7 Ethics Statement

This work presents RNS, a rule-based neural network for interpretable classification. We acknowledge and adhere to the ICLR Code of Ethics in all aspects of this research. Our work focuses on developing interpretable machine learning methods that can provide transparent decision-making processes, which aligns with ethical AI principles of explainability and accountability.

Data and Experimental Ethics: All datasets used in our experiments are publicly available benchmark datasets commonly used in machine learning research. We did not collect new data involving human subjects, and our use of existing datasets follows standard academic practices with proper attribution. The datasets include UCI Machine Learning Repository datasets and other established benchmarks that have been widely used in prior interpretable machine learning research.

Potential Applications and Societal Impact: While our method improves interpretability in machine learning, we acknowledge that rule-based models, like all AI systems, can potentially encode or amplify biases present in training data. We encourage practitioners to carefully evaluate fairness and bias when applying RNS to sensitive domains such as healthcare, finance, or criminal justice. The interpretable nature of our model can actually aid in identifying and addressing such biases by making decision logic transparent and auditable.

Research Integrity: We have conducted all experiments with scientific rigor, reported results honestly, and made efforts to ensure reproducibility by providing implementation details and planning to release code. We have appropriately cited prior work and clearly distinguished our contributions from existing methods. Our experimental comparisons use established baselines and evaluation metrics to ensure fair assessment.

8 Use of Large Language Models

We used large language models (LLMs) solely for text polishing and improving the clarity of our writing. All technical contributions, experimental design, implementation, analysis, and scientific insights are entirely our own work. The LLMs were not used for generating ideas, conducting experiments, writing code, or producing any substantive content. Specifically, LLMs assisted only with grammar corrections, sentence structure improvements, and enhancing the readability of our manuscript. The mathematical formulations, algorithmic innovations, experimental methodology, and all results interpretation remain exclusively the product of the authors' research efforts.

9 Reproducibility Statement

We have made extensive efforts to ensure the reproducibility of our results. **Implementation Details:** Section "Reproducibility" in the Appendix provides comprehensive hyperparameter settings, training procedures, and architectural specifications for RNS. We specify the number of Logic Selection Layer neurons (grid-searched from 32 to 4096), regularization coefficients (10^{-5} to 10^{-9}), binning strategies, batch sizes, learning rates, and training schedules for both small and large datasets.

Code and Data Availability: All datasets used are publicly available from the UCI Machine Learning Repository and other standard sources, with complete statistics provided in Table 2 of the Appendix. We will release our PyTorch implementation of RNS along with experimental scripts upon paper acceptance. The anonymous code repository is currently available at the URL provided in the paper.

Experimental Reproducibility: Our experiments use 5-fold cross-validation with multiple random seeds to ensure statistical reliability. We provide detailed baseline configurations following prior work citations, use established evaluation metrics (F1 scores, rule quality measures), and specify all preprocessing steps, including feature binarization methods. The simulation experiments use controlled synthetic data generation procedures that are fully specified to enable exact replication.

Baseline Comparisons: We use implementations and settings from established prior work, particularly following the high-quality codebase from Wang et al. (2024) for fair comparison with existing rule-based neural networks. All hardware specifications (NVIDIA A100 80GB GPU, Linux server) and software dependencies (PyTorch) are documented to facilitate reproduction of computational results.

A Related Work

A.1 Traditional Rule Learning Methods

Historically, example-based rule learning algorithms (S., 1969; Michalski, 1973; Pagallo & Haussler, 1990) were initially proposed by selecting a random example and finding the best rule to cover it. However, due to their computational inefficiency, CN2 (Clark & Niblett, 1989) explicitly changed the strategy to finding the best rule that covers as many examples as possible. Building on these heuristic algorithms, FOIL (Quinlan, 1990), a system for learning first-order Horn clauses, was subsequently developed. While some algorithms learn rule sets directly, such as RIPPER (Cohen, 1995), PART (Frank & Witten, 1998), and CPAR (Yin & Han, 2003), others post-process a decision tree (Quinlan, 2014) or construct sets of rules by post-processing association rules, like CBA (Liu et al., 1998) and CMAR (Li et al., 2001). All these algorithms use different strategies to find and use sets of rules for classification.

Most of these algorithms are based on Disjunctive Normal Form (DNF) (S., 1969; Michalski, 1973; Pagallo & Haussler, 1990; Clark & Niblett, 1989; Liu et al., 1998; Li et al., 2001; Frank & Witten, 1998; Yin & Han, 2003; Cohen, 1995) expressions. CNF learners have been shown to perform competitively with DNF learners (Mooney, 1995), inspiring a line of CNF learning algorithms (Dries et al., 2009; Jain et al., 2021; Beck et al., 2023; Sverdlik, 1992; Hong & Tsang, 1997). Traditional rule-based models are valued for their interpretability but struggle to find the global optimum due to their discrete, non-differentiable nature. Extensive exploration of heuristic methods (Quinlan, 2014; Loh, 2011; Cohen, 1995) has not consistently yielded optimal solutions.

In response, recent research has turned to Bayesian frameworks to enhance model structure (Letham et al., 2015; Wang et al., 2017; Yang et al., 2017), employing strategies such as if-then rules (Lakkaraju et al., 2016) and advanced data structures for quicker training (Angelino et al., 2018). Despite these advancements, extended search times, scalability challenges, and performance issues limit the practicality of rule-based models compared to ensemble methods like Random Forest (Breiman, 2001) and Gradient Boosted Decision Trees (Chen & Guestrin, 2016; Ke et al., 2017), which trade off interpretability for improved performance.

A.2 Rule Learning Neural Networks

Neural rule learning-based methods integrate rule learning with advanced optimization techniques, enabling the discovery of complex and nuanced rules that combine the interpretability of symbolic models with the generalization power of neural networks. Unlike tree-based models, which explicitly follow feature-condition rules, neural approaches rely on weight parameters to control the rule learning process, offering improved robustness and scalability through data-driven training. However, existing approaches such as neural decision trees and rule extraction from neural networks (Frosst & Hinton, 2017; Ribeiro et al., 2016; Wang et al., 2020; 2021; Zhang et al., 2023) face challenges in fidelity and scalability. In particular, RRL (Wang et al., 2021; 2024), a state-of-the-art rule-based neural network, requires a predefined structure of CNF and DNF layers, which limits flexibility, leads to inefficient rule discovery, and exacerbates optimization issues such as gradient vanishing (Wang et al., 2020). Our proposed Rule Network with Selective Logical Operators (RNS) addresses these challenges through its novel architecture and optimization strategies, improving scalability, rule quality, and training stability, as detailed in Section 3.

A.3 Binarized Neural Network

A related topic to this work is Binarized Neural Networks (BNNs), which optimize deep neural networks by employing binary weights. The deployment of deep neural networks typically requires substantial memory storage and computing resources. To achieve significant memory savings and energy efficiency during inference, recent efforts have focused on learning binary model weights while maintaining the performance levels of their floating-point counterparts Courbariaux et al. (2015; 2016a); Rastegari et al. (2016); Bulat & Tzimiropoulos (2019); Liu et al. (2018). Innovations such as bit logical operations Kim & Smaragdis (2016) and novel training strategies for self-binarizing networks Lahoud et al. (2019), along with integrating scaling factors for weights and activations Sakr et al. (2018), have advanced BNNs. However, due to the binary nature of their weights, BNNs face optimization challenges. The Straight-Through Estimator (STE) method Courbariaux et al. (2015;

Dataset	#instances	#classes	#features	feature type
adult	32561	2	14	mixed
bank	45211	2	16	mixed
chess	28056	18	6	discrete
connect-4	67557	3	42	discrete
letRecog	20000	26	16	continuous
magic04	19020	2	10	continuous
wine	178	3	13	continuous
activity	10299	6	561	continuous
dota2	102944	2	116	discrete
facebook	22470	4	4714	discrete
fashion	70000	10	784	continuous

Table 3: Datasets statistics.

2016a); Cheng et al. (2019) allows gradients to "pass through" non-differentiable functions, making it particularly effective for discrete optimization.

Despite both using binarized model weights and employing STE for optimization, our work diverges significantly from BNNs. First, RNS adopts specialized logical activation functions to perform logical operations on features, whereas BNNs typically use the Sign function to produce binary outputs. Second, BNNs are fully connected neural networks, while RNS features a learning mechanism for its connections. Most importantly, these distinctions enable RNS to learn logical rules for both prediction and interpretability, setting it apart from BNNs, which are primarily designed to enhance model efficiency.

B Dataset Statistics

In Table 3, the first nine data sets are small, while the last four are large. The "Discrete" or "Continuous" feature type indicates that all features in the data set are either discrete or continuous, respectively. The "Mixed" feature type indicates that the corresponding data set contains both discrete and continuous features.

C Reproducibility

Reproducibility. RNS includes two Logic Selection Layers (LSLs), with the number of logical neurons grid-searched from 32 to 4096 based on dataset complexity. For training, we use cross-entropy loss with L2 regularization to control model complexity, with the regularization coefficient searched in the range 10^{-5} to 10^{-9}. The number of bins in the feature binarization layer is selected from $\{15, 30, 50\}$. The model is trained using the Adam optimizer (Kingma & Ba, 2014) with a batch size of 32. For small datasets, training runs for 400 epochs, with the learning rate reduced by 10% every 100 epochs. For large datasets, training is conducted for 100 epochs with a similar learning rate schedule, decreasing every 20 epochs. Baseline settings follow those described in (Wang et al., 2021; 2024). RNS is implemented in PyTorch (Paszke et al., 2019), and we use the high-quality code base from (Wang et al., 2021; 2024) for baseline comparisons. Experiments are performed on a Linux server equipped with an NVIDIA A100 80GB GPU. All code and data are available at `https://anonymous.4open.science/r/RNS_-4A67/`.

D Hyperparameter Study

LSL Dimension K. We analyze the effect of layer dimension K (number of neurons) in the two LSLs. A larger K increases model complexity, potentially improving performance but risking overfitting. We test dimensions K in $\{32@32, 64@64, 128@128, 256@256, 512@512, 1024@1024, 2048@2048\}$ on a small dataset (bank) and a large dataset (activity) shown in Figure 5. The F1 score initially rises and then falls as K increases, peaking at $1024@1024$ for the activity dataset and $128@128$ for the bank dataset. This indicates that the activity dataset benefits from larger models, while the bank

dataset performs best with smaller models, suggesting that optimal model size depends on dataset complexity.

L2 Regularization λ. We examine the impact of the L2 regularization weight λ, which controls RNS complexity. A larger λ reduces model complexity, resulting in fewer and simpler logical rules. We vary λ from 10^{-5} to 10^{-9} and show the model's performance in Figure 5 . The F1 score initially rises and then falls as λ increases, indicating that performance improves with an appropriately chosen λ. Balancing λ is essential to avoid overfitting or underfitting and achieve optimal model complexity.

E EFFICIENCY

We evaluate learning efficiency by the number and length of learned rules across all datasets, as shown in Figure 7 and Figure 8, respectively. Figure 7 presents scatter plots of F1 score against log(#edges) across 10 datasets. The boundary connecting the results of different RNS architectures separates the upper left corner from the best baseline methods. This indicates that RNS consistently learns fewer rules while achieving high prediction accuracy across various scenarios. The average length of rules in RNS trained on different datasets is shown in Figure 8. The average rule length is less than 6 for all datasets except fashion and facebook, which are unstructured datasets and have more complex features. These results indicate that the rules learned by RNS are generally easy to understand across different scenarios.

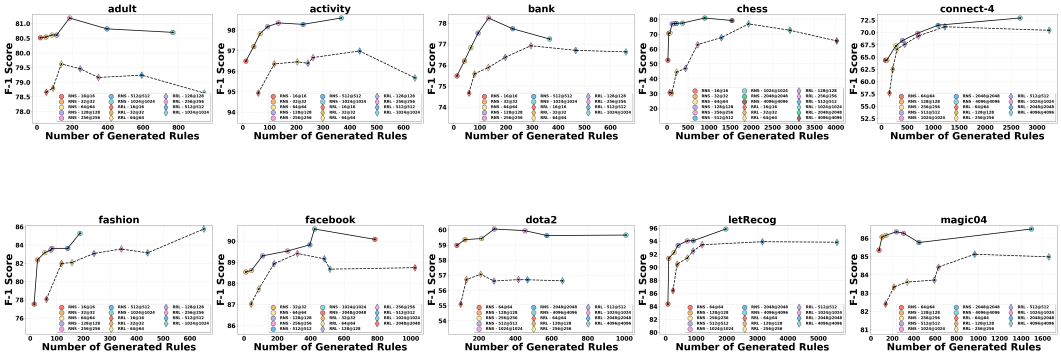

Figure 7: Rule Number comparison across different architectures.

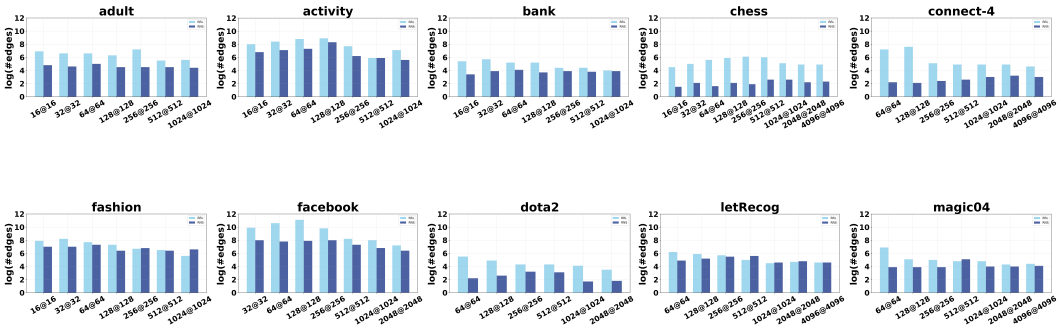

Figure 8: Rule Length comparison across different architectures.

F RULE QUALITY

Rule Quality Metrics. Following prior work Yu et al. (2023); Lakkaraju et al. (2016), we evaluate rule quality using three key metrics: diversity, coverage, and single-rule accuracy. Let I_r denote the

set of data instances covered by rule r, and D denote the complete dataset. The metrics are defined as:

Single-rule Accuracy measures the prediction accuracy of a single rule for the instances it covers:

$$\text{Single-rule Accuracy} = \frac{|\{i \in I_r : \text{rule prediction matches } y_i\}|}{|I_r|} \tag{6}$$

This metric evaluates how accurately a single rule classifies the data instances it covers when used independently for prediction. It measures the proportion of correctly classified instances among all instances that satisfy the rule's conditions. Higher single-rule accuracy indicates that the rule is more reliable for making predictions on its own, without relying on other rules in the rule set. **Coverage** quantifies the proportion of data instances covered by a rule:

$$\text{Coverage} = \frac{|I_r|}{|D|} \tag{7}$$

This metric measures the scope or generality of a rule. Lower coverage indicates that rules are more specific and easier for human experts to understand, as they apply to a smaller, more focused subset of the data. Very high coverage may suggest overly general rules that lack specificity.

Diversity measures the overlap ratio between pairs of rules, with higher diversity reflecting that rules capture distinct, non-redundant logic:

$$\text{Diversity} = 1 - \frac{|I_i \cap I_j|}{|I_i \cup I_j|} \tag{8}$$

This metric quantifies how different rules are from each other by measuring their overlap. Higher diversity values indicate that rules capture different patterns in the data with minimal redundancy, leading to a more comprehensive and non-redundant rule set. Low diversity suggests that multiple rules are covering similar data instances, which reduces the interpretability and efficiency of the rule set.

Results. We conduct extensive experiments across 10 datasets, as shown in Figure 9, Figure 10, and Figure 11. RNS consistently produces rules with superior results: higher accuracy, greater diversity, and lower coverage deviation. This indicates that the rules learned by RNS are easier to distinguish and exhibit better prediction and generalization power.

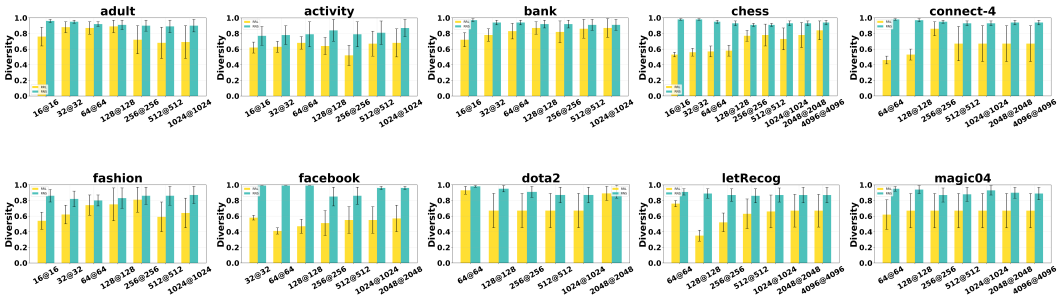

Figure 9: Rule Diversity comparison across different architectures.

G Simulation Experiment

We conduct a controlled simulation experiment to assess RNS's ability to learn the exact logical rules used to generate synthetic data.

Setup. We generate synthetic data based on predefined logical rules and train rule-based models to evaluate their ability to reconstruct these ground-truth rules. The dataset is synthesized by defining three probability parameters, $\mathbf{p} = (p_1, p_2, p_3)$, corresponding to feature variables $\{x_1, x_2, x_3\}$, each drawn independently from a uniform distribution $U(0, 1)$. Using these as Bernoulli parameters, we generate $X_{gen} \in \mathbb{R}^{3\times 50000}$ by repeating Bernoulli sampling, resulting in $n = 50000$ binary vectors of length 3, where each element is sampled from $Bernoulli(p_i)$. Labels are assigned to these binary

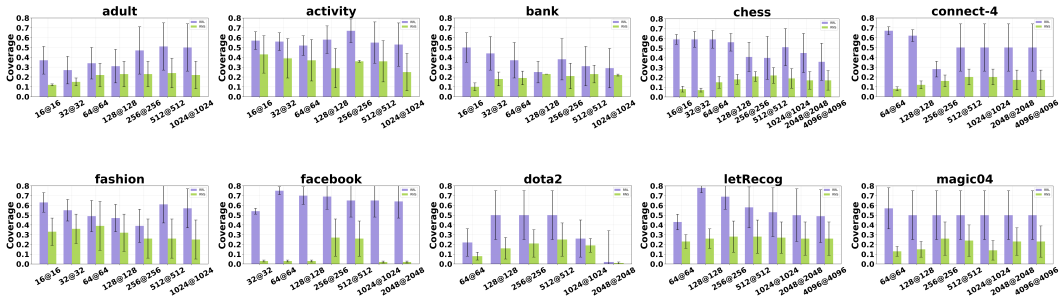

Figure 10: Rule Coverage comparison across different architectures.

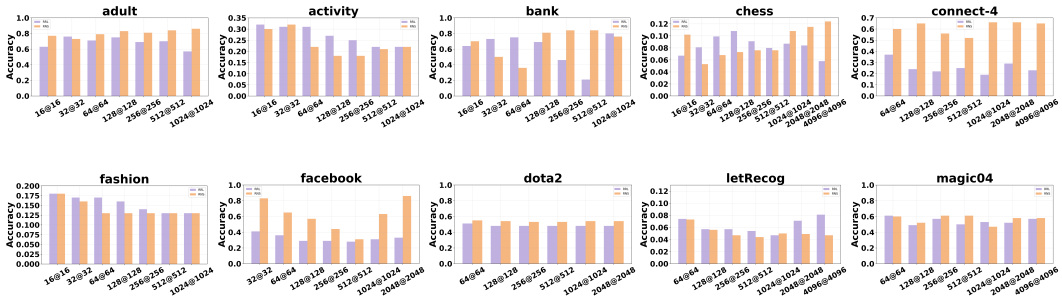

Figure 11: Rule Accuracy comparison across different architectures.

vectors based on specific logical rules. For example, a rule $x_1 \wedge x_2 \wedge x_3 \rightarrow 1$ assigns a label of 1 if all three features are 1. We define four different types of rules, presented as "Ground-Truth Rules" in Table 2. The dataset is split evenly into training and test sets. Both RNS and RRL are configured with logical layer dimensions of 64@64.

Results. As shown in Table 2, RNS not only achieves near 100% accuracy but also recovers the exact structure of the ground-truth rules. In contrast, RRL, even with a naive negation setting, tends to produce overly simplified, redundant, or logically incomplete rules.

For example, RNS is able to precisely recover rules such as $x_1 \wedge \neg x_2 \wedge x_3$ and $x_2 \wedge \neg x_3$, faithfully mirroring the underlying logic used to generate the data. RRL, on the other hand, often produces rules that are either too general (e.g., x_1, x_2) or combine terms in a way that does not fully reflect the intended logic (e.g., $(x_1 \wedge x_3), x_2$). This pattern holds even when RRL is augmented with the naive negation setting: RNS consistently recovers all ground-truth rules, whereas RRL fails to identify several rules in their correct logical form and often outputs multiple trivial or repeated variants for a single ground-truth pattern.

H Ablation Study (RQ4)

Negation Layer and NFC. To evaluate the effectiveness of the Negation Layer and Normal Form Constraint (NFC), we compare RNS trained with and without these components. The Negation Layer enables functional completeness by supporting negation operations in learned rules, while NFC narrows the search space for learning connections between logical layers, improving performance. Table 4 shows F1 scores for RNS and its variants on the bank (small) and activity (large) datasets. The F1 score decreases without these components, highlighting their importance. NFC also improves learning efficiency. With all other factors constant, training time decreases by 11.22% and 25% for the activity and bank datasets, respectively, as shown in Table 5, emphasizing its utility.

Binning Function. We evaluate RNS's performance with different binning functions introduced in Section 2.2. F1 scores for RNS with these methods are shown in Table 6. RNS exhibits strong flexibility, with RanInt performing the best due to its stochastic diversity, which helps prevent overfitting. Additionally, its computational efficiency and simplicity are advantageous. KInt, in contrast, clusters based solely on feature values, neglecting target variables, which can reduce its

Model Variant	Activity	Bank
RNS w/o Negation Layer	97.68	76.92
RNS w/o NFC	97.81	76.36
RNS (Full)	**98.37**	**77.18**

Table 4: Ablation study: F1 Score (%) of RNS and its variants on two datasets.

Configuration	Activity	Bank
RNS w/o NFC	1h 38m 9s	1h 24m 50s
RNS (with NFC)	**1h 27m 3s**	**1h 3m 56s**

Table 5: Training efficiency: Impact of Normal Form Constraint (NFC) on training time.

Binning Method	Activity	Bank
KInt (K-means)	92.71	74.37
EntInt (Entropy)	97.09	77.02
AutoInt (Auto-interval)	95.62	75.59
RanInt (Random)	**98.80**	**77.18**

Table 6: Feature preprocessing: F1 Score (%) comparison of different binning methods.

effectiveness. EntInt incorporates label information to minimize entropy within bins, potentially improving accuracy. AutoInt, while adaptable, incurs significant computational overhead due to its parameter optimization, posing challenges in practice.

I Model Interpretation

Figure 6 presents the three most discriminative rules extracted from RNS, trained on the bank-marketing dataset to identify customer profiles likely to subscribe to a term deposit through telesales. These rules provide detailed profiles by highlighting specific conditions that increase subscription likelihood. 'Coverage' denotes the proportion of training samples satisfying a rule.

The transparent interpretations reveal several insights. Subscriptions are less likely during colder seasons, possibly due to reluctance to make financial decisions during holidays. The model identifies a correlation between longer call durations and successful marketing, suggesting that extended dialogues often indicate interest in deposits. It also highlights that housemaids, entrepreneurs, and self-employed individuals are less inclined towards term deposits, potentially due to financial constraints or the need for liquidity. Additionally, the analysis shows age-related trends in call duration, with younger individuals having shorter calls and middle-aged clients engaging in longer discussions. These findings demonstrate the model's ability to reflect plausible real-world behaviors while avoiding direct causal assertions.

J Gradient vanishing

Despite the high interpretability of discrete logical layers, training them is challenging due to their discrete parameters and non-differentiable structures. This challenge is addressed by drawing inspiration from the training methods used in binary neural networks, which involve searching for discrete solutions within a continuous space. Wang et al. (2021) leverages the logical activation functions proposed by Payani & Fekri (2019) in RRL:

$$AND(h, W^i_{conn}) = \prod_{j=1}^{n} F_c(h_j, W_{i,j}) \tag{9}$$

$$OR(h, W^i_{conn}) = 1 - \prod_{j=1}^{n}(1 - F_d(h_j, W_{i,j})) \tag{10}$$

where $F_c(h, w) = 1 - w(1 - h)$ and $F_d(h, w) = h \cdot w$.

If $\mathbf{h}$ and W_i are both binary vectors, then $\text{Conj}(\mathbf{h}, W_i) = \wedge_{W_{i,j}=1} h_j$ and $\text{Disj}(\mathbf{h}, W_i) = \vee_{W_{i,j}=1} h_j$. $F_c(\mathbf{h}, W)$ and $F_d(\mathbf{h}, W)$ decide how much h_j would affect the operation according to $W_{i,j}$. After using continuous weights and logical activation functions, the AND and OR operators, denoted by R and S, are defined as follows:

$$r_i^{(l)} = AND(u^{(l-1)}, W_i^{(l,0)}) \tag{11}$$

$$s_i^{(l)} = OR(u^{(l-1)}, W_i^{(l,1)}) \tag{12}$$

Although the whole logical layer becomes differentiable by applying this continuous relaxation, the above functions are not compatible with RNS. In this setting, the output of the node $h \in [0, 1]$ conflicts with our binarized setting, where all logical parameters are either -1 or $+1$. This not only breaks the inherently discrete nature of RNS but also suffers from the serious vanishing gradient problem. The reason can be found by analyzing the partial derivative of each node with respect to its directly connected weights and with respect to its directly connected nodes as follows:

$$\frac{\partial r}{\partial W_{i,j}} = (u_j^{(l-1)} - 1) \cdot \prod_{k \neq j} F_c(u_k^{(l-1)}, W_{i,k}^{(l,0)}) \tag{13}$$

$$\frac{\partial r}{\partial u_j^{(l-1)}} = W_{i,j}^{(l,0)} \cdot \prod_{k \neq j} F_c(u_k^{(l-1)}, W_{i,k}^{(l,0)}) \tag{14}$$

Since $u_j^{(l-1)}$ and $W_{i,k}^{(l,0)}$ fall within the range of 0 to 1, the values of $F_c(\cdot)$ also lie within this range. If the number of inputs is large and most $F_c(\cdot)$ are not 1, the gradient tends toward 0 due to multiplications. Additionally, in the discrete setting, only when $u_j^{(l-1)}$ and $1 - F_d(\cdot)$ are all 1, can the gradient be non-zero, which is quite rare in practice.

J.1 Detailed Analysis

Training discrete logical layers in neural networks is challenging due to their binary parameters and non-differentiable operations. A common approach to make such models trainable is to relax the logical functions to a continuous, differentiable form Wang et al. (2024). These relaxations enable gradient-based training, but they often suffer from severe **vanishing gradient** problems due to the multiplicative structure of the logical functions. Moreover, the RRL activations output values in $[0, 1]$, which conflicts with a fully binarized ± 1 logic setting (such as our RNS approach) and breaks the discrete semantics of the network. We detail below why vanishing gradients occur in such formulations and describe the solutions proposed in RRL as well as the different strategy taken by RNS to overcome these issues.

J.2 Vanishing Gradients in Multiplicative Logical Activations (RRL)

In a conventional formulation of a logical AND gate Payani & Fekri (2019), the output is the product of its binary inputs. For binary $x_i \in \{0, 1\}$, one can write

$$\text{AND}(x_1, \ldots, x_n) = \prod_{i=1}^{n} x_i, \tag{15}$$

and similarly, a logical OR can be written (using De Morgan's law) as

$$\text{OR}(x_1, \ldots, x_n) = 1 - \prod_{i=1}^{n}(1 - x_i). \tag{16}$$

While these expressions are correct for binary values, their direct use in a neural network leads to zero gradients almost everywhere – a phenomenon known as *gradient vanishing*. The gradient of the AND with respect to one input is

$$\frac{\partial \text{AND}(x)}{\partial x_j} = \prod_{i \neq j} x_i, \tag{17}$$

which is zero whenever any other input x_i is zero. More generally, if x_i are in $[0, 1]$, this derivative equals the product of the other $(n - 1)$ inputs. As n grows or if the inputs are often fractional values < 1, this product becomes extremely small – decaying exponentially with n – thus severely diminishing the gradient signal. In the case of OR (product of $(1 - x_i)$ terms), the same issue arises by symmetry (the gradient will contain a product of many factors in $[0, 1]$). The consequence is that standard product-based logic units have large regions of the input space where the gradient is essentially zero, hampering learning.

In a fully discrete setting, if a conjunction has more than one false literal, changing any single input from 0 to 1 does not alter the output (since at least one false remains to make the AND false). Thus, the gradient at that point is exactly zero in all those input directions – creating broad "dead zones" where the network cannot learn.

To enable gradient-based training, RRL replaces the hard logical operations with differentiable approximations that produce continuous outputs. Let $h = (h_1, \ldots, h_n)$ be the input vector to a logical neuron (with $h_j \in [0, 1]$ as relaxations of Boolean values) and let $W_i = (W_{i,1}, \ldots, W_{i,n})$ be a set of weights indicating which inputs are involved in the i-th clause (for example, $W_{i,j} \in \{0, 1\}$ or $[0, 1]$, with 1 meaning the jth input is included in the clause). RRL defines smoothed conjunction and disjunction functions as:

$$\text{Conj}(h, W_i) = \prod_{j=1}^{n} F_c(h_j,\, W_{i,j}), \qquad F_c(h, w) = 1 - w(1 - h), \tag{18}$$

$$\text{Disj}(h, W_i) = 1 - \prod_{j=1}^{n} \Big(1 - F_d(h_j,\, W_{i,j})\Big), \qquad F_d(h, w) = h \cdot w, \tag{19}$$

where $F_c(h, w)$ and $F_d(h, w)$ are specific smooth blending functions for inputs h and weights w. If $h \in \{0, 1\}^n$ and W_i is binary, $\text{Conj}(h, W_i)$ reduces to the logical AND of the selected inputs (those with $W_{i,j} = 1$), and $\text{Disj}(h, W_i)$ becomes the logical OR. For continuous h and W_i, these give differentiable outputs in $[0, 1]$.

However, the gradient remains problematic. Consider $r_i = \text{Conj}(h, W_i)$. Using the chain rule:

$$\frac{\partial r_i}{\partial W_{i,j}} = \Big(\prod_{k \neq j} F_c(h_k, W_{i,k})\Big) \frac{\partial F_c(h_j, W_{i,j})}{\partial W_{i,j}} = (h_j - 1) \prod_{k \neq j} F_c(h_k, W_{i,k}) \,, \tag{20}$$

$$\frac{\partial r_i}{\partial h_j} = \Big(\prod_{k \neq j} F_c(h_k, W_{i,k})\Big) \frac{\partial F_c(h_j, W_{i,j})}{\partial h_j} = W_{i,j} \prod_{k \neq j} F_c(h_k, W_{i,k}) \,. \tag{21}$$

Each partial derivative is proportional to a product of $(n - 1)$ terms $F_c(h_k, W_{i,k})$, each of which lies in $[0, 1]$. Unless all those terms are very close to 1, the product will be small; if any term is 0, the product (and thus the gradient) is zero. In other words, the magnitude of the gradient *shrinks multiplicatively* with every additional input in the clause.

A similar calculation for $s_i = \text{Disj}(h, W_i)$ yields, by symmetry (with $G_k := 1 - F_d(h_k, W_{i,k})$):

$$\frac{\partial s_i}{\partial W_{i,j}} = h_j \prod_{k \neq j} G_k \,, \qquad \frac{\partial s_i}{\partial h_j} = W_{i,j} \prod_{k \neq j} G_k \,, \tag{22}$$

which again contains a product of $(n - 1)$ factors $G_k \in [0, 1]$. Thus, for large n or for typical fractional values of h_j and $W_{i,j}$, these gradients **vanish** to near-zero. In practice, when many inputs

of an AND are not extremely close to 1, the gradient through that AND node becomes negligibly small. And in the discrete limit ($h, W \in \{0, 1\}$), $\frac{\partial r_i}{\partial W_{i,j}}$ and $\frac{\partial r_i}{\partial h_j}$ are zero in almost all cases: only if *all* other inputs of the AND are 1 (so that the output is sensitive to the remaining input/literal) will a change in that input make a difference. Such a situation—e.g., a clause where all but one literal is true—is rare, meaning the network spends most of its time in regimes where the loss gradient is zero with respect to each individual parameter. This underscores the fundamental incompatibility of naive multiplicative logical units with gradient-based learning: the more literals in a rule, the more pronounced the vanishing gradient problem becomes.

Incompatibility with ± 1 Binarization (RNS). An additional issue is that the RRL activations $\text{Conj}(h, W)$ and $\text{Disj}(h, W)$ produce outputs in the range $[0, 1]$ (since they are essentially probabilities or fractional truth values). This is incompatible with the design of RNS, where all internal logical signals are meant to be binary $\{-1, +1\}$ values at runtime. In RNS, we require a hard True or False, encoded as $+1$ or -1. Using RRL's continuous outputs inside an RNS network would break the discrete semantics and require additional mechanisms to re-binarize the outputs at each layer. Moreover, as discussed above, those continuous outputs suffer from vanishing gradients when used in deep clauses. Thus, while RRL's relaxation makes a logical layer differentiable, it does so at the cost of deviating from binary ± 1 representation and encountering extremely small gradients for large rules. These drawbacks motivate alternative approaches that maintain binary representations and avoid multiplicative shrinkage.

J.3 RRL's Mitigations: Log-Domain Smoothing and Gradient Grafting

RRL and related frameworks have proposed a couple of techniques to alleviate the vanishing gradient and training difficulties while still using product-based logic. We briefly summarize two such strategies: a log-domain scaled activation function, and a hybrid training method known as gradient grafting.

Log-space smoothing of logical activations. One idea introduced with RRL is to modify the product formulation by amplifying small values in the product through a log transformation. Specifically, define a projection function $P(v)$ that boosts a small product v:

$$P(v) \;=\; \frac{1}{1 - \log v}\,, \qquad \frac{dP}{dv} \;=\; \frac{P(v)^2}{v}\,. \tag{23}$$

Here $v > 0$ is a value (in practice, v will be a small positive number representing the product of several terms). The function $P(v)$ is chosen such that when v is small, $-\log v$ is large, so $P(v)$ will significantly exceed v (for example, if $v = 10^{-3}$, then $P(v) \approx 1/(1 - (-6.9)) \approx 1/7.9 \approx 0.127$, whereas v itself is 0.001). In this way, $P(v)$ stretches the lower end of the range upward.

RRL incorporates this into the logical units by first computing the product of inputs in log-space and then projecting back. Using a small $\varepsilon > 0$ to avoid $\log 0$, the *improved* conjunction and disjunction are defined as:

$$\text{Conj}^{+}(h, W_i) \;=\; P\!\left(\prod_{j=1}^{n} \big(F_c(h_j, W_{i,j}) + \varepsilon\big)\right), \tag{24}$$

$$\text{Disj}^{+}(h, W_i) \;=\; 1 \;-\; P\!\left(\prod_{j=1}^{n} \big(1 - F_d(h_j, W_{i,j}) + \varepsilon\big)\right). \tag{25}$$

When $\varepsilon \to 0$ and h, W are binary, Conj^{+} and Disj^{+} recover the exact AND/OR as before. However, for intermediate values, these use $P(\cdot)$ to prevent the product from becoming too small. If we let

$$v \;=\; \prod_{j=1}^{n} \big(F_c(h_j, W_{i,j}) + \varepsilon\big)$$

be the raw product inside P, then by the chain rule the gradient of Conj^+ with respect to a parameter becomes:

$$\frac{\partial \,\text{Conj}^+}{\partial W_{i,j}} = \frac{P(v)^2}{v} \left(\prod_{k \neq j} \left(F_c(h_k, W_{i,k}) + \varepsilon \right) \right) \frac{\partial F_c(h_j, W_{i,j})}{\partial W_{i,j}}, \tag{26}$$

$$\frac{\partial \,\text{Conj}^+}{\partial h_j} = \frac{P(v)^2}{v} \left(\prod_{k \neq j} \left(F_c(h_k, W_{i,k}) + \varepsilon \right) \right) \frac{\partial F_c(h_j, W_{i,j})}{\partial h_j}. \tag{27}$$

Comparing these to the original gradients equation 20–equation 21, we see that the pure product term $\prod_{k \neq j} F_c(h_k, W_{i,k})$ is now multiplied by $\frac{P(v)^2}{v}$. For moderately small v, this factor can be significantly larger than 1, thereby attenuating the vanishing effect. Intuitively, instead of the gradient shrinking proportional to v (the product of many small terms), it shrinks proportional to $P(v)^2$, and since $P(v) > v$ when v is small, the decay is slower. This log-space trick can appreciably increase gradient magnitudes when the clause length n is not too large or the inputs are not too extreme.

Comparing these to the original gradients equation 20–equation 21, we see that the pure product term $\prod_{k \neq j} F_c(h_k, W_{i,k})$ is now multiplied by $\frac{P(v)^2}{v}$. For moderately small v, this factor can be significantly larger than 1, thereby attenuating the vanishing effect.

To understand how the gradient shrinks, let's analyze this mathematically. Consider the product term:

$$v = \prod_{j=1}^{n} F_c(h_j, W_{i,j}) = \prod_{j=1}^{n} (1 - W_{i,j}(1 - h_j)) \tag{28}$$

For typical intermediate values where $h_j \approx 0.5$ and $W_{i,j} \approx 0.5$, we have:

$$F_c(h_j, W_{i,j}) \approx 1 - 0.5(1 - 0.5) = 0.75 \tag{29}$$

Therefore, the product becomes:

$$v \approx (0.75)^n \tag{30}$$

As n increases, this decays exponentially:

$$n = 10: \quad v \approx 0.75^{10} \approx 0.056 \tag{31}$$

$$n = 20: \quad v \approx 0.75^{20} \approx 0.003 \tag{32}$$

$$n = 50: \quad v \approx 0.75^{50} \approx 5.7 \times 10^{-7} \tag{33}$$

$$n = 100: \quad v \approx 0.75^{100} \approx 3.2 \times 10^{-13} \tag{34}$$

The gradient magnitude without log-smoothing is proportional to v:

$$\left| \frac{\partial r_i}{\partial W_{i,j}} \right| \propto v \approx \alpha^n \quad \text{where } \alpha < 1 \tag{35}$$

With the log-space projection $P(v)$, the gradient becomes:

$$\left| \frac{\partial \,\text{Conj}^+}{\partial W_{i,j}} \right| \propto \frac{P(v)^2}{v} = \frac{1}{v(1 - \log v)^2} \tag{36}$$

For small v, we have $P(v) \approx \frac{1}{-\log v}$, so:

$$\frac{P(v)^2}{v} \approx \frac{1}{v(\log v)^2} \tag{37}$$

Let's evaluate this for different clause sizes:

$$n = 10: \quad v \approx 0.056, \quad \frac{P(v)^2}{v} \approx \frac{1}{0.056 \times (-2.88)^2} \approx 2.15 \tag{38}$$

$$n = 20: \quad v \approx 0.003, \quad \frac{P(v)^2}{v} \approx \frac{1}{0.003 \times (-5.81)^2} \approx 9.87 \tag{39}$$

$$n = 50: \quad v \approx 5.7 \times 10^{-7}, \quad \frac{P(v)^2}{v} \approx \frac{1}{5.7 \times 10^{-7} \times (-14.4)^2} \approx 8.5 \times 10^3 \tag{40}$$

While $P(v)^2/v$ grows as v shrinks, for very small v (large n), the growth is only polynomial in $\log(1/v) \approx n \log(1/\alpha)$, not enough to fully compensate for the exponential decay. Eventually, for very large n:

$$\lim_{n\to\infty} \frac{P(v)^2}{v} \approx \frac{1}{\alpha^n \cdot n^2 \cdot (\log \alpha)^2} \to 0 \tag{41}$$

While $P(v)^2/v$ grows as v shrinks, for very small v (large n), the growth is only polynomial in n, not enough to fully compensate for the exponential decay.

To see this clearly, recall that $v \approx \alpha^n$ where $\alpha < 1$ (e.g., $\alpha = 0.75$). Then:

$$P(v) = \frac{1}{1 - \log v} = \frac{1}{1 - \log(\alpha^n)} = \frac{1}{1 - n \log \alpha} \tag{42}$$

Since $\alpha < 1$, we have $\log \alpha < 0$, so $-\log \alpha > 0$. For large n:

$$P(v) \approx \frac{1}{n|\log \alpha|} \tag{43}$$

Therefore:

$$\frac{P(v)^2}{v} \approx \frac{1/n^2(\log \alpha)^2}{\alpha^n} = \frac{1}{\alpha^n \cdot n^2 \cdot (\log \alpha)^2} \tag{44}$$

The key observation is:

- The numerator grows as $\mathcal{O}(1/n^2)$ (polynomial decay)
- The denominator decays as $\mathcal{O}(\alpha^n)$ (exponential decay)

Since exponential decay dominates polynomial growth:

$$\lim_{n\to\infty} \frac{1}{\alpha^n \cdot n^2 \cdot (\log \alpha)^2} = \lim_{n\to\infty} \frac{1}{n^2(\log \alpha)^2} \cdot \frac{1}{\alpha^n} \to 0 \tag{45}$$

because $\frac{1}{\alpha^n}$ approaches 0 much faster than $\frac{1}{n^2}$ approaches infinity.

Thus, while the log-space trick delays the vanishing, it cannot prevent it for large n. Intuitively, instead of the gradient shrinking proportional to v (the product of many small terms), it shrinks proportional to $P(v)^2$, and since $P(v) > v$ when v is small, the decay is slower. This log-space trick can appreciably increase gradient magnitudes when the clause length n is not too large or the inputs are not too extreme.

However, this modification does **not fully eliminate** the vanishing gradient problem. When n is very large or many inputs are significantly below 1, the initial product v becomes extremely tiny (e.g. 10^{-10} or smaller), making $\log v$ a large negative number. In such extreme cases, $P(v)$ itself approaches 0 (since $1 - \log v$ is huge), and consequently $P(v)^2/v$ can also become very small. In the worst case, if any factor in the product is 0, $v = 0$ and no finite smoothing can help ($P(0)$ is undefined without ε and effectively $P(v)^2/v$ remains near 0 for very small v). Thus, for very complex clauses or highly non-saturated inputs, the gradients may still collapse to nearly zero. Moreover, once we ultimately project the network to discrete weights and inputs ($h, W \in \{0, 1\}$), the gradient at those exact binary points is again zero in most directions (as discussed earlier). In summary, $P(v)$-based smoothing improves gradient flow for moderately small signals but does not fundamentally overcome vanishing gradients when training very large logical expressions. Additional strategies are required to train purely discrete models.

Gradient Grafting for discrete training. Another technique used in RRL to handle training with binary decisions is **Gradient Grafting**. The idea is to maintain two parallel models during training: a continuous one that is used for backpropagation, and a discrete one that defines the actual loss. Let θ denote the set of trainable continuous parameters (weights) and $q(\theta)$ be a binarization function that maps these to discrete values (for instance, thresholding each weight to 0 or 1). We denote by $\hat{Y} = F(\theta, X)$ the output of the continuous model on input X, and by $\bar{Y} = F(q(\theta), X)$ the output of the corresponding binarized model (i.e., the actual logical network with hard decisions). Training proceeds by computing the loss $L(\bar{Y})$ on the discrete model's output, but then updating θ using gradients from the continuous model. In formula:

$$\theta_{t+1} \;=\; \theta_t \;-\; \eta \left[\frac{\partial L(\bar{Y})}{\partial \bar{Y}}\right] \left[\frac{\partial \hat{Y}}{\partial \theta_t}\right], \tag{46}$$

where η is the learning rate. In this update, $\frac{\partial L(\bar{Y})}{\partial \bar{Y}}$ is the gradient of the loss with respect to the discrete model's output (this measures how the final loss would change if the discrete output changed), and $\frac{\partial \hat{Y}}{\partial \theta_t}$ is the Jacobian of the continuous model's output with respect to its parameters (which is well-defined and non-zero because $\hat{Y}$ is produced by smooth activations). The chain of these two terms provides an effective surrogate gradient for θ that steers the discrete model's loss $L(\bar{Y})$ downward, even though $\bar{Y}$ itself has zero or undefined gradients w.r.t. θ. In essence, the discrete network's error signal is "grafted" onto the continuous network's sensitivity.

This approach circumvents the vanishing gradient at the discrete points by always following the continuous proxy's gradients, and can successfully optimize a logical network where direct backpropagation would fail. Gradient grafting, however, comes at the cost of increased training complexity. The optimization is no longer a simple gradient descent on a single well-defined objective; instead, it couples two models (one binary, one continuous) and relies on their interplay. The update in Eq. equation 46 is not the true gradient of any single loss function with respect to θ, since $L(\bar{Y})$ is evaluated on a different forward path than $\hat{Y}$. Thus, careful tuning and heuristics may be needed to make this training scheme converge reliably. Nonetheless, this method has been shown to help train discrete logical networks that would otherwise be stuck due to vanishing gradients.

In summary, RRL's approach to training discrete logical rules involves smoothing the logical operations (to keep gradients alive) and using a hybrid training procedure to inject discrete loss information, partially mitigating the vanishing gradient issue.

J.4 RNS: ± 1 Encoding and min/max Logical Activations

RNS takes a fundamentally different route to avoid vanishing gradients: it redesigns the logical neuron computations and encoding so that gradients do not collapse in the first place, even at discrete points. There are two key aspects to this strategy:

(1) ± 1 Binary Encoding (No Zeros). Instead of representing False as 0 and True as 1, RNS encodes Boolean values as -1 (false) and $+1$ (true). All intermediate logical signals in the network use this $\{-1, +1\}$ domain. The advantage of this encoding is that multiplying by -1 flips a signal's truth value while multiplying by $+1$ leaves it unchanged – and importantly, **zero is never an output of a logical unit.** This means we never encounter the situation of a gradient being multiplied by 0 (which was a major issue in the $0/1$ encoding). By design, removing 0 from the state space ensures that no single input can annihilate the gradient by being zero.

(2) Min/Max-Based Logical Operations (Non-multiplicative). Instead of using products to represent AND/OR, RNS uses *piecewise-linear extremum functions* that exactly mimic logic under the ± 1 encoding. Specifically, if a set of inputs $\{x_1, x_2, \ldots, x_n\}$ are all either -1 or $+1$, we define:

$$\text{AND}(x_1, \ldots, x_n) \;=\; \min\{x_1, \ldots, x_n\}, \qquad \text{OR}(x_1, \ldots, x_n) \;=\; \max\{x_1, \ldots, x_n\}.$$

For example, if any $x_j = -1$ (False), the minimum will be -1, correctly giving AND = False; only if all $x_j = +1$ will the minimum be $+1$ (True). Similarly, the maximum returns $+1$ if any input is true. These operators perfectly realize the Boolean logic of AND/OR for ± 1 inputs, but unlike products, they are not multiplicative and do not cause exponential shrinkage of gradients. Instead,

they behave as selectors: the AND output is whichever input is the "most false" (most negative), and the OR output is the "most true" (most positive).

Because $\min$ and $\max$ are piecewise linear functions, we can define well-behaved subgradients for them. Suppose $y = \max(x_1, \ldots, x_n)$. In the case of no ties, exactly one input attains this maximum value; say x_k is the largest. A small change in x_k will change y equally (a 1-to-1 slope), whereas changes in any other x_j (that are below the max) have no effect on y (as long as they don't exceed x_k). One convenient choice of subgradient is to assign $\frac{\partial y}{\partial x_k} = 1$ for one of the maximal indices k, and $\frac{\partial y}{\partial x_j} = 0$ for all other $j \neq k$. More generally, when there are ties (multiple inputs share the max value), the gradient can be split among them. A common subgradient is:

$$\frac{\partial y}{\partial x_i} = \begin{cases} \frac{1}{|M|} & \text{if } i \in M, \\ 0 & \text{if } i \notin M, \end{cases} \qquad \text{where } M = \{j : x_j = \max_k x_k\} . \tag{47}$$

An analogous definition applies for $y = \min(x_1, \ldots, x_n)$: the subgradient is $1/|m|$ for inputs i that attain the minimum and 0 for others (where $m = \{j : x_j = \min_k x_k\}$). In words, the gradient of a $\max$ gate is distributed equally to the input(s) that are currently "winning" (i.e., the True inputs in an OR, or the least False inputs in an AND when all are true). Crucially, this gradient *does not vanish with* n: at least one input of an AND/OR receives a substantial gradient (of order 1), indicating it is responsible for the output. Even in the worst case of a tie among all n inputs (e.g., all inputs are -1 for an AND, or all $+1$ for an OR), each input would get a gradient of $1/n$ – which decays only linearly with n, not exponentially. In most cases, only one or a few inputs determine the extremum, and they get a full magnitude gradient. There is no multiplicative chaining of many factors as in RRL's F_c or F_d products. Additionally, negation in RNS is handled simply by a sign-flip: each literal may have a trainable indicator that either uses the variable as $+1$ (positive literal) or negates it (-1). Negation is just multiplication by -1, which is trivially differentiable (its subgradient is -1 when active, or essentially treated as a constant factor).

Training with Straight-Through Estimators (STE). Both the ± 1 encoding and the use of $\min$ / $\max$ activations are still inherently non-differentiable as functions (the $\max$ has a kink where two inputs are equal, and the sign function that produces ± 1 outputs is discontinuous). However, RNS is amenable to standard techniques for training binarized networks, particularly the *straight-through estimator* (STE) for gradients. In backpropagation, whenever a gradient hits a non-differentiable threshold (such as the sign function that produces ± 1 outputs), RNS simply treats that operation as an identity mapping for the sake of gradient computation – meaning the gradient is "straight-through" passed as if the threshold were not there **?**. This is a common approach in binary neural networks to approximate gradients through quantization. In the case of the $\min$ / $\max$ gates, we use the subgradient defined in Eq. equation 47 during backpropagation. The combination of STE and subgradient for $\min$ / $\max$ ensures that at *discrete* operating points, the network can still propagate meaningful gradients.

For example, consider an AND implemented as $y = \min(x_1, \ldots, x_n)$ with all inputs either $+1$ or -1. Suppose $y = -1$ (False) because at least one $x_j = -1$. In the forward pass, this yields $y = -1$. In the backward pass, the subgradient will assign a non-zero derivative to each input that was equal to the minimum (i.e., to each x_j that is -1). This correctly identifies that increasing any of those x_j from -1 to $+1$ (i.e., flipping a false literal to true) would change the AND output to a higher value (potentially making the whole conjunction true if that was the only false). Thus, even though the forward function is flat for changes in any single input (since you need all falses to flip for the AND to turn true), the chosen subgradient provides a direction for learning: it tells the network to try flipping those false literals. By contrast, in a multiplicative AND, if more than one input is false, the true gradient is strictly zero for a single-input change – there is no signal at all indicating which inputs are candidates to flip. An STE cannot magically invent a meaningful gradient in that case without risking divergence from the actual function's behavior. In RNS, however, the STE is effectively aligning with the inherent combinatorial structure of the logic: it distributes blame to all currently-false conditions for an AND (or to all currently-true conditions for an OR that outputs true, in case of ties). This yields a robust training signal even in fully discrete regimes.

J.5 WHY RNS AVOIDS VANISHING GRADIENTS

In summary, the design choices in RNS eliminate the root causes of vanishing gradients that plague RRL's product-based logic:

- **Activation Form:** RRL uses multiplicative activations (product for AND, complement-product for OR), which cause gradient magnitudes to contract multiplicatively with clause size. Even the improved RRL with the $P(v)$ log-smoothing still relies on a product (modulated by a corrective factor) and can suffer when many terms are far from 1. In contrast, RNS uses $\min / \max$ activations, which are piecewise linear selectors. There is no long product over many inputs; the gradient comes from identifying the extremal input(s). This fundamental difference means RNS does not inherently squeeze the gradient as the number of inputs grows.
- **Binary Encoding:** RRL's $0/1$ encoding introduces an actual zero output (false = 0) that can outright nullify gradients (any factor of 0 in a product zeroes out the whole gradient). RNS's ± 1 encoding avoids this so that a false value is -1 instead of 0, so it never multiplicatively annihilates an entire gradient. Every input always has the potential to influence the output by changing sign, and the gradient methods used in RNS take advantage of that.
- **Gradient Scaling with Clause Size:** In RRL, the magnitude of a gradient component is on the order of $\prod_{k \neq j} \alpha_k$ for some $\alpha_k \in [0, 1]$ related to each of the other inputs (e.g. $F_c(h_k, W_{i,k})$). This leads to an *exponential decay* in gradient as n increases, unless all α_k are extremely close to 1. The log-domain trick rescales this by a factor of $P(v)^2/v$, which slows the decay but does not stop it for very large n. In RNS, by contrast, a gradient to a decisive input is $\mathcal{O}(1)$ – it does not diminish with n at all (one input typically gets full gradient 1 if it alone determines the output). In worst-case tie scenarios, the gradient might be split among n inputs, giving each about $1/n$, i.e., decaying linearly with n, which is far milder than exponential decay. Thus, RNS scales to clauses with many literals without facing an overwhelming vanishing gradient issue.
- **Discrete Training Behavior:** RRL must resort to special training techniques like gradient grafting to handle discrete parameters, because directly backpropagating through a binarized product-form network yields zero gradients in most places. RNS, on the other hand, can be trained with standard backpropagation augmented with STE, which is a simpler and more direct approach. The reason STE works well for RNS is that its surrogate gradient (identity for the sign function, plus the $\min / \max$ subgradient) aligns with actual changes that would affect the output. In RRL's case, an STE would have to assign gradients to inputs that in reality do not affect the output unless combined with others – a fundamentally ambiguous credit assignment. Therefore, RNS avoids the need for a two-model training setup; one can optimize the ± 1 network in one go by using surrogate gradients, without the gradients vanishing.

Overall, by using ± 1 encoding and min/max logic, RNS preserves discrete interpretability while ensuring that gradient signals remain strong and informative. The network is able to learn large logical formulas (many-input clauses) because it never multiplies a long chain of fractional terms during backpropagation. Each logical neuron in RNS passes a gradient to the input(s) that currently determine its output, providing a clear learning direction. These properties enable RNS to train effectively, where a product-based logical network would struggle or stall due to vanishing gradients.