Differentiable Rule Induction with Learned Relational Features

Remy Kusters[1,2,*], Yusik Kim[1,2], Marine Collery[2,3], Christian de Sainte Marie[2] and Shubham Gupta[1,2]

[1]*IBM Research, Orsay, France*
[2]*IBM France Lab, Orsay, France*
[3]*Inria Saclay Ile-de-France, Palaiseau, France*

Abstract

Rule-based decision models are attractive due to their interpretability. However, existing rule induction methods often result in long and consequently less interpretable rule models. This problem can often be attributed to the lack of appropriately expressive vocabulary, i.e., relevant predicates used as literals in the decision model. Most existing rule induction algorithms presume pre-defined literals, naturally decoupling the definition of the literals from the rule learning phase. In contrast, we propose the Relational Rule Network (R2N), a neural architecture that learns literals that represent a linear relationship among numerical input features along with the rules that use them. This approach opens the door to increasing the expressiveness of induced decision models by coupling literal learning directly with rule learning in an end-to-end differentiable fashion. On benchmark tasks, we show that these learned literals are simple enough to retain interpretability, yet improve prediction accuracy and provide sets of rules that are more concise compared to state-of-the-art rule induction algorithms.

Keywords

Decision rules, Rule learning, Feature learning

1. Introduction

Over the last decade, black box decision models (e.g. neural networks) are increasingly used in high stakes decision making, yet there has been equally growing concern over their use [1]. Regulations in various industries are demanding accountability and transparency from the models involved in the decision making process. Besides the regulatory concerns, the end-users of many complex decision models are also increasingly demanding interpretability of the final decision. Rule based models are a potential solution but, to date, most rule-based decision support systems do not support learning practically useful decision models *directly from the data*, often due to the resulting rules being too long to be considered interpretable [2, 3]. Focusing on learning rule models which model higher-level concepts expressed in terms of the lower-level input features, rather than rule models that *only* use these lower-level input features has been brought forward as one of the primary avenues to improve the expressiveness and interpretability of rule models [2, 4]. The scope of this paper is learning

rule models, expressed in Disjunctive Normal Form (DNF), for classification tasks of tabular data. Of particular interest is to enrich the representation language used by rule models with higher-level latent representations which are functions of lower-level input features. This is in contrast to most state-of-the-art rule learning algorithms which use pre-defined literals (either hand-crafted or obtained from a-priori binarized features) as literals in the rule model (e.g., BRCG [5], RIPPER [6], CORELS [7]). The main difficulty of learning the appropriate literals is in incorporating feedback from the rule model that uses them. One way to achieve this is through learning the literals and the rule model that uses them *jointly* via a single differentiable neural architecture.

We propose the *Relational Rule Network (R2N)*: a three-layer neural network which *learns literals* in the first layer, each representing a partition of the input feature space delimited by a hyperplane, which we call a *halfspace*. The second and third layers map the binary vector of literals to a binary predictions, encoding a crisp logical formula in the form of a DNF. We consider halfspaces as literals as they are simple enough to retain interpretability, yet significantly improve expressivity and model accuracy. Encoding this structure in an end-to-end differentiable neural network has the benefit that the learning of the literals is directly informed by their usage in the rule. To showcase the value of using appropriate predicates as literals in a rule model, we present a simple toy example: Given a dataset with two numerical input features (x_0, x_1) and a corresponding label y determined by the ground truth

$$\text{if}\,(x_0/x_1 > 1.0 \wedge x_0 > 0.5)\ \text{then class = True; else class = False;} \tag{1}$$

the task is to find a rule model that accurately predicts the label given the input. This model involves two distinct decision boundaries: $x_0 = 0.5$ and $x_0/x_1 = 1.0$. Training our model with default parameters (see Appendix C) recovers the ground truth (see Fig. 1) whereas when literals are predefined as univariate value comparisons (e.g. $x_0 < 0.2$), we learn a longer DNF. This is because each conjunction represents a rectangle in the feature space (x_0, x_1), and many of them are needed to adequately approximate the sloped decision boundary, $x_0/x_1 = 1$ (Fig. 1b). Therefore, by improving the expressivity of the rule model with the ability to represent ratio as a single literal rather than a conjunction of many, we simplified the model and improved accuracy.

We summarize our contribution as follows: i) We propose a neural network architecture (R2N) that learns a DNF together with the literals it uses. The literals represent halfspaces, whose boundary hyperplanes are *learned* by the network. ii) We demonstrate that compared to SOTA algorithms (RIPPER, BRCG, DR-Net, etc.) we improve sparsity and interpretability of the learned rule model both for a set of benchmark problems where the underlying decision models are known as well as where they are not.

2. Related works

Our work contributes to the large body of work on neuro-symbolic rule learning systems that represent atoms and clauses as individual nodes in a neural network (NN) and that learn rules as configurations of weights on the edges that link these nodes. This approach is different from existing *distillation*-based approaches that construct a simpler interpretable network to mimic
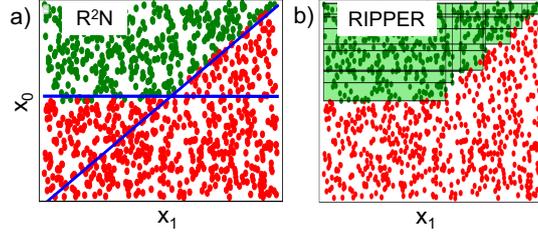
Figure 1: Decision boundaries for Eq. (1) (a) R2N, and (b) the RIPPER algorithm [6]. The decision boundaries learned by R2N (a) correspond to $x_0 > 0.5$ and $x_1/x_0 > 1.0$, while the RIPPER algorithm (b) learns nine rules that correspond to the nine rectangles inside the green area.

the behavior of a complex black-box model [8, 9]. While they generate post-hoc explanations that only approximate a model, R2N and the like natively learn explainable rules: the rules do not approximate the NN, they are equivalent to it by construction.

The representation of rules in R2N is essentially the same as in C-IL2P [10] and followers (e.g. [11, 12, 13]). R2N differs from C-IL2P and similar systems in two main ways: whereas they focus on approximating logic programs and their semantics in discrete domains (although infinite and continuous domains can also be dealt with, it remains in a probabilistic setting [14]), R2N learns crisp classification rules in continuous numerical domains, like [15, 16]. However, our approach differs from [15] in two regards: i) the encoding and ii) the binarization of the rules as we discuss in Section 4.

In addition, R2N learns a set of literals that map the *n*-dimensional continuous numerical input space to an *m*-dimensional Boolean space in which the rule models are learned. Similar neuro-symbolic systems require binarized input data using predefined predicates [10, 15, 16, 17]; are limited to learning univariate threshold values [18]; or learn latent relations in a finite Herbrand base [4, 13, 19]. The latter systems rely on prior knowledge about the relations to be learned [4, 19], whereas R2N does not require prior knowledge. Rather than predicate invention, learning literals in R2N can be considered as a form of propositionalization [20]. Contrary to other systems relying on it [12, 20, 21], propositionalization is intrinsically part of the rule learning in R2N: indeed, R2N is able to learn a more useful grounding of the learned literals in the input space without prior knowledge because the literals are learned as part of the supervised rule learning process. This is similar to the way DeepProbLog uses the semantics of the program to supervise the grounding of predicates [22].

Extending the branching conditions from simple value comparisons to linear halfspaces has been proposed for decision trees [23]. While it is possible to translate the resulting decision tree to a DNF, such mapping usually results in excessively long rules, hindering interpretability. Furthermore, single literal learning layer that grounds the learned literals using hyperplanes in R2N can be replaced with an arbitrarily complex NN (learning hypersurfaces) at the cost of interpretability.
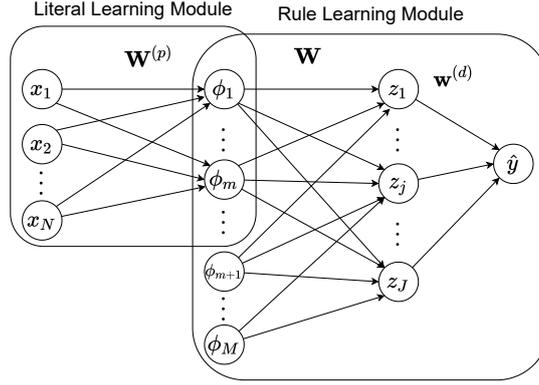
Figure 2: Relational Rule Network (R2N): The literal learning module encodes halfspaces and evaluates to Boolean literals ϕ. Alongside these learned literals, zero or more predefined literals $\phi_{m+1}, ..., \phi_M$ are supplied directly to the rule learning module. The rule learning module uses these literals in the conjunctions z in the first layer. In the final layer, the relevant conjunctions are selected into a DNF.

3. Rule language

The rule model that we consider for binary classification is a DNF, whose formal grammar is defined by the following production rules,

$$\texttt{rulemodel} \rightarrow \text{if } \texttt{DNF} \text{ then } \text{class = True; else class = False}$$

$$\texttt{DNF} \rightarrow \texttt{conjunction} \,|\, \texttt{conjunction} \vee \texttt{DNF}$$

$$\texttt{conjunction} \rightarrow \texttt{literal} \,|\, \texttt{literal} \wedge \texttt{conjunction}$$

$$\texttt{literal} \rightarrow \underline{\phi_1} \,|\, ... \,|\, \underline{\phi_m} \,|\, \underline{\phi_{m+1}} \,|\, ... \,|\, \underline{\phi_M}$$

The underlined terms are terminal symbols, and the typewriter terms are the non-terminal symbols. Among the predicates (of arbitrary arity) in our dictionary of literals, some have learnable parameters, $\{\underline{\phi_1},...,\underline{\phi_m}\}$, e.g., coefficients of a hyperplane defining a halfspace, and some do not, $\{\underline{\phi_{m+1}},...,\underline{\phi_M}\}$, meaning they are pre-determined (see Fig. 2). We call the subset of literals with learnable parameters *learned literals*, and those without learnable parameters *predefined literals*. Note that negations are not part of the rule language. Therefore, negated predicates, if desired, must be explicitly included in the dictionary of literals.

4. The Relational Rule Network (R2N)

The Relational Rule Network (R2N) is composed of two modules connected sequentially:

1. The **literal learning module** that learns halfspaces to be used as literals of the rule language.
2. The **rule learning module** that maps the binary vector of evaluated literals to a single binary prediction of the class label, in such a way that it encodes an equivalent logical formula in disjunctive normal form (DNF).

Any predefined literals we want to use, including categorical value comparisons, are directly supplied to the input layer of the rule learning module. See Fig. 2 for an illustration.

We introduce the following notation for the rest of this paper:

- x_i: Numerical input variable i.
- $\phi_i(\mathbf{x})$: The i-th literal in our rule language. Learned literals ($i = 1, ..., m$) represent half-spaces in the N-dimensional space of $\mathbf{x}$. We omit the argument when the context is clear.
- z_j: The j-th conjunction formed by a combination of literals.

Unless stated otherwise, normal-faced lower-case variables with subscripts (e.g., x_i) represent scalar components of its bold-faced counterparts, representing vectors (lower-case; $\mathbf{x}$) or matrices (upper-case; $\mathbf{X}$). All vectors are column vectors.

4.1. The literal learning module

To enrich the vocabulary of our rule language, we consider a neural network layer that learns predicates representing halfspaces of the numerical feature space. These learned predicates can then be used as literals in the rule language. The literal learning module is represented by a single layer perceptron with a learnable bias b_i and weight vector $\mathbf{w}_j^{(p)}$ for each halfspace ϕ_i that we wish to learn (See Fig. 2). Extending this to multiple output nodes using matrix notation, the transformation is represented by

$$\boldsymbol{\phi}(\mathbf{x}) = \sigma\left(\frac{\mathbf{x}^T \mathbf{W}^{(p)} + \mathbf{b}}{\tau}\right), \tag{2}$$

where $\mathbf{W}^{(p)}$ is the weight matrix, $\mathbf{b}$ the corresponding biases, σ is the sigmoid activation function applied component-wise, and τ is a temperature parameter that changes according to a cooling schedule. Observing that $\sigma(x/\tau)$ converges[1] to the Heaviside step function as $\tau \to 0$, we effectively approach strict binarization at the end of the cooling schedule (step function is used during testing), while avoiding zero gradients during training.

4.2. The rule learning module

The rule learning module maps the binary vector of literals to a single binary prediction of the class label, in such a way that it encodes an equivalent logical formula in DNF. We specify this rule learning module as a two-layer neural network architecture respectively implementing an **AND** and a subsequent **OR** operation, similar to [10, 15].

AND-layer The **AND**-layer maps a binary vector of literals $\boldsymbol{\phi} = (\phi_1, ..., \phi_M)$ to another binary vector $(z_1, ..., z_J)$ whose components represent conjunctions made from some combination of the input literals. This combination is specified by a binary vector $\mathbf{w}_j$ indicating which of the M literals are present in conjunction z_j. The i-th component w_{ij} of $\mathbf{w}_j$ is 1 if literal ϕ_i is included in conjunction z_j and 0 otherwise [16].

---

[1] It pointwise converges on $\mathbb{R} \setminus \{0\}$.

Proposition 1. *Let w_{ij} be binary parameter and ϕ_i be Boolean variables whose values can be represented by binary values. Then the mappings*

$$\phi \mapsto \bigwedge_{i\,:\,w_{ij}=1} \phi_i \quad and \quad \phi \mapsto 1 - \min\left\{\sum_i w_{ij}(1-\phi_i), 1\right\} \tag{3}$$

are equivalent. Proof: See Appendix A

The **AND**-layer implements transformation (3) as a computational graph. Although the parameter we want to eventually learn is the binary weight matrix **W**, we learn it indirectly through reparameterizing it with a scaled sigmoid function that converges to a Heaviside step function in the limit, similarly as in the literal learning module. The learnable parameter u_{ij} for each binary weight w_{ij} is continuous and unconstrained:

$$w_{ij} = \sigma(u_{ij}/\tau). \tag{4}$$

The resulting computational graph defined by equations (3) and (4) is differentiable with non-zero derivatives. Note that ensuring non-zero gradients during the backwards pass, rather than using a straight through estimator, as in [15], is essential to ensure robust training, as described in [24].

OR-layer The **OR**-layer maps a binary vector **z** to a single binary value by implementing a logical disjunction operation among select components of **z**. The binary weights $\mathbf{w}^{(d)}$ of this layer encode which of the conjunctions learned in the **AND**-layer are to be included in the final disjunction. We seek a differentiable approximation that approaches the non-differentiable **OR** operation in the limit, similar to what we have done for the **AND**-layer.

Proposition 2. *Let $w_j^{(d)}$ be binary parameters and z_j be Boolean variables. Then the mappings*

$$\mathbf{z} \mapsto \bigvee_{j\,:\,w_j^{(d)}=1} z_j \quad and \quad \mathbf{z} \mapsto \max\left\{w_j^{(d)} z_j; j = 1, ..., J\right\} \tag{5}$$

are equivalent. Proof: See Appendix B

The binary weight vector $\mathbf{w}^{(d)}$ is learned exactly the same way as we learn the weight matrix **W** of the **AND**-layer, through reparameterizing the binary weights using a scaled sigmoid:

$$w_j^{(d)} = \sigma(v_j/\tau). \tag{6}$$

The **OR**-layer implements the computational graph defined by equations (5) and (6), where v_j are the continuous and free learnable parameters.

Decoding the DNF from the trained network The 2-layer network composing the **AND** and **OR** layers can be represented as a composition of the two differentiable functions (3) and

(5). As a direct result of Propositions 1 and 2, this network implements the evaluation of the DNF

$$\phi \mapsto \bigvee_{j\,:\,w_j^{(d)}=1} \bigwedge_{i\,:\,w_{ij}=1} \phi_i. \tag{7}$$

The binary weights w_{ij} and $w_j^{(d)}$ can be retrieved by inspecting the sign of the learned parameters u_{ij} and v_j.

Note, however, that during training the network weights are not binary when the temperature is different from 0. Hence proposition 1 is not strictly applicable at training time, so there is no guarantee that at this stage the network output and the output of the decoded DNF are equivalent. However, we can make the outputs arbitrarily close to each other by driving the temperature down to zero. We observed that the difference between the network output and the output of the decoded DNF is negligible for reasonably cold temperatures ($\tau = 10^{-4}$). Therefore at the end of training we learn a crisp DNF as classifcation rule.

5. Results

Results with known ground truth									
Examples	R2N		DR-Net		RIPPER		BRCG		MLP
TRUE IF	Acc.	n_r, l_r	Acc.	n_r, l_r	Acc.	n_r, l_r	Acc.	n_r, l_r	Acc
1) $(x_0 > 0.25) \vee (x_1 < 0.5)$	1.0	2, 1.0	0.99	28, 1.6	0.99	5, 2.2	1.0	2, 1.0	1.0
2) $(x_0/x_1 > 0.5) \vee (x_4 < 0.25)$	1.0	2, 1.5	0.99	39, 2.1	0.99	22, 2.8	0.98	6, 1.5	1.0
3) $(x_0 + 0.5x_1 > 0.5) \vee (x_4 < 0.25)$	0.99	3, 1.7	0.98	40, 2.3	0.99	15, 2.7	0.97	6, 1.7	1.0
4) $(x_0 < 0.2 \wedge x_1 + x_2 > 0.5) \vee$ $(x_4/x_3 > 1 \wedge x_1 < 0.5)$	0.99	4, 2.3	0.98	40, 3.7	0.98	23, 4.8	0.94	5, 2.2	1.0
5) $(x_4 < 0.2 \wedge x_0/x_1 > 0.5) \vee$ $(0.5x_3 + 0.2x_1 > 0.5) \vee (x_0 < 0.2)$	0.99	3, 1.3	0.98	39, 2.0	0.98	15, 3.5	0.97	6, 1.8	0.99
Results with unknown ground truth									
Examples	R2N		DR-Net*		RIPPER*		BRCG*		MLP*
Magic	0.86	6, 3.3	0.84	18, 5.2	0.82	27, 6.0	0.84	22, 3.7	0.87
Heloc	0.72	3, 2.0	0.70	2, 6.3	0.70	12, 5.2	0.69	1, 1.9	0.71
Adult	0.83	2, 3.5	0.83	6, 13.5	0.82	20, 4.7	0.83	24, 3.8	0.84
House	0.86	5, 2.2	0.86	12, 6.3	0.82	41, 7.0	0.84	5, 5.2	0.89

Table 1
Accuracy, number of conjunctions in the DNF (n_r) and average number of literals per conjunction (l_r) for (top) the examples with known ground truth and (bottom) four datasets taken from the UCI ML database. Note that for the columns with an *, the benchmark accuracy and (n_r, l_r) are taken from [15] to ensure a fair comparison.

In this section we study the accuracy and rule complexity for a set of benchmark problems (i) where the underlying DNF is known and (ii) where this is not the case. We compare with three SOTA methods: RIPPER [6], DR-Net [15] and BRCG [5]. The hyperparameters, training routine and benchmarks are discussed in the Appendix F.

We first consider five simple DNFs with increasing complexity (see Table 1) and compare the prediction accuracy, number of conjunctions in the DNF (n_r) and average number of literals per

conjunction (l_r) of the R2N with SOTA methods. We generate a dataset of 10^4 samples from the underlying rules where the five input features ($x_0, ..., x_4$) are randomly sampled between $[0, 1]$. For all examples, R2N obtains near perfect accuracy ($> 99\%$) and the DNF we obtain approximate the ground truth very well. E.g., for the most complex example, Ex. 5, we obtain the DNF $(0.2x_1 + 0.5x_3 > 0.5) \vee (x_0 < 0.2) \vee (x_0/x_1 > 0.59 \wedge x_4 < 0.2)$. This example shows that R2N has not only obtained a compact rule model, but has also learned the ground truth literals involving the linear relationships and ratios of input features. The DNF we obtain for the other examples are detailed in Appendix E.

The three benchmark methods (RIPPER, DR-Net and BRCG) all require binarizing the numerical input features, hence trading off accuracy and length of the optimal DNF (fewer pre-defined literals will result in more compact rule models, but cruder approximations of the decision boundary). We selected the default parameters to make a fair comparisons in this case (See Appendix F for a detailed description), but these methods inherently require this accuracy/DNF complexity trade-off. RIPPER and DR-Net provide a comparable accuracy, yet provide DNFs that are considerably longer than R2N while BRCG provides more compact DNFs, but compromises significantly on the prediction accuracy. Note that this discrepancy is particularly important when the decision boundaries cannot be captured by univariate value comparisons as pre-defined literals (Ex. 2-5 in Table 1). In all cases, the rule language is not expressive enough to describe the linear decision boundary accurately with a sparse set of univariate literals.

Next we consider four datasets for which the underlying rule models are unknown (Taken from the UCI ML-repository, [25]). Note that besides the numerical attributes, these datasets contain categorical data which are added as additional input to the rule learning module using one-hot encoding as described in Section 4. Similarly as for the rule models with known underlying DNF, we obtain significantly shorter DNFs compared to all the SOTA models (see Table 1). This shows that even though we do not have a priori information of the presence of multi-variate literals, R2N provides more concise rulesets while outperforming SOTA methods on prediction accuracy. The accuracy we obtain with R2N approaches that of a Multi Layer Perceptron[2].

In Appendix G and D we show the sensitivity of our approach with respect to the size of the dataset/noise level and sparsity parameters. Overall we show that the obtained rule model and its near-optimal accuracy can still be obtained for noise-levels (randomly flipping a fraction of the labels in the dataset) up to 10% and for training set sizes of the order of 10^3 training samples.

6. Outlook

The method presented in this paper extends the vocabulary of rule-learning algorithms by learning literals that represent halfspaces. R2N can also serve in other applications as part of the domain vocabulary (or ontology); for instance as input predicates for other rule models like RIPPER or BRCG. We show that using these *learned* literals from R2N as input for RIPPER reduces the number of rules that it learns as compared to RIPPER with univariate value comparisons for all examples listed in Table 1 (see Appendix H). Note that while in general rules with fewer

---

[2]Comparisons are taken directly from [15]

conjunctions and literals per conjunction can be considered more easily interpretable, true interpretability might have to be examined by subject matter experts.

Although learning literals defined through hyperplanes has been shown to be a good first step towards improving the expressiveness of the rules while retaining interpretability, they are by no means sufficient for all practical applications. The framework we present can be extended to non-linear decision boundaries. A simple yet powerful example of this involves black-box function approximators (e.g. MLPs and LSTMs) to learn non-linear literals using our general machinery from Section 4 for retaining differentiability. While this results in literals that are not explicitly interpretable, the final DNF that use these literals can still be understood and audited by humans who can then try to understand the learned literals in light of the rules that use them. As a concrete example, consider an application where the input data is sequential and the ground-truth class assignments depend on complex aggregates such as variance (e.g., $var(c) > \theta_1 \Rightarrow True$) or counts (e.g. $count(x_t > \theta_2) \leq \theta_3 \Rightarrow False$). Halfspace predicates are inadequate for representing these relationships. However LSTMs can learn binary "non-linear" predicates from the data, that can then be used as literals in the rule model. Post-hoc analysis of these literals can then shed light on their meaning, reducing the complex problem of interpreting the entire model to a simpler problem of interpreting individual literals in the context of an a priori understandable rule model.

Acknowledgments

References

[1] J. Angwin, J. Larson, S. Mattu, L. Kirchner, Machine bias, https://www.propublica.org/article/machine-bias-risk-assessments-in-criminal-sentencing, 2016. URL: https://www.propublica.org/article/machine-bias-risk-assessments-in-criminal-sentencing, accessed: 2021-12-21.

[2] C. de Sainte Marie, Learning decision rules or learning decision models?, in: International Joint Conference on Rules and Reasoning, Springer, 2021, pp. 276–283.

[3] S. Kramer, A brief history of learning symbolic higher-level representations from data (and a curious look forward), in: C. Bessiere (Ed.), Proceedings of the Twenty-Ninth International Joint Conference on Artificial Intelligence, IJCAI-20, International Joint Conferences on Artificial Intelligence Organization, 2020, pp. 4868–4876. URL: https://doi.org/10.24963/ijcai.2020/678. doi:10.24963/ijcai.2020/678, survey track.

[4] S. Dumančić, H. Blockeel, Demystifying relational latent representations, in: International Conference on Inductive Logic Programming, Springer, 2017, pp. 63–77.

[5] S. Dash, O. Günlük, D. Wei, Boolean decision rules via column generation, in: Proceedings of the 32nd International Conference on Neural Information Processing Systems, NIPS'18, Curran Associates Inc., Red Hook, NY, USA, 2018, p. 4660–4670.

[6] W. W. Cohen, Fast effective rule induction, in: Machine learning proceedings 1995, Elsevier, 1995, pp. 115–123.

[7] E. Angelino, N. Larus-Stone, D. Alabi, M. Seltzer, C. Rudin, Learning certifiably optimal rule lists for categorical data, Journal of Machine Learning Research 18 (2018) 1–78. URL: http://jmlr.org/papers/v18/17-716.html.

[8] G. Hinton, O. Vinyals, J. Dean, Distilling the knowledge in a neural network, NIPS Deep Learning and Representation Learning Workshop (2015).

[9] J. R. Zilke, E. L. Mencía, F. Janssen, Deepred–rule extraction from deep neural networks, in: International Conference on Discovery Science, Springer, 2016, pp. 457–473.

[10] A. S. Avila Garcez, G. Zaverucha, The connectionist inductive learning and logic programming system, Applied Intelligence 11 (1999) 59–77.

[11] M. V. França, G. Zaverucha, A. S. d'Avila Garcez, Fast relational learning using bottom clause propositionalization with artificial neural networks, Machine learning 94 (2014) 81–104.

[12] K. Gao, K. Inoue, Y. Cao, H. Wang, Learning first-order rules with differentiable logic program semantics, arXiv preprint arXiv:2204.13570 (2022).

[13] G. Marra, M. Diligenti, F. Giannini, M. Gori, M. Maggini, Relational neural machines, arXiv preprint arXiv:2002.02193 (2020).

[14] V. Belle, Symbolic logic meets machine learning: A brief survey in infinite domains, in: International Conference on Scalable Uncertainty Management, Springer, 2020, pp. 3–16.

[15] L. Qiao, W. Wang, B. Lin, Learning accurate and interpretable decision rule sets from neural networks, Proceedings of the AAAI Conference on Artificial Intelligence 35 (2021) 4303–4311. URL: https://ojs.aaai.org/index.php/AAAI/article/view/16555.

[16] F. Beck, J. Fürnkranz, An investigation into mini-batch rule learning, arXiv preprint arXiv:2106.10202 (2021).

[17] L. Fu, Rule generation from neural networks, IEEE Transactions on Systems, Man, and Cybernetics 24 (1994) 1114–1124. doi:10.1109/21.299696.

[18] Z. Wang, W. Zhang, N. Liu, J. Wang, Scalable rule-based representation learning for interpretable classification, Advances in Neural Information Processing Systems 34 (2021).

[19] G. Sourek, V. Aschenbrenner, F. Zelezny, O. Kuzelka, Lifted relational neural networks, arXiv preprint arXiv:1508.05128 (2015).

[20] S. Kramer, E. Frank, Bottom-up propositionalization., in: ILP Work-in-progress reports, 2000.

[21] M.-A. Krogel, S. Rawles, F. Železný, P. A. Flach, N. Lavrač, S. Wrobel, Comparative evaluation of approaches to propositionalization, in: International Conference on Inductive Logic Programming, Springer, 2003, pp. 197–214.

[22] R. Manhaeve, S. Dumancic, A. Kimmig, T. Demeester, L. De Raedt, Deepproblog: Neural probabilistic logic programming, Advances in Neural Information Processing Systems 31 (2018).

[23] H. Zhu, P. Murali, D. Phan, L. Nguyen, J. Kalagnanam, A scalable mip-based method for learning optimal multivariate decision trees, in: Advances in Neural Information Processing Systems, volume 33, Curran Associates, Inc., 2020, pp. 1771–1781. URL: https://proceedings.neurips.cc/paper/2020/file/1373b284bc381890049e92d324f56de0-Paper.pdf.

[24] Y. Bengio, N. Léonard, A. Courville, Estimating or propagating gradients through stochastic

neurons for conditional computation, arXiv preprint arXiv:1308.3432 (2013).

[25] D. Dua, C. Graff, UCI machine learning repository, 2017. URL: http://archive.ics.uci.edu/ml.

A. Proof of Proposition 1

As the functions are binary-valued, it suffices to show that one function evaluating to 1 is equivalent to the other function evaluating to 1.

$$\bigwedge_{i\,:\,w_{ij}=1} \phi_i = 1 \iff \forall i \ (w_{ij} = 1) \to (\phi_i = 1)$$

$$\iff \forall i \ (w_{ij} = 0) \vee (\phi_i = 1)$$

$$\iff \forall i \ w_{ij}(1 - \phi_i) = 0$$

$$\iff \sum_i w_{ij}(1 - \phi_i) = 0 \tag{8}$$

$$\iff 1 - \min\left\{\sum_i w_{ij}(1 - \phi_i), 1\right\} = 1 \tag{9}$$

B. Proof of Proposition 2

$$\bigvee_{j\,:\,w_j^{(d)}=1} z_j = 1 \iff \exists j \ w_j^{(d)} = 1 \wedge z_j = 1 \tag{10}$$

$$\iff \max\left\{w_j^{(d)} z_j; j = 1, ..., J\right\} = 1 \tag{11}$$

C. Loss function and training

Our loss function has a mean-square error component together with components that promote sparsity of the obtained rule model. The overall loss function is:

$$\mathscr{L}_{mse} + \lambda_{and}\|\mathbf{W}\|_1 + \lambda_{or}\|\mathbf{w}^{(d)}\|_1 + \lambda_p\|\mathbf{W}^{(p)}\|_1 \tag{12}$$

where $\|\cdot\|_1$ is the 1-norm. Increasing λ_p reduces the sparity of the learned literals, increasing λ_{and} shortens conjunctions, and increasing λ_{or} shortens the disjunction. For simplicity, we choose a single regularization penalty for all the different layers $\lambda = \lambda_{and} = \lambda_{or} = \lambda_p$. Unless stated otherwise, we use $\lambda = 10^{-2}$ for training on synthetic datasets (with known ground truth) and a more conservative value of $\lambda = 10^{-3}$ for the UCI ML datasets [25].

We use the Adam optimizer from the PyTorch ecosystem with default learning rate and parameters to optimize Eq. 12. In order to avoid local minima throughout the training routine, we randomly re-initialize the weights of the neural network every 10^4 epochs and run the network for 2.5×10^5 epochs in total. The training is done with a fixed batch size of 100. We select the epoch with the lowest loss function on the training set and report the prediction accuracy at that stage for the test-set (train/test split of 80/20).

As mentioned in the section on the literal learning module, to ensure smooth training as well as reaching binary output of the literal learning layer, we propose a cooling schedule of the sigmoid activation function with high temperature in the beginning allowing larger gradients to exist across a wider range of the input so that training can occur. As the temperature approaches zero, the sigmoid is progressively scaled and approaches the Heaviside function in the limit, so that we achieve strict binarization in the end. The temperature is cooled down with a factor $\gamma = 0.995$ at every subsequent epoch for all the examples presented in the paper.

The number of output nodes of the literal learning module $m = 10$ while the number of nodes in the rule learning module is $J = 25$. Increasing or decreasing these numbers would impact the expressivity of the network but we empirically observed that varying this number in the range 10-50 did not show a significant sensitivity to these values.

D. Sparsity of the DNF

In the examples presented in Table 1, the number of conjunctions in the DNF is always smaller than the number of possible conditions in the hidden layer of the rule learning module. As described in the Appendix C, the loss function of R2N contains sparsity regulation on the weights of the literal learning, AND and OR-layer, controlling the sparsity of the resultant DNF. To assess the role of the sparsity promoting parameter, λ, we trained the R2N for values of λ in the range $[10^{-5}, 1]$, both for Ex. 4 and 5 from Table 1. As shown in Fig. 3 (red), the number of conjunctions in the DNF decreases upon increasing λ. The accuracy however (black), remains near optimal for values inferior to $\lambda < 10^{-1}$, suggesting that the most compact DNF that has the smallest compromise on accuracy can be obtained for values of $\lambda \approx 10^{-2}$ or 10^{-3}. At larger values of λ, the strength of the sparsity constraint will result in DNFs that are shorter than the ground-truth, resulting in approximations of the hyperplanes that are sparser, and compromise on the prediction accuracy, e.g. for Ex. 4, 77% accuracy is obtained with a DNF containing a single conjunction $(-x_0 - 0.7x_1 - 0.4x_3 + 0.3x_4 > -0.7)$, while the ground-truth DNF contains three conjunctions. Controlling the value of λ provides the user an opportunity to balance the prediction accuracy with the sparsity of the DNF.

E. DNFs obtained with R2N

In this section we show the DNFs obtained with R2N and discuss how they are obtained. The DNF associated to the learned rule model can be extracted directly from the binary weight matrix $\mathbf{W}$ and $\mathbf{w}^{(d)}$, while the values of the learned literals are extracted from the weight matrix $\mathbf{W}^{(p)}$. Since the values of $\mathbf{W}^{(p)}$ are constrained with a sparsity penalty (see Appendix C), most values are small, yet not strictly zero. We therefore threshold all the values that have a normalized coefficient that is smaller than 2.5% of the magnitude of the largest coefficient in a predicate.

For instance, for example 5 in Table 1, the unprocessed output of R2N is given by,

$$
\begin{aligned}
&[(-0.7x_0 - 0.7x_1 - x_2 - 0.2x_3 - 0.8x_4 > -1173) \wedge (x_4 < 0.2) \wedge (x_0 - 0.6x_1 > -0.1)] \\
&\vee[(x_4 < 0.2) \wedge (-x_0 - 0.4x_1 - 0.1x_2 - 0.2x_4 > 704)] \\
&\vee[x_0 < 0.2] \\
&\vee[(0.5x_0 - 0.2x_1 - x_2 - 0.4x_3 + 0.1x_4 > -540) \wedge (-x_0 - 0.4x_1 - 0.1x_2 - 0.2x_4 > 704)] \\
&\vee[(x_4 < 0.2) \wedge (0.3x_0 + 0.1x_2 - x_3 - 0.3x_4 > 3050)] \\
&\vee[0.4x_1 + x_3 > 1.0]
\end{aligned} \tag{13}
$$

This can be simplified by identifying and removing all the constant false ($\perp$) or true ($\top$) predicates (always true or false), e.g., $0.5x_0 - 0.2x_1 - x2 - 0.4x_3 + 0.1x_4 > -540 \rightarrow \top$, into,

$$
\begin{aligned}
&[(x_4 < 0.2) \wedge (x_0 - 0.6x_1 > -0.1)] \\
&\vee[x_0 < 0.2] \\
&\vee[0.4x_1 + x_3 > 1.0].
\end{aligned} \tag{14}
$$

The simplified DNFs we obtain for the other examples in Table 1 are:
For ex. 1:

$$
\begin{aligned}
&[(x_0 > 0.26)] \\
&\vee[(x_1 < 0.49)].
\end{aligned} \tag{15}
$$

For ex. 2:

$$
\begin{aligned}
&[(x_0 - 0.41x_1 > 0) \wedge (x_0 - 0.53x_1 > 0)] \\
&\vee[(x_4 < 0.25)].
\end{aligned} \tag{16}
$$

For ex. 3:

$$
\begin{aligned}
&[(x_0 + 0.49x_1 > 0.41) \wedge (x_4 < 0.23)] \\
&\vee[(x_0 + 0.49x_1 > 0.41) \wedge (x_0 + 0.50x_1 > 0.51)] \\
&\vee[(x_4 < 0.25)].
\end{aligned} \tag{17}
$$

Finally for ex. 4:

$$
\begin{aligned}
&(x1 < 0.49) \wedge (-x_3 + x_4 >= 0) \\
&\vee(x_0 < -0.20) \\
&\vee[(x_0 - 0.1x_1 - 0.2x_2 - 0.4x_3 - 0.2x_4 > -2.2) \wedge (x_0 < 0.2) \wedge (-x_3 + x_4 > 0)] \\
&\vee[(x_0 - 0.1x_1 - 0.2x_2 - 0.4x_3 - 0.2x_4 > -2.1) \wedge (x_4 - x_3 > 0) \wedge (0.1x_0 + x_1 + x_2 - 0.1x_3 + > 0.46)]
\end{aligned} \tag{18}
$$

F. Baselines

This section describes the configuration settings for the three baselines: DR-Net [15], RIPPER [6], and BRCG [5] used in our experiments. For all cases, we used a common training set of
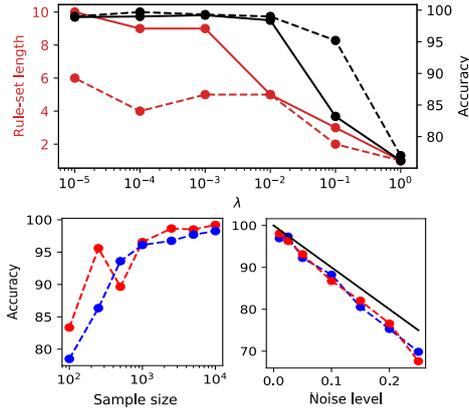
Figure 3: (top) Accuracy on the test-set and number of conjunctions in the DNF, n_r, as function of $\lambda = \lambda_{and} = \lambda_{or} = \lambda_p$ for Ex. 4 (dashed) and Ex. 5 (solid) from Table 1. Accuracy on the test set as function of (left) the size of the training set and (right) the noise level for (red) Ex. 4 and (blue) Ex. 5 from Table 1.

size 8000 and evaluated on a common test set of size 2000, and used the equi-quantile feature binarizer included in the BRCG repository[3] with 40 bins.

For BRCG, we use the publicly available implementation from the AI Explainability 360 repository[4]. The algorithm uses two parameters λ_1 and λ_2 to penalize the learning of more and longer clauses respectively. We set $\lambda_1 = \lambda_2 = 0.001$, which is the default value recommended in the implementation. We also use the default values for the maximum number of iteration (100), maximum number of columns generated per iteration (10), and maximum degree (10) and width (5) parameters for the beam search.

For DR-Net we use the implementation by the authors[5] with the default parameters except for the number of training epochs, which was increased from 2000 to 10000 according to the authors' recommendation to learn sparse rules. For RIPPER, we used the wittgenstein implementation[6]. But instead of using its internal binarizer, we binarized it upfront using the aforementioned feature binarizer.

For evaluating the predictive accuracy for the synthetic examples in Table 1 in the main text we also report the values obtained from a MLP (Multi Layer Perceptron). We used a simple two layer architecture, with 10 neurons in the hidden layer, trained for 10^4 epochs with default optimizer parameters (Adam optimizer).

G. Robustness w.r.t. noise and sample size

Data from which rule models are learned are oftentimes noisy and sparse. To assess the robustness of our approach w.r.t. number of training examples, we plot the prediction accuracy

---

[3]https://github.com/Trusted-AI/AIX360/blob/master/aix360/algorithms/rbm/features.py
[4]https://github.com/Trusted-AI/AIX360
[5]https://github.com/Joeyonng/decision-rules-network
[6]https://github.com/imoscovitz/wittgenstein/tree/master/wittgenstein

Example.	1	2	3	4	5
n_r with quantile binning	5	22	15	23	15
n_r with learned literals	3	3	2	5	3

Table 2
Number of conjunctions obtained with RIPPER by using quantile binning and the *learned* literals obtained from R2N for the 5 synthetic examples presented in Table 1

on the test set for Ex. 4 and 5 of Table 1 as function of the size of the training set (Fig. 3) and find that, even for a training set size of $O(10^3)$ we obtain a good accuracy ($> 0.95\%$). This shows that our method is not overly sensitive to the size of the training dataset, a common problem with neural-network based models.

Alternatively, when we add white noise to the data provided to the algorithm (randomly flipping a fraction of the labels in the dataset) we find that for both examples the accuracy of the test-set approaches near-optimal accuracy up to noise levels of 10% (Fig. 3: The solid line indicates optimal performance, note that since the data is noisy, this line is the upper limit for the performance).

H. Interoperability with other rule learning algorithms

The relational literals that are learned by R2N can also serve in other applications as part of the domain vocabulary (or ontology); for instance as input literals for other rule learning algorithms, ensuring interoperability with existing rule learning workflows. To showcase how this can augment algorithms like RIPPER, we used the *learned* literals from R2N as input literals for RIPPER and found that for all synthetic examples in Table 1, the number of rules learned is considerably less as compared to RIPPER with univariate value comparisons. As we display in Table 2, the learned number of rules decreases and becomes comparable with that of R2N, along with an improvement in accuracy for 4 of the 5 cases (not shown in the table), demonstrating that the learned literals are not only specific to R2N but can also benefit other rule learning algorithms.