc^O-operation layer accuracy, $\mathbb{X}^O$-set of optimised target predicates
$\mathbb{P}, C_p \leftarrow$ createPredicateNames$(\{X_1, \cdots, X_m\}, \mathsf{KB}_O)$
$D_{d,T} \leftarrow$ transformDataset$(D_{d,c}, \{\mathbb{X}^T_{X_1}, \cdots, \mathbb{X}^T_{X_m}\})$
$\mathcal{B} \leftarrow$ buildBackground$(C_p, \mathbb{P}, D_{d,T})$
$\mathbb{X} \leftarrow$ dNLNLmodel$(C_p, \mathbb{P}, \mathcal{B})$
for $(I \in \{1, 2, \cdots, I_{max}\})$ {
    $\mathcal{B}^{(I)} = []$
    for $(p \in \mathbb{P})$ {
        $\mathcal{B}^{(I)} \cup [\mathcal{P}_p \leftarrow \{\gamma \models p | \gamma \in \mathcal{B}\} \bigcup \mathcal{N}_p \leftarrow \{\gamma \not\models p | \gamma \in \mathcal{B}\}]_{\mathbf{b}}$
    }
    $acc^O, \mathbb{X}^O = \text{AdamOptimizer}(\mathcal{L}_{agg}, \mathbb{X}, \mathcal{B}^{(I)})$
}

4.7 Rule extraction pipeline

In Algorithm 5, we integrate the *transformation layer*, *operation layer*, and construction of non-linear function approximation F^* to compute the true loss $Loss_{true}$ (see Eq. 11). In Fig. 2, a high-level structure of the transformation and operation layers is displayed along with the general pipeline connecting the layers. Our initial step discretizes the continuous dataset $D_{c,c}$ into d classes, yielding $D_{d,c}$. Using equal-width binning, we copy the transformation knowledge base KB^*_T, updating it during training if extraction fails. We introduce Boolean flags for each variable $flag_{\{X_1, X_2, \cdots, X_m\}}$ to determine the use of the modified

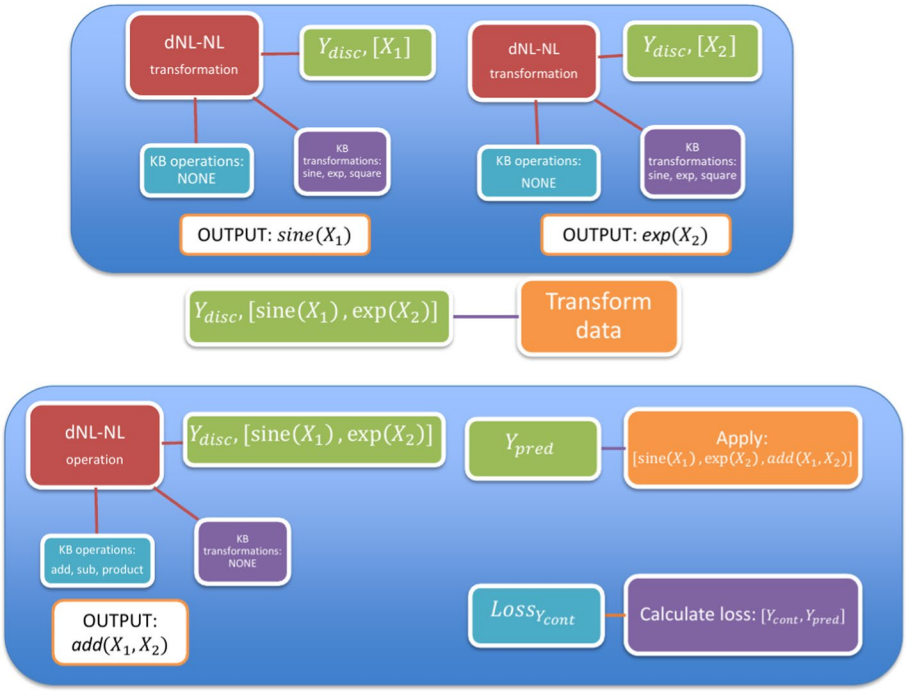

Fig. 2 High level view of transformation and operation layers

transformation knowledge base KB_T^*. A data structure $\mathbf{F}^*$ stores non-linear function approximations for updating the operation knowledge base KB_O^*.

The pipeline's main loop calculates $Loss_{true}$ after the operation and transformation layers. We continue until $Loss_{true}$ is below a threshold β (set at 0.05). Transformation layers, associated with each feature X_i, are trained first. Depending on the feature flag $flag_{X_i}$, we use either the reduced or full knowledge base. After training, we save the model $\mathbb{X}_{X_i}^T$ and its accuracy score $accX_i^T$. Using accuracy scores, the least performing layer is identified, and its associated predicate is removed from the updated knowledge base for the next iteration.

The true loss's value guides whether to update the knowledge base. This is done using the CALCLOSS function. Beforehand, we check the data structure $\mathbf{F}^*$ for any derived function approximation F^*. The method CHECKOPERATIONS compares functions to see if certain transformation pairings have been previously derived. Recognized pairings lead to the removal of corresponding operations from KB_O^* to avoid redundancy. A boolean flag $flag_{operation}$ indicates if the operation layer uses the full transformation set KB_O or its updated version KB_O^*.

The CALCLOSS function, using optimised models from each layer, constructs the non-linear function per Eq. (1213). This equation computes the loss on dataset $D_{c,c}$ with the continuous target, yielding $Loss_{true}$. After calculating the loss, the non-linear approximation is stored in $\mathbf{F}^*$.

Algorithm 5 Pipeline

Input: C_p-Set of continuous predicates, $\mathbb{P}$-set of target predicates, $D_{c,c}$ -data with continuous target and variables, KB_T-transformation KB, KB_O-operation KB
Output: F^*- non-linear approximation function
$D_{d,c} \leftarrow$ equalWidthBinning$(d, D_{c,c})$
$Loss_{true} = inf$
$\mathsf{KB}_T^* \leftarrow$ copy(KB_T)
$flag_{\{X_1, X_2, \cdots, X_m\}} = [False]_m$
$\mathbf{F}^* = []$
while ($Loss_{true} > \beta$) **do** {
  for ($X_i \in \{X_1, X_2, \cdots, X_m\}$) {
    if ($flag_{X_i}$ is true) **then** {
      if (KB_T^* is empty) **then** {
        $\mathsf{KB}_T^* \leftarrow$ copy(KB_T)
      }
      $acc_{X_i}^T, \mathbb{X}_{X_i}^T \leftarrow$ transformationLayer$(D_{d,c}^{(i)}, \mathsf{KB}_T^*, X_i)$
      $flag_{X_i} = False$
    else
      $acc_{X_i}^T, \mathbb{X}_{X_i}^T \leftarrow$ transformationLayer$(D_{d,c}^{(i)}, \mathsf{KB}_T, X_i)$
    }
  }
  $flag_{operation} = True$
  if ($\mathbf{F}^*$ not empty) **then** {
    $\mathsf{KB}_O^* \leftarrow$ checkOperations$(\mathbf{F}^*, \{\mathbb{X}_{X_1}^T, \cdots, \mathbb{X}_{X_m}^T\})$
    if (KB_O^* != KB_O) **then** {
      $flag_{operation} = False$
    }
    if (KB_O^* is empty) **then** {
      $\mathsf{KB}_O^* \leftarrow$ copy(KB_O)
      $flag_{operation} = True$
    }
  }
  if ($flag_{operation}$) **then** {
    $acc^O, \mathbb{X}^O \leftarrow$ operationLayer$(D_{d,c}, \mathsf{KB}_O, \{\mathbb{X}_{X_1}^T, \cdots, \mathbb{X}_{X_m}^T\})$
    $Loss_{true}, \mathsf{F}^* \leftarrow$ calcLoss$(\mathbb{X}^O, \{\mathbb{X}_{X_1}^T, \cdots, \mathbb{X}_{X_m}^T\}, D_{c,c})$
    $\mathbf{F}^* \cup (\mathsf{F}^*)$
  else
    $acc^O, \mathbb{X}^O \leftarrow$ operationLayer$(D_{d,c}, \mathsf{KB}_O^*, \{\mathbb{X}_{X_1}^T, \cdots, \mathbb{X}_{X_m}^T\})$
    $Loss_{true}, \mathsf{F}^* \leftarrow$ calcLoss$(\mathbb{X}^O, \{\mathbb{X}_{X_1}^T, \cdots, \mathbb{X}_{X_m}^T\}, D_{c,c})$
    $\mathbf{F}^* \cup (\mathsf{F}^*)$
  }
  if ($Loss_{true} > \beta$) **then** {
    $X_j \leftarrow$ varmin$(acc_{X_1}^T, \cdots, acc_{X_m}^T)$
    $flag_{X_j} = True$
    $\mathsf{KB}_T^* \leftarrow$ updateTransformationKB$(\mathbb{X}_{X_j}^T)$
  }
}

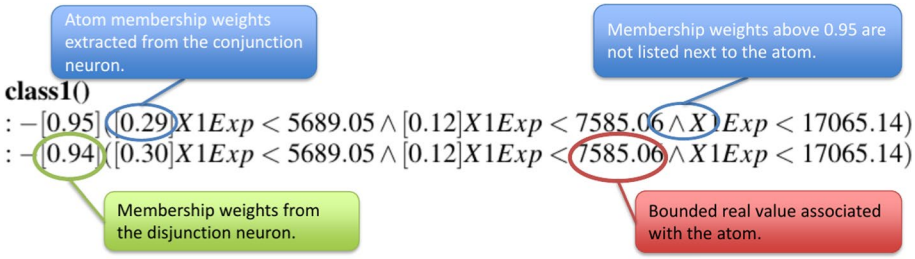

Fig. 3 Explanation of target predicate outputs

5 Experiments

The capacity of the dNL-NL pipeline to extract non-linear functions F^* is evaluated on synthetically generated datasets, where the datasets were generated by non-linear functions F. As dNL-NL represents a unique application of ILP based frameworks, we do not include comparative models, but instead focus on the capacity of dNL-NL to extract the correct non-linear functions from the generated data.[1] Ultimately, the goal is to extract non-linear rules using dNL-NL reasoning in order to construct a function that simulates the original ($\mathsf{F}^* \sim \mathsf{F}$). With the following experiments, we demonstrate that the dNL-NL framework can be used to extract non-linear functions from data provided a relevant KB.

Prior to discussing the dNL-NL extracted rules, we briefly explain the syntax in the rule hypotheses. As seen in Fig. 3, each dNL-NL network produces fitted predicate functions associated with each class, *class*1() in the figure. The clausal body is defined by a conjunction of atoms where the associated membership weight is placed before the atom in brackets, extracted from the conjunction neuron. If the membership weight is above 0.95, i.e. the model is confident it should be in the definition, then the weight is not listed next to the atom. As we stack two disjunction neurons in our dNL-NL architecture, we can have at most two hypotheses for a given class. Again, the value in brackets before the rule represents the membership weights from the disjunction neuron. If no rule is listed then no prior membership weights in the associated conjunction vector were above a reasonable confidence threshold.

5.1 Performance on noisy data

While the primary focus of this proposal is to employ our dNL-NL architecture for non-linear function extraction, we also tested it on noisy data to illustrate its capacity for discerning algorithmic patterns. This assists in determining which non-linear transformations or operations between features are pertinent for classification.

We evaluated our dNL-NL architecture using the Yacht Hydrodynamics dataset, available on the UCI machine learning repository (Dheeru & Karra Taniskidou, 2017). In this case, we explored non-linear transformations across all features and then defined

[1] In the future we intend to identify applications where it might be possible to compare learning approaches to non-linear modelling, and in which case a comparison could be drawn between other approaches and our proposal here. Moreover, we plan on applying this framework to learning explainable policies in reinforcement learning where we believe a reasonable model comparison can be performed.

Table 3 Each dNL learned rule for the discrete classes on the Yacht Hydrodynamics dataset

Yacht hydrodynamics	Rules
dNL: accuracy: 0.90	**class1()**
	: $-[1.00](Froude < 0.44 \wedge Froude < 0.27)$
	class2()
	: $-[1.00]([0.26]Froude > -0.49 \wedge Froude > 0.15 \wedge Froude < 0.44)$
	: $-[1.00]([0.14]Froude > -0.49 \wedge Froude > 0.15 \wedge Froude < 0.44)$
	class3()
	: $-[1.00]([0.25]Beamdraught > 3.28 \wedge [0.22]Beamdraught > 3.28 \wedge$ $[0.19]Beamdraught > 3.28 \wedge [0.12]Beamdraught > 3.27 \wedge$ $[0.11]Beamdraught < 5.86 \wedge [0.17]Froude > -0.49 \wedge$ $Froude > 0.15 \wedge Froude > 0.30)$
	: $-[1.00]([0.11]Beamdraught < 5.88 \wedge [0.23]Froude > -0.49 \wedge$ $Froude > 0.15 \wedge Froude > 0.30)$

Target is discretized into three classes and variable space includes 6 continuous features

operation predicates based on the two features present within the class rules. For comparison, the baseline model was built on dNL, relying solely on continuous Boolean predicate functions.

The yacht hydrodynamics dataset encompasses a range of adimensional parameters, chiefly concerned with hull geometry coefficients and the influential Froude number. These parameters provide insights into the yacht's hydrodynamic performance and its interactions with water. The parameters included in the dataset are as follows, Longitudinal position of the center of buoyancy, Prismatic coefficient, Length-displacement ratio, Beam-draught ratio, Length-beam ratio, and the Froude number. The target variable measured in this dataset is the residuary resistance per unit weight of displacement. This is a vital metric as it quantifies the additional resistance a yacht encounters beyond the frictional resistance due to its hull shape and other hydrodynamic factors. As the target feature is continuous, we use equal-frequency binning to parse the data into three discrete classes corresponding to the following features ranges, $class1() \sim 0.001 \leq \mathsf{F}(x) < 1.286$, $class2() \sim 1.286 \leq \mathsf{F}(x) < 7.806$, and $class3() \sim 7.806 \leq \mathsf{F}(x) \leq 62.42$.

In Table 3 we observe the performance of the baseline dNL model when using just continuous Boolean predicates. We note the accuracy is lower than our dNL-NL model using non-linear transformation predicates, see Table 4. In both cases, we observe that the Froude number feature and Beam-draught ratio are prevalent in the causal definitions. This indicates that these two features are more significant when determining the classification.

In Table 4, we further discern the specific non-linear transformations applied to each feature. Notably, the Beam-draught ratio feature predominantly undergoes the square function transformation. Meanwhile, the Froude number is subjected to a variety of transformations: sine, exponential, and square, with the exponential transformation being the most pronounced.

In Table 5, we focus on the two significant features: Beam-draught ratio and Froude number, aiming to discern the relevant operations between them. The Froude number is subjected to sine, exponential, and square transformations, while the Beam-draught ratio is primarily squared. We identify several operation predicates denoting specific operations. Notably, for $class1()$, $class2()$, and $class3()$, the prevailing operation is

Table 4 Each dNL-NL learned rule for the discrete classes on the Yacht Hydrodynamics dataset, using non-linear transformation predicates

Yacht hydrodynamics	Rules
dNL-NL: accuracy:0.98	**class1()**
	: $-[1.00](FroudeExp < 1.27 \wedge [0.66]FroudeSine < 0.30)$
	: $-[1.00](FroudeExp < 1.27 \wedge [0.68]FroudeSine < 0.30)$
	class2()
	: $-[1.00](FroudeExp > 1.28 \wedge FroudeSquare < 0.14)$
	class3()
	: $-[1.00]([0.70]BeamdraughtSquare > 13.53 \wedge FroudeExp > 1.44 \wedge FroudeExp > 1.28)$
	: $-[1.00](FroudeExp > 1.44 \wedge FroudeExp > 1.28 \wedge [0.73]FroudeSquare > 0.04)$

Target is discretized into three classes and variable space includes 6 continuous features

($Beamdraught^2 \times Froude^2$). It is also worth noting that this approach outperforms the baseline dNL model.

5.2 Learning two variable functions

We use the dNL-NL framework to extract the following non-linear equations which have all been synthetically generated on two continuous variables using various transformations and operations. We tested the proposed dNL-NL networks on 6 separate non-linear equations. Each equation contains a transformation on a continuous variable and a mathematical operation between the two transformed variables.

In Table 6, the value ranges on each variable used are continuous float values between [0, 10]. Each synthetically generated equation contains 200 instances. For the majority of equations, the continuous target was discretized into three classes ($d = 3$) using equal-width binning, and we set the number of boundaries to ($k = 7$) for the continuous variables. The extracted predicates for Table 6 are the result of 5-fold cross-validation for each dNL-NL layer in the framework. We also take the average run time for 10 training sessions to demonstrate the speed in deriving the non-linear rules. From the results, we see that we were able to extract the correct predicates representing the non-linear relationship of the synthetic data, but in some cases, the computation required a significant amount of time. This is notable with equation $exp(X1) - (X2)^2$, where the time to completion was far larger than the other equations. Similarly, $sin(X1) - exp(X2)$ also had a long computation time. In both cases the equations shared the subtraction operation, suggesting an area of investigation. We also note that the heuristics for updating the KB is another potential research avenue.

In reading the class rules, we will explain the meaning of the definitions in the context of the example non-linear function ($\mathbf{exp(X1)} \times \mathbf{(X2)^2}$) and one of its classes, see Table 7. Each target predicate ($class1()$, $class2()$, $class3()$) represents an output range based on the non-linear function which generated the data `F` (ranges determined by equal-width binning), so the clausal body states what inequalities need to be true on the transformed features in order for the non-linear function's output to fall within a given range. For the transformation layers, this is only done on a single feature, so we observe rule definitions

Table 5 Each dNL-NL learned rule for the discrete classes on the Yacht Hydrodynamics dataset, using operation predicate functions

Yacht hydrodynamics	Rules
dNL-NL: accuracy:0.95	**class1()**
	: $-((Beamdraught^2 \times Froude^2) < 1.68 \wedge (Beamdraught^2 \times Froude^2) < 2.12 \wedge$
	$[0.76](Beamdraught^2 \times sin(Froude) < 4.03)$
	: $-((Beamdraught^2 \times Froude^2) < 0.75 \wedge [0.94](Beamdraught^2 \times Froude^2) < 1.68$
	$\wedge [0.91](Beamdraught^2 \times Froude^2) < 2.12)$
	class2()
	: $-((Beamdraught^2 + exp(Froude) > 17.09 \wedge$
	$[0.90](Beamdraught^2 - exp(Froude) > 11.65 \wedge$
	$[0.90](exp(Froude) - Beamdraught^2) < -11.65 \wedge$
	$(Beamdraught^2 \times Froude^2) > 0.54 \wedge (Beamdraught^2 \times Froude^2) > 1.12$
	$\wedge [0.82](Beamdraught^2 \times Froude^2) < 2.12$
	$\wedge (Beamdraught^2 \times Froude^2) < 3.17)$
	: $-([0.76](exp(Froude) - Beamdraught^2) > -14.67 \wedge$
	$(Beamdraught^2 \times Froude^2) > 0.54 \wedge (Beamdraught^2 \times Froude^2) < 1.68$
	$\wedge (Beamdraught^2 \times Froude^2) < 2.12 \wedge$
	$[0.79](Beamdraught^2 \times Froude^2) < 3.17 \wedge$
	$[0.90](Beamdraught^2 \times sin(Froude) > 1.33)$
	class3()
	: $-((Beamdraught^2 - exp(Froude) < 14.14 \wedge$
	$(exp(Froude) - Beamdraught^2) > -14.67 \wedge$
	$(Beamdraught^2 \times Froude^2) > 0.54 \wedge (Beamdraught^2 \times Froude^2) > 1.12$
	$\wedge [0.78](Beamdraught^2 \times sin(Froude) > 1.33 \wedge$
	$[0.89](Beamdraught^2 - sin(Froude)) < 16.08 \wedge$
	$[0.86](sin(Froude) - Beamdraught^2) > -16.10)$
	: $-([0.88](Beamdraught^2 \times Froude^2) > 0.54 \wedge (Beamdraught^2 \times Froude^2) > 1.73$
	$\wedge (Beamdraught^2 \times Froude^2) > 1.12 \wedge$
	$[0.73](Beamdraught^2 \times sin(Froude) > 1.33)$

Target is discretized into three classes and variable space includes 2 continuous features

Table 6 Average computation time (in seconds) and results for extracting non-linear equations composed of two variables with float values between [0, 10]

True equation	Extracted predicates	Avg. time
$\mathbf{exp(X1) \times (X2)^2}$	`X1Exp` ∧`X2Square` ∧ `X1X2Prod`	169.54
$\mathbf{sin(X1) + (X2)^2}$	`X1Sine` ∧`X2Square` ∧ `X1X2Add`	181.89
$\mathbf{exp(X1) \times sin(X2)}$	`X1Exp` ∧`X2Sine` ∧ `X1X2Prod`	221.09
$\mathbf{exp(X1) - (X2)^2}$	`X1Exp` ∧`X2Square` ∧ `X1X2Sub`	1287.35
$\mathbf{(X1)^2 + exp(X2)}$	`X1Square` ∧`X2Exp` ∧ `X1X2Add`	61.79
$\mathbf{sin(X1) - exp(X2)}$	`X1Sine` ∧`X2Exp` ∧ `X1X2Sub`	679.35

containing just one variable. In the case of the transformation layer on $X1$, based on the output we could state that *"class 3 is true if the exponentiation of variable* $X1$ *is greater*

Table 7 Each dNL-NL layer's associated learned rules, where rules are defined by extracted non-linear predicates for the function $(exp(X_1) \times (X_2)^2)$ where the target was discretized into three classes and the variable space was floats in the range [0, 10]

$\mathbf{exp(X1) \times (X2)^2}$	Rules
Transformation: X1 accuracy: 0.775	**class1()**
	: $-[0.95]([0.29]X1Exp < 5689.05 \wedge [0.12]X1Exp < 7585.06 \wedge X1Exp < 17065.14)$
	: $-[0.94]([0.30]X1Exp < 5689.05 \wedge [0.12]X1Exp < 7585.06 \wedge X1Exp < 17065.14)$
	class2()
	: $-[0.39](X1Exp > 5689.05 \wedge [0.50]X1Exp > 7585.06 \wedge X1Exp < 11377.09 \wedge$ $[0.23]X1Square < 80.92)$
	class3()
	: $-[0.97]([0.80]X1Exp > 17065.14 \wedge X1Square > 91.18)$
Transformation: X2 accuracy: 0.875	**class1()**
	: $-[0.94]([0.20]X2Sine > -0.49 \wedge [0.12]X2Sine < 0.76 \wedge [0.38]X2Sine < 0.79)$
	: $-[0.96]([0.20]X2Square < 28.00 \wedge [0.38]X2Square < 54.06 \wedge$ $[0.73]X2Square < 64.23)$
	class2()
	: $-[0.68]([0.93]X2Square > 46.92 \wedge [0.81]X2Square > 71.66 \wedge$ $[0.94]X2Square < 72.94 \wedge [0.16]X2Square < 90.48)$
	class3()
	: $-[0.21](X2Square > 55.74 \wedge [0.92]X2Square < 72.94)$
Operation: X1 $\wedge$ X2 accuracy: 0.975	**class1()**
	: $-[0.96]([0.89]X1X2Prod < 377422.12 \wedge X1X2Prod < 566133.19)$
	: $-[0.96]([0.24]X1X2Add < 17146.57 \wedge [0.75]X1X2Prod < 377422.12 \wedge$ $X1X2Prod < 566133.19 \wedge [0.26]X1X2Sub < 17047.03 \wedge$ $[0.45]X2X1Sub > -17047.03)$
	class2()
	: $-[0.99](X1X2Prod > 377422.12 \wedge [0.53]X1X2Prod < 943555.38 \wedge$ $X1X2Prod < 1132266.38)$
	class3()
	: $-[0.99]([0.23]X1X2Prod > 754844.25 \wedge X1X2Prod > 943555.38 \wedge$ $[0.94]X1X2Prod > 1132266.38)$

*than 17065.14 and the square of variable X*1 *is greater than 91.8"* and this rule in conjunction with the rules for *class*1() and *class*2() are accurate 77.5% of the time. In the case of the transformation on *X*2, we could state *"class 3 is true if the square of variable X*2 *is greater than 55.75 and the square of variable X*2 *is less than 72.94"*. In the operation layer, which now defines the discrete classes based on operations between the two features, key to note is that the features have been transformed based on the best performing transformation in the previous transformation layers. The rules learned in the operation layer include operation atoms which describe the operation between transformed variables. In Table 7, this indicates the variable *X*1 is transformed by the exponent function and *X*2 is transformed by the square function, which results in rules that can be interpreted as follows *"class 3 is true if the product of exp*(*X*1) *and* $(X2)^2$ *is greater than 754844.25 and the product of exp*(*X*1) *and* $(X2)^2$ *is greater than 1132266.38"*. In various rule definitions we observe redundant literals (e.g.

$X1X2Prod < 94355.38 \wedge X1X2Prod < 1132266.38$ for $class2()$ of the operation layer). This is due to the stochastic nature of training of the membership weights in that some literals will supersede others based on their inequalities however redundant literals may still be present in the rule definition as the model was not overall penalised for including them. While they do not impact model performance, future research may consider a post-processing step to remove redundant literals for better interpretability. Note, the output in Table 7 has been propositionalised for clarity, however for inference to be performed on unseen instances, the predicate logic format of the rules is used to input values from $X1$ and $X2$.

The original output from various dNL-NL layers can be seen in Table 7, which depicts the resulting belief state for the equation $(\mathbf{exp(X1)}) \times (\mathbf{X2})^{\mathbf{2}})$ after 100 iterations for each associated dNL-NL layer. The three discrete classes correspond to the following features ranges, $class1() \sim 1.69 \times 10^{-2} \leq \mathsf{F}(x) < 4.43 \times 10^5$, $class2() \sim 4.43 \times 10^5 \leq \mathsf{F}(x) < 8.86 \times 10^5$, and $class3() \sim 8.86 \times 10^5 \leq \mathsf{F}(x) \leq 1.33 \times 10^6$. In the table it can be observed that the resulting rules were stronger at the operation layer than at the individual transformation layers. In the case of the transformation layer, it can be observed that the transformation on the second variable ($(X2)^2$) was better able to define the discrete classification predicates, as indicated by the higher accuracy. Across the layers, the learned predicate rules for each class were still able to identify the correct transformations based on the confidence associated with each grounding. We do note atoms in some of the definitions are not representative of the original non-linear function, for example in the transformation layer for $X2$, the first definition for *class1()* is defined by bounded atoms of the sine function. The pipeline still can learn atoms which model the target predicate yet do not represent the true non-linear function, but taken as a whole across all definitions and layers, the pipeline is able to extract the relevant transformations and operations.

In Table 8 the dNL-NL layers extract the non-linear equation $(\mathbf{sin(X1)} + \mathbf{(X2)^2})$, again with features sampled from a uniform distribution between [0, 10]. The three discrete classes correspond to the following features ranges, $class1() \sim -0.92 \leq \mathsf{F}(x) < 32.0$, $class2() \sim 32.0 \leq \mathsf{F}(x) < 64.9$, and $class3() \sim 64.9 \leq \mathsf{F}(x) \leq 97.8$. The synthetic datasets are again comprised of 200 instances with three classes and 7 equal-width boundaries on the continuous predicates. From Table 8, we again are able to extract the correct non-linear equations. Note the accuracy scores being 100 percent correct for the $X2$ transformation layer. This indicates the definitions for the three target classes in this layer were able to entail all positive and negative examples successfully, even if not all atoms were relevant to the true non-linear function (e.g. $X2Exp > 3911.76$). Also note the transformation layer accuracy for $X1$ is rather poor, such that the definitions for the various classes do not model the discrete target predicates well. This can be attributed to the sine functions periodic output and so greater emphasis is given to the non-linear output of the second transformation $(X2)^2$ in modelling the discrete classes independently. As a whole, the pipeline is able to extract the non-linear function which simulates the data.

In Table 9 the dNL-NL layers extract the non-linear equation $(\mathbf{exp(X1)} \times \mathbf{sin(X2)})$. The three discrete classes correspond to the following features ranges, $class1() \sim -1.26 \times 10^4 \leq \mathsf{F}(x) < -2.06 \times 10^3$, $class2() \sim -2.06 \times 10^3 \leq \mathsf{F}(x) < 8.43 \times 10^3$, and $class3() \sim 8.43 \times 10^3 \leq \mathsf{F}(x) \leq 1.89 \times 10^4$. Again note the accuracy of the first transformation later ($exp(X1)$) which outperforms the second transformation layer, and also note the accuracy of the second layer containing various irrelevant transformation continuous atoms. Given that, the confidence of the model for the sine function is far stronger across the classification predicates. In general, the periodic transformation of the sine function

Table 8 Each dNL-NL layer's associated learned rules, where rules are defined by extracted non-linear predicates for the function $(sin(X_1) + (X_2)^2)$ where the target was discretized into three classes and the variable space was floats in the range [0, 10]

$\mathbf{sin(X1) + (X2)^2}$	Rules
Transformation: X1 accuracy: 0.475	**class1()** : $-[0.82]([0.45]X1Sine > 0.47 \wedge [0.13]X1Sine < -0.60 \wedge [0.14]X1Sine < -0.94 \wedge$ $[0.45]X1Sine < 0.79)$: $-[0.80]([0.40]X1Sine < -0.60 \wedge [0.51]X1Sine < -0.94 \wedge [0.39]X1Sine < 0.79)$ **class2()** : $-[0.80]([0.79]X1Sine > -0.86 \wedge [0.51]X1Sine > -0.03, [0.28]X1Sine > -0.04 \wedge$ $[0.25]X1Sine < 0.17 \wedge [0.34]X1Sine < 0.17 \wedge [0.75]X1Sine < 0.15)$: $-[0.42](X1Sine > -0.86 \wedge [0.17]X1Sine > -0.03 \wedge X1Sine > 0.75 \wedge$ $[0.14]X1Sine < 0.17)$ **class3()** : $-[0.72]([0.92]X1Sine > 0.47 \wedge [0.85]X1Sine < 0.51)$: $-[0.86]([0.24]X1Sine > -0.61 \wedge [0.83]X1Sine > 0.47 \wedge [0.90]X1Sine < 0.51)$
Transformation: X2 accuracy: 1.00	**class1()** : $-[0.95]([0.60]X2Square < 28.06 \wedge X2Square < 35.54 \wedge [0.86]X2Square < 45.20)$: $-[0.95]([0.63]X2Square < 28.06 \wedge X2Square < 35.54 \wedge [0.85]X2Square < 45.20)$ **class2()** : $-[0.99](X2Exp < 3911.76 \wedge X2Square > 28.03 \wedge [0.27]X2Square < 64.17 \wedge$ $[0.83]X2Sine > -0.55)$ **class3()** : $-[0.96]([0.89]X2Exp > 3911.76 \wedge [0.19]X2Square > 54.31 \wedge X2Square > 64.03)$: $-[0.96]([0.31]X2Exp > 1956.46 \wedge [0.53]X2Exp > 3911.76 \wedge X2Square > 64.03 \wedge$ $[0.55]X2Sine < 0.95 \wedge [0.22]X2Sine < 0.97)$
Operation: X1 $\wedge$ X2 accuracy: 1.00	**class1()** : $-[0.96](X1X2Add < 35.04 \wedge X1X2Prod < 26.78 \wedge [0.26]X1X2Sub > -27.77 \wedge$ $[0.17]X2X1Sub < 27.77 \wedge [0.25]X2X1Sub < 35.93)$: $-[0.96](X1X2Add < 35.04 \wedge X1X2Prod < 26.78 \wedge [0.10]X1X2Sub > -35.93 \wedge$ $[0.16]X1X2Sub > -27.77 \wedge [0.33]X2X1Sub < 35.93)$ **class2()** : $-[0.99](X1X2Add > 35.06 \wedge [0.42]X1X2Add < 65.19 \wedge$ $X1X2Add < 71.85 \wedge [0.10]X1X2Sub > -64.36 \wedge [0.43]X2X1Sub < 64.40)$: $-[0.59](X1X2Add > 28.87 \wedge [0.23]X1X2Add > 35.06 \wedge X1X2Add < 71.85 \wedge$ $[0.52]X1X2Prod < 9.44)$ **class3()** : $-[0.99]([0.93]X1X2Add > 65.00 \wedge [0.94]X1X2Add > 71.84 \wedge$ $[0.89]X1X2Sub < -64.26)$: $-[0.98](X1X2Add > 65.00 \wedge [0.90]X1X2Prod > -9.84 \wedge [0.92]X2X1Sub > 64.31)$

leads to specific challenges when trying to model the discrete ranges associated with the target predicates.

In Table 10 the dNL-NL layers extract the non-linear equation $(\mathbf{exp(X1) - (X2)^2})$. The five discrete classes correspond to the following features ranges, $class1() \sim -84.6 \leq \mathsf{F}(x) < 4.09 \times 10^3$, $class2() \sim 4.09 \times 10^3 \leq \mathsf{F}(x) < 8.26 \times 10^3$,

Table 9 Each dNL-NL layer's associated learned rules, where rules are defined by extracted non-linear predicates for the function ($exp(X_1) \times sine(X_2)$) where the target was discretized into three classes and the variable space was floats in the range [0, 10]

$\mathbf{exp(X_1) \times sine(X_2)}$	Rules
Transformation: X1 accuracy: 0.925	**class1()**
	: $-[0.65]([0.82]X1Exp > 1897.02 \wedge [0.68]X1Exp > 5689.05 \wedge$
	$[0.53]X1Exp > 13273.11 \wedge [0.81]X1Exp < 15169.12)$
	: $-[0.30]([0.12]X1Exp > 1897.02 \wedge X1Exp > 5689.05 \wedge$
	$[0.84]X1Exp > 13273.11 \wedge [0.85]X1Exp < 15169.12)$
	class2()
	: $-[0.92]([0.11]X1Exp < 1897.02 \wedge [0.12]X1Exp < 3793.03 \wedge$
	$[0.45]X1Exp < 5689.05 \wedge X1Exp < 13273.11 \wedge$
	$[0.87]X1Exp < 15169.12)$
	: $-[0.93]([0.13]X1Exp < 1897.02 \wedge [0.16]X1Exp < 3793.03 \wedge$
	$[0.38]X1Exp < 5689.05 \wedge X1Exp < 13273.11 \wedge$
	$[0.88]X1Exp < 15169.12)$
	class3()
	: $-[0.96](X1Exp > 9481.08 \wedge X1Exp > 15169.12 \wedge [0.15]X1Exp < 17065.14 \wedge$
	$[0.67]X1Sine < 0.16)$
	: $-[0.94](X1Exp > 9481.08 \wedge [0.94]X1Exp < 11377.09 \wedge [0.89]X1Sine < 0.16)$
Transformation: X2 accuracy: 0.825	**class1()**
	: $-[0.48]([0.89]X2Exp < 17598.91 \wedge [0.60]X2Square > 28.23 \wedge$
	$[0.30]X2Square < 18.01 \wedge X2Sine < -0.29)$
	class2()
	: $-[0.92]([0.11]X2Sine > -0.28 \wedge [0.18]X2Sine < 0.84)$
	: $-[0.93]([0.20]X2Sine > -0.28 \wedge [0.13]X2Sine > -0.28 \wedge$
	$[0.11]X2Sine < 0.84 \wedge [0.17]X2Sine < 0.84)$
	class3()
	: $-[0.49]([0.58]X2Exp < 1956.46 \wedge [0.29]X2Square > 54.30 \wedge$
	$[0.19]X2Square < 7.95 \wedge X2Sine > 0.84)$
	: $-[0.24]([0.21]X2Exp < 1956.46 \wedge [0.71]X2Square > 54.30 \wedge$
	$[0.14]X2Square < 7.95 \wedge [0.95]X2Sine > 0.84)$
Operation: X1 $\wedge$ X2 accuracy: 1.00	**class1()**
	: $-[0.99]([0.48]X1X2Add > 1896.20 \wedge [0.13]X1X2Add > 5688.59 \wedge$
	$X1X2Prod < -1892.88 \wedge [0.13]X1X2Sub > 5688.59 \wedge$
	$[0.84]X2X1Sub < -1896.20)$
	class2()
	: $-[0.96]([0.27]X1X2Prod > -5684.47 \wedge [0.93]X1X2Prod > -1892.88 \wedge$
	$[0.66]X1X2Prod < 5690.31 \wedge X1X2Prod < 9481.91)$
	: $-[0.97]([0.32]X1X2Add < 9480.99 \wedge [0.15]X1X2Prod > -5684.47 \wedge$
	$[0.92]X1X2Prod > -1892.88 \wedge X1X2Prod < 9481.91 \wedge$
	$[0.25]X1X2Sub < 9480.99 \wedge [0.25]X2X1Sub > -9480.99)$
	class3()
	: $-[0.73]([0.59]X1X2Prod > 5690.31 \wedge X1X2Prod > 9481.91)$
	: $-[0.99](X1X2Add > 9480.99 \wedge X1X2Prod > 5690.31 \wedge$
	$[0.45]X1X2Prod > 9481.91)$

Table 10 Each dNL-NL layer's associated learned rules, where rules are defined by extracted non-linear predicates for the function $(exp(X_1) - (X_2)^2)$ where the target was discretized into three classes and the variable space was floats in the range [0, 10]

$\mathbf{exp(X_1) - (X_2)^2}$	Rules
Transformation layer: X1 accuracy: 1.00	**class1()**
	: $-[0.94]([0.70]X1Exp < 3793.03 \wedge [0.73]X1Exp < 5689.05 \wedge X1Square < 71.17)$
	: $-[0.95]([0.63]X1Exp < 3793.03 \wedge [0.83]X1Exp < 5689.05 \wedge X1Square < 71.17)$
	class2()
	: $-[0.99](X1Exp > 3793.03 \wedge [0.50]X1Square > 71.06 \wedge$ $X1Square < 80.95 \wedge [0.23]X1Sine > 0.16)$
	class3()
	: $-[0.99](X1Exp < 11377.09 \wedge X1Square > 80.93 \wedge [0.42]X1Square < 89.56)$
	class4()
	: $-[0.48](X1Exp > 11377.09 \wedge X1Exp < 17065.14)$
	: $-[0.98](X1Exp > 11377.09 \wedge X1Exp < 17065.14)$
	class5()
	: $-[0.94]([0.95]X1Exp > 15169.12 \wedge X1Exp > 17065.14)$
	: $-[0.74](X1Exp > 17065.14)$
Transformation layer: X2 accuracy: 0.75	**class1()**
	: $-[0.89]([0.19]X2Square > 17.83 \wedge [0.17]X2Square < 28.12 \wedge$ $[0.38]X2Square < 48.06)$
	: $-[0.89]([0.19]X2Square > 17.83 \wedge [0.44]X2Square > 70.13)$
	class2()
	: $-[0.18]([0.28]X2Square > 8.63 \wedge [0.53]X2Square > 47.96 \wedge$ $[0.18]X2Square < 18.05 \wedge [0.83]X2Square < 54.14 \wedge X2Square < 89.56)$
	: $-[0.21]([0.29]X2Square > 8.63 \wedge [0.63]X2Square > 47.96 \wedge$ $[0.12]X2Square < 18.05 \wedge [0.74]X2Square < 54.14 \wedge X2Square < 89.56)$
	class3()
	: $-[0.12]([0.93]X2Square > 55.89 \wedge [0.58]X2Square < 80.96)$

Table 10 (continued)

$\exp(\mathbf{X_1}) - (\mathbf{X_2})^2$	Rules
Operation layer: accuracy: 1.00	**class1()**
	: −[0.94]([0.93]$X1X2Sub < 3711.60 \wedge X1X2Sub < 5616.66\wedge$
	[0.25]$X1X2Sub < 7521.72$)
	: −[0.94]([0.93]$X1X2Sub < 3711.60 \wedge X1X2Sub < 5616.66\wedge$
	[0.10]$X1X2Sub < 7521.72$)
	class2()
	: −[0.84]([0.39]$X1X2Prod > 377422.12 \wedge$ [0.34]$X1X2Prod < 754844.25\wedge$
	$X1X2Sub > 3711.60 \wedge$ [0.29]$X1X2Sub < 7521.72\wedge$
	[0.94]$X1X2Sub < 9426.78$)
	: −[0.92]($X1X2Sub > 3711.60 \wedge X1X2Sub < 7521.72 \wedge$ [0.72]$X1X2Sub < 9426.78$)
	class3()
	: −[0.99]([0.13]$X1X2Prod < 566133.19 \wedge X1X2Sub > 7521.72\wedge$
	[0.26]$X1X2Sub > 9426.78 \wedge X1X2Sub < 11331.84\wedge$
	[0.10]$X1X2Sub < 13236.90$)
	class4()
	: −[0.59]($X1X2Sub > 11331.84 \wedge X1X2Sub < 17047.03$)
	: −[0.98]([0.86]$X1X2Sub > 9426.78 \wedge X1X2Sub > 11331.84 \wedge X1X2Sub < 17047.03$)
	class5()
	: −[0.91]($X1X2Sub > 15141.96 \wedge X1X2Sub > 17047.03$)
	: −[0.96]($X1X2Sub > 15141.96 \wedge X1X2Sub > 17047.03$)

$class3() \sim 8.26 \times 10^3 \leq \mathsf{F}(x) < 1.24 \times 10^4$, $class4() \sim 1.24 \times 10^4 \leq \mathsf{F}(x) < 1.66 \times 10^4$, and $class5() \sim 1.66 \times 10^4 \leq \mathsf{F}(x) \leq 2.07 \times 10^4$. Compared to other experiments, extraction of this equation required 5 discrete target predicates. Running the pipeline with 3 classification predicates resulted in the model failing to learn the true non-linear function. The accuracy for the second transformation layer ($X2$) is lower than the other layers, even though the model is confident in the presence of bounded atoms associated with the power function. The subtraction operation, was also a challenge for the operation layer to extract, as seen by the computational time in Table 6.

In Table 11 the dNL-NL layers extract the non-linear equation $((\mathbf{X1})^\mathbf{2} + \mathbf{exp}(\mathbf{X2}))$. The three discrete classes correspond to the following features ranges, $class1() \sim 1.94 \leq \mathsf{F}(x) < 6.93 \times 10^3$, $class2() \sim 6.93 \times 10^3 \leq \mathsf{F}(x) < 1.39 \times 10^4$, and $class3() \sim 1.39 \times 10^4 \leq \mathsf{F}(x) \leq 2.08 \times 10^4$. The transformation layers are able to extract the correct transformations, as seen by the various bounded predicates and their measure of confidence. The transformation on feature $X1$ achieved a lower accuracy, and the issue with modelling the discrete classes can be seen with the various bounded continuous predicates in the definitions of this layer. The lower performance of the ($X1$) can likely be attributed to the perfect performance of the second transformation layer ($X2$).

Table 11 Each dNL-NL layer's associated learned rules, where rules are defined by extracted non-linear predicates for the function $((X_1)^2 + exp(X_2))$ where the target was discretized into three classes and the variable space was floats in the range [0, 10]

$(\mathbf{X_1})^2 + \mathbf{exp}(\mathbf{X_2})$	Rules
Transformation: X1 accuracy: 0.875	**class1()** : $-[0.92]([0.29]X1Square < 45.25)$: $-[0.93]([0.25]X1Exp > 1897.02 \wedge [0.13]X1Exp < 5689.05 \wedge [0.19]X1Exp < 7585.06 \wedge [0.18]X1Square > 60.69)$ **class2()** : $-[0.55]([0.68]X1Square > 9.69 \wedge [0.42]X1Square > 27.08 \wedge X1Square < 36.26 \wedge [0.85]X1Sine > -0.43)$: $-[0.22]([0.74]X1Exp > 5689.05 \wedge [0.79]X1Exp < 13273.11 \wedge [0.94]X1Square > 27.08 \wedge [0.29]X1Square < 80.93)$ **class3()** : $-[0.25]([0.80]X1Square > 18.45 \wedge X1Square < 26.34 \wedge [0.34]X1Sine < 0.41 \wedge [0.21]X1Sine < 0.41)$: $-[0.38]([0.17]X1Exp > 7585.06 \wedge [0.34]X1Exp < 15169.12 \wedge [0.13]X1Square > 18.45 \wedge [0.79]X1Square > 45.10 \wedge [0.47]X1Square > 80.93 \wedge [0.72]X1Square < 60.73)$
Transformation: X2 accuracy: 1.00	**class1()** : $-[0.95]([0.27]X2Exp < 5867.07 \wedge X2Exp < 7822.37 \wedge [0.16]X2Square < 73.42 \wedge [0.92]X2Square < 81.03)$: $-[0.94]([0.25]X2Exp < 5867.07 \wedge X2Exp < 7822.37 \wedge [0.13]X2Square < 73.42 \wedge [0.92]X2Square < 81.03)$ **class2()** : $-[0.98](X2Exp > 5867.07 \wedge [0.74]X2Exp > 7822.37 \wedge [0.93]X2Exp < 15643.60 \wedge [0.30]X2Sine > -0.10)$ **class3()** : $-[0.95](X2Exp > 13688.29 \wedge [0.74]X2Exp > 15643.60 \wedge X2Square > 90.54)$: $-[0.92](X2Exp > 13688.29 \wedge [0.59]X2Exp > 15643.60 \wedge X2Square > 90.54)$
Operation: X1 $\wedge$ X2 accuracy: 1.00	**class1()** : $-[0.96]([0.15]X1X2Add < 5894.04 \wedge [0.22]X1X2Add < 7858.34 \wedge X1X2Sub > -7759.43 \wedge [0.17]X1X2Sub > -5795.13 \wedge [0.16]X2X1Sub < 5795.13 \wedge [0.92]X2X1Sub < 7759.43)$: $-[0.96]([0.21]X1X2Add < 5894.04 \wedge X1X2Add < 7858.34 \wedge [0.71]X1X2Sub > -7759.43 \wedge [0.21]X1X2Sub > -5795.13 \wedge [0.12]X2X1Sub < 5795.13 \wedge [0.37]X2X1Sub < 7759.43)$ **class2()** : $-[0.21](X1X2Add > 7858.34 \wedge [0.44]X1X2Add < 15715.54 \wedge [0.21]X1X2Sub > -15616.62 \wedge [0.73]X2X1Sub < 15616.62)$: $-[0.98]([0.10]X1X2Sub > -13652.33 \wedge [0.71]X1X2Sub < -7759.43 \wedge X2X1Sub > 5795.13 \wedge [0.95]X2X1Sub < 13652.33)$ **class3()** : $-[0.97](X1X2Add > 13751.24 \wedge [0.69]X1X2Prod > 193413.66 \wedge X1X2Sub < -13652.33)$: $-[0.96](X1X2Add > 13751.24 \wedge [0.25]X1X2Add > 15715.54 \wedge [0.31]X1X2Sub < -15616.62 \wedge [0.81]X2X1Sub > 13652.33 \wedge [0.25]X2X1Sub > 15616.62)$

Table 12 Each dNL-NL layer's associated learned rules, where rules are defined by extracted non-linear predicates for the function $(sine(X_1) - exp(X_2))$ where the target was discretized into three classes and the variable space was floats in the range [0, 10]

$\mathbf{sine(X_1) - exp(X_2)}$	Rules
Transformation: X1 accuracy: 0.85	**class2()** : $-[0.16]([0.48]X1Sine > 0.35 \wedge X1Sine < -0.11)$: $-[0.10](X1Sine > 0.35)$ **class3()** : $-[0.79]([0.11]X1Sine > -0.09 \wedge [0.12]X1Sine > -0.09)$: $-[0.75]([0.12]X1Sine > -0.09 \wedge [0.11]X1Sine > -0.09)$
Transformation: X2 accuracy: 0.975	**class1()** : $-[0.94](X2Exp > 13688.29 \wedge [0.40]X2Exp > 15643.60 \wedge X2Square > 90.56)$: $-[0.90](X2Exp > 13688.29 \wedge [0.44]X2Exp > 15643.60 \wedge X2Square > 90.56)$ **class2()** : $-[0.99](X2Exp > 5867.07 \wedge [0.73]X2Exp > 7822.37 \wedge [0.14]X2Exp < 15643.60 \wedge$ $[0.93]X2Square < 91.70)$ **class3()** : $-[0.95]([0.17]X2Exp < 5867.07 \wedge X2Exp < 7822.37 \wedge [0.23]X2Square < 73.09 \wedge$ $[0.92]X2Square < 81.07)$: $-[0.95]([0.34]X2Exp < 5867.07 \wedge X2Exp < 7822.37 \wedge [0.28]X2Square < 73.09 \wedge$ $[0.89]X2Square < 81.07)$
Operation: X1 $\wedge$ X2 accuracy: 1.00	**class1()** : $-[0.99](X1X2Prod < 5866.06 \wedge [0.76]X1X2Prod < 9776.73 \wedge$ $X1X2Sub < -13688.51)$: $-[0.99](X1X2Sub < -15644.01 \wedge X1X2Sub < -13688.51 \wedge$ $[0.90]X1X2Sub < -11733.01)$ **class2()** : $-[0.99]([0.48]X1X2Sub > -15644.01 \wedge [0.86]X1X2Sub > -13688.51 \wedge$ $[0.63]X1X2Sub < -7822.01 \wedge X1X2Sub < -5866.50)$ **class3()** : $-[0.95](X1X2Prod > -1955.29 \wedge [0.14]X1X2Sub > -9777.51 \wedge$ $X1X2Sub > -7822.01)$: $-[0.96](X1X2Sub > -7822.01 \wedge X1X2Sub > -5866.50)$

In Table 12 the dNL-NL layers extract the non-linear equation $(\mathbf{sin(X1) - exp(X2)})$. The three discrete classes correspond to the following features ranges, $class1() \sim -2.08 \times 10^4 \leq \mathsf{F}(x) < -1.39 \times 10^4$, $class2() \sim -1.39 \times 10^4 \leq \mathsf{F}(x) < -6.93 \times 10^3$, and $class3() \sim -6.93 \times 10^3 \leq \mathsf{F}(x) \leq -3.29 \times 10^{-1}$. The performance is again noted to be slower, as seen in Table 6. The subtraction operation, which also resulted in a slow performance for Table 10, struggles to be efficiently extracted by the operation layer, thus we propose future research to investigate improving the subtraction operation.

Future Research:

As indicated, there are areas of improvement for the dNL-NL functions and the overall pipeline. Specifically related to the logical reasoning of the dNL-NL layers, it was found that extracting subtraction operations was computationally expensive. This issue is potentially

related to the logical structure of the continuous Boolean predicates. Periodic functions such as the sine function also posed certain challenges when modelling the target predicates, and we leave this as an area of investigation for future research.

A more notable future investigation is that of the heuristics used to update the knowledge bases which determined which continuous predicates were available to the dNL-NL layers. In the pipeline proposed here, we rely on a simple heuristic of removing predicates based on their layer's accuracy scores. Future research might consider a modelling-based approach which uses accuracy scores and the true loss to train a model for selecting the various transformation and operation functions for continuous predicate instantiation.

The proposed architecture here focuses on demonstrating the capacity for ILP inspired models to extract non-linear functions from mixed-continuous domains. Our experiments focus on the two-variable case as a proof of concept. A further extension would be to consider three or more variables. This extension would also need to factor in the heuristics for updating the knowledge base, as well as the structuring of the transformation layers, exploring applications where more complex rules are needed and also coming up with a comparison strategy to relate to other learning approaches. To the best of our knowledge, a key issue in devising a comparison strategy is that little research is available on learning explainable non-linear models, such that we may be the first.

Given that, future work could reconfigure our dNL-NL model for a direct comparison with Duvenaud et al. (2013). A prospective extension to our dNL-NL framework could involve transitioning from our transformation/operation Boolean predicate functions to kernel Boolean predicate functions. By leveraging compositional techniques to define structures via kernels, there is potential to incorporate the method of creating composite kernel spaces from sums and products of base kernels into dNL-NL networks. This could entail embedding kernel predicates while retaining our operational Boolean predicates, enabling a comprehensive and direct model comparison.

While experiments in this proposal focused on transformation and operations between variables. Future extensions could apply constant continuous Boolean predicates, where a constant is multiplied on a random variable as seen in Eq. (14). Here the constant C could be a predetermined value provided by the modeller, or a trainable weight akin to the lower and upper bound values (u_{Cxi}, l_{Cxi}).

$$\mathcal{F}_{gt^i_{Cx}} = \sigma(c((C \times x) - u_{Cxi})), \quad \mathcal{F}_{lt^i_{Cx}} = \sigma(-c((C \times x) - l_{Cxi})) \tag{14}$$

6 Conclusion

We expand on a differentiable extension to ILP, so-called differentiable Neural Logic (dNL) networks, by focusing on the extraction of non-linear rules from synthetic data which has been generated by various non-linear functions. We provide a scheme for learning non-linear transformations and operations between variables using the logical framework of dNL and assessing the success of our derived logical rules by comparing them to the continuous target values. This work allows dNL networks to derive logical solutions in mixed discrete and continuous domains. Furthermore, from our results, we demonstrate that a pipeline of dNL networks can successfully extract non-linear functions from mixed discrete-continuous domains. For possible future extensions, we intend to investigate more extensive domain applications such as those common to other scientific disciplines. As there is no current work of a comparable nature, future

investigations will also look into possible model comparisons, as well as integration with reinforcement learning. The avenue of combing dNL-NL with reinforcement learning allows for applications in machine vision and possibly natural language processing.

Appendix A: Example input matrix and dNL-NL network

In discussing the dNL-NL framework, it is important to clarify the design of the deep network architecture in the context of linear transformations of the input variables. As an example, presume we are trying to extract non-linear functions for a simple dataset consisting of two random variables $\{X_1, X_2\}$ with continuous values and a target Y_c, where c denotes that the target is real-valued and continuous. In our example we will also include the following list of continuous transformation predicates $C_p = \{X_1, X_2, SqrX_1, SqrX_2, ExpX_1, ExpX_2\}$, and we will discretize Y_c into three distinct target predicate classes $\mathbb{P} = \{class_1, class_2, class_3\}$, done by performing equal-width binning on Y_c to derive our descretized target Y_d where $d = 3$. In this example, three discrete classes are selected arbitrarily, but note higher values can be used. We will also set our batch size $\mathbf{b} = 6$ and the number of continuous variable boundaries $k = 2$, note that these parameters can be adjusted. By setting $k = 2$ we are creating two lower and upper boundary predicates for each predicate in C_p. We start first by structuring our continuous input matrix $\mathbf{I}$, which is done for each $class_i \in \mathbb{P}$ but for our example we will discuss the Boolean deep architecture for $class_1$. In this example, as $N_e = 24$, we have an input matrix with dimensions (6, 24).

$$\mathbf{I}_{(6,24)} = \begin{bmatrix} \mathcal{F}_{lt^1_{X_1}} & \mathcal{F}_{lt^2_{X_1}} & \mathcal{F}_{gt^1_{X_1}} & \mathcal{F}_{gt^2_{X_1}} & \mathcal{F}_{lt^1_{X_2}} & \cdots & \mathcal{F}_{gt^2_{ExpX_2}} \\ \vdots & \ddots & & & & & \end{bmatrix} \tag{A1}$$

In Eq. (A1), $\mathbf{I}$ is composed of continuous Boolean predicate functions for each variable and each transformation on said variable. When values from $\mathcal{B}$ are passed into the input matrix, they are transformed by the Boolean predicate functions into a matrix of Boolean values, which the rest of the dNL architecture can reason on. Again, the lower and upper bound weights (l_i, u_i) can be optimised and will be adjusted during training. The weight values are initially set by taking the maximum and minimum from the continuous range for a given variable and dividing by k.

The process of creating our predicate function $\mathcal{F}_{class_1}$ for $class_1$ requires stacking conjunction neurons, and in our example, the number of conjunction stacks N_p is set to 2. In order to create our classification target predicate function, we need to create trainable weights as seen in Eq. (A2) where $(c = 2)$. The list of linear transformations for the conjunction layer can be seen in Eq. (A2).

$$\begin{aligned} \mathbf{W}^{conj}_{(2,24)} &\sim \mathbb{N} \\ \mathbf{m}^{conj}_{(2,24)} &= \sigma\left(\mathbf{W}^{conj}_{(2,24)} \times c\right) \\ \mathbf{Z}^{conj}_{(6,2,24)} &= [\mathbf{m}^{conj}_{(2,24)}]_{(1,2,24)} \times \left(1.0 - [\mathbf{I}_{(6,24)}]_{(6,1,24)}\right) \\ \mathbf{S}^{conj}_{(6,2)} &= \prod_{i=1}^{24} \left(1.0 - \mathbf{Z}^{conj}_{(6,2,i)}\right) \end{aligned} \tag{A2}$$

We use the brackets "[•]" to denote the extension of matrices with additional dimensions to facilitate matrix-to-matrix operations. Following the construction of the conjunction layer, we then stack disjunction layers based on the activations found in the conjunction layer, as seen in Eq. (A3).

$$\begin{aligned}
&\mathbf{W}^{disj}_{(1,2)} \sim \mathbb{N} \\
&\mathbf{m}^{disj}_{(1,2)} = \sigma\big(\mathbf{W}^{disj}_{(1,2)} \times c\big) \\
&\mathbf{Z}^{disj}_{(6,1,2)} = [\mathbf{m}^{disj}_{(1,2)}]_{(1,1,2)} \times [\mathbf{S}^{conj}_{(6,2)}]_{(6,1,2)} \\
&\mathcal{F}_{class_1(6,1)} = 1.0 - \prod_{i=1}^{2} \big(1.0 - \mathbf{Z}^{disj}_{(6,2,i)}\big)
\end{aligned} \tag{A3}$$

As our intensional target rules are defined by body definitions composed of extensional predicates, we only need to take a single step in forward chaining to ground the rule for our predicate $class_1$ during the current training iteration. The final aggregation of the groundings from the batch input defines the body of our target predicate function $\mathcal{F}_{class_1}$. The construction of the dNL-NL network is performed for each discrete classification (with corresponding predicate function) and eventually collected in a data structure $\mathbb{X}$.

Acknowledgements We wish to thank Nijesh Upreti for helping revise the manuscript, Moy Yuan for his coding contribution, Rafael-Michael Karampatsis for his advisory guidance in writing this manuscript, and Ionela Mocanu for feedback on an earlier draft.

Author contributions All authors contributed to the proposed framework's conception and design. Development, data collection and analysis were performed by AB and VB. The first draft of the manuscript was written by AB and all authors commented on previous versions of the manuscript. All authors read and approved the final manuscript.

Funding Vaishak Belle was supported by a Royal Society University Research Fellowship. He was also supported by a grant from the UKRI Strategic Priorities Fund to the UKRI Research Node on Trustworthy Autonomous Systems Governance and Regulation (EP/V026607/1, 2020-2024).

Availability of data and material https://github.com/acbueff/dNL-NL/tree/main/dNL-NL/data/continuous.

Code availability https://github.com/acbueff/dNL-NL.

Declarations

Conflict of interest The authors declare that they have no Conflict of interest.

Ethical approval Not Applicable.

Consent to participate Not Applicable.

Consent for publication Not Applicable.

References

Ahlgren, J., & Yuen, S. Y. (2013). Efficient program synthesis using constraint satisfaction in inductive logic programming. *Journal of Machine Learning Research, 14*, 3649–3682.

Balog, M., Gaunt, A. L., Brockschmidt, M., Nowozin, S., & Tarlow, D. (2017). Deepcoder: Learning to write programs.

Barredo Arrieta, A., Díaz-Rodíguez, N., Del Ser, J., Bennetot, A., Tabik, S., Barbado, A., & Herrera, F. (2020). Explainable artificial intelligence (XAI): Concepts, taxonomies, opportunities and challenges toward responsible AI. *Information Fusion, 58*, 82–115. https://doi.org/10.1016/j.inffus.2019.12.012

Belle, V., Van den Broeck, G., & Passerini, A. (2016). Component caching in hybrid domains with piecewise polynomial densities. AAAI (pp. 3369-3375).

Bueff, A., Speichert, S., & Belle, V. (2021). Probabilistic tractable models in mixed discrete-continuous domains. Data Intelligence, 228–260.

Chavira, M., & Darwiche, A. (2008). On probabilistic inference by weighted model counting. *Artificial Intelligence, 172*(6–7), 772–799.

Chollet, F. (2019). On the measure of intelligence.

Cropper, A., & Dumančic, S. (2020). Learning large logic programs by going beyond entailment. In C. Bessiere (Ed.), *Proceedings of the twenty-ninth international joint conference on artificial intelligence, IJCAI-20* (pp. 2073–2079). International Joint Conferences on Artificial Intelligence Organization. Retrieved from https://doi.org/10.24963/ijcai.2020/287 (Main track)

Cropper, A., & Morel, R. (2021). Learning programs by learning from failures. *Machine Learning, 110*(4), 801–856. https://doi.org/10.1007/s10994-020-05934-z

Cropper, A., & Morel, R. (2021b). Predicate invention by learning from failures.

d'Avila Garcez, A. S., Gori, M., Lamb, L.C., Serafini, L., Spranger, M., & Tran, S. N. (2019). Neural-symbolic computing: An effective methodology for principled integration of machine learning and reasoning. CoRR, arXiv:abs/1905.06088 . Retrieved from http://arxiv.org/abs/1905.06088

De Raedt, L., & Dehaspe, L. (1997). Clausal discovery. *Machine Learning, 26*, 99–146. https://doi.org/10.1023/A:1007361123060

Dheeru, D., & Karra Taniskidou, E. (2017). UCI machine learning repository. Retrieved from http://archive.ics.uci.edu/ml

Duvenaud, D., Lloyd, J. R., Grosse, R., Tenenbaum, J. B., & Ghahramani, Z. (2013). Structure discovery in nonparametric regression through compositional kernel search.

Ellis, K.,Wong, C., Nye, M., Sable-Meyer, M., Cary, L., Morales, L., & Tenenbaum, J. B. (2020). Dreamcoder: Growing generalizable, interpretable knowledge with wake-sleep Bayesian program learning.

Evans, R., & Grefenstette, E. (2017). Learning explanatory rules from noisy data. CoRR, arXiv:abs/1711.04574 . Retrieved from http://arxiv.org/abs/1711.04574

Gaunt, A. L., Brockschmidt, M., Singh, R., Kushman, N., Kohli, P., Taylor, J., & Tarlow, D. (2016). Terpret: A probabilistic programming language for program induction.

Hitzler, P., & Sarker, K. (2022). Neuro-symbolic artificial intelligence: The state of the art (Vol. 342). Frontiers in Artificial Intelligence and Applications.

Hocquette, C., & Cropper, A. (2023). Learning programs with magic values. *Machine Learning, 112*(5), 1551–1595. https://doi.org/10.1007/s10994-022-06274-w

Kersting, K., De Raedt, L., & Kramer, S. (2000). Interpreting bayesian logic programs. In *Proceedings of the AAAI-2000 workshop on learning statistical models from relational data* (pp. 29–35).

Kimmig, A., Bach, S., Broecheler, M., Huang, B., & Getoor, L. (2012). A short introduction to probabilistic soft logic. Nips workshop on probabilistic programming: Foundations and applications (pp. 1–4). Retrieved from https://linqs.soe.ucsc.edu/sites/default/files/papers/pslpp12.pdf

Krishnan, G. P., Maier, F., & Ramyaa, R. (2021). Learning rules with stratified negation in differentiable ILP. In *Advances in programming languages and neurosymbolic systems workshop*. Retrieved from https://openreview.net/forum?id=BOtQHCVIhK

Muggleton, S., Dai, W.-Z., Sammut, C., Tamaddoni-Nezhad, A., Wen, J., & Zhou, Z.-H. (2018). Meta-interpretive learning from noisy images. *Machine Learning, 107*, 1–22. https://doi.org/10.1007/s10994-018-5710-8

Muggleton, S., & de Raedt, L. (1994). Inductive logic programming: Theory and methods. *The Journal of Logic Programming, 19*, 629–679. https://doi.org/10.1016/0743-1066(94)90035-3

Muggleton, S., & De Raedt, L. (1994). Inductive logic programming: Theory and methods. *The Journal of Logic Programming, 19*, 629–679.

Muggleton, S. H. (1995). Inverse entailment and progol. *New Generation Computing, 13*, 245–286.

Nitti, D., De Laet, T., & De Raedt, L. (2016). Probabilistic logic programming for hybrid relational domains. *Machine Learning, 103*(3), 407–449.

Payani, A., & Fekri, F. (2019). Inductive logic programming via differentiable deep neural logic networks. CoRR, arXiv:abs/1906.03523 . Retrieved from http://arxiv.org/abs/1906.03523

Quinlan, J. R. (1990). Learning logical definitions from relations. *Machine Learning, 5*, 239–266.

Ray, O. (2009). Nonmonotonic abductive inductive learning. *Journal of Applied Logic, 7*(3), 329–340. https://doi.org/10.1016/j.jal.2008.10.007

Richardson, M., & Domingos, P. (2006). Markov logic networks. *Machine learning, 62*(1), 107–136.

Sen, P., de Carvalho, B. W. S. R., Riegel, R., & Gray, A. (2021). Neuro-symbolic inductive logic programming with logical neural networks.

Shindo, H., Nishino, M., & Yamamoto, A. (2021). Differentiable inductive logic programming for structured examples.

Speichert, S., & Belle, V. (2018). Learning probabilistic logic programs in continuous domains.

Srinivasan, A. (2001). The Aleph Manual [Computer software manual]. Retrieved from http://web.comlab.ox.ac.uk/oucl/research/areas/machlearn/Aleph/

Srinivasan, A., & Camacho, R. (1999). Numerical reasoning with an ILP system capable of lazy evaluation and customized search. *The Journal of Logic Programming, 40*(2), 185–213. https://doi.org/10.1016/S0743-1066(99)00018-7

Yang, Y., & Song, L. (2019). Learn to explain efficiently via neural logic inductive learning.