SUMMER: Recursive Zero-Knowledge Proofs for Scalable RNN Training

Yuange Li
Rutgers University
yl1407@rutgers.edu

Xiong Fan
Cysic, Inc.
leofanxiong@gmail.com

Abstract

Zero-knowledge proofs of training (zkPoT) enable a prover to certify that a model was trained on a committed dataset under a prescribed algorithm without revealing the model or data. Proving recurrent neural network (RNN) training is challenging due to hidden-state recurrence and cross-step weight sharing, which require proofs to enforce recurrence, gradients, and nonlinear activations across time.

We present SUMMER (SUMcheck and MERkle tree), a recursive zkPoT for scalable RNNs. SUMMER generates sumcheck-based proofs that backpropagation through time (BPTT) was computed correctly over a quantized finite field, while nonlinearities such as tanh and softmax are validated by lookup arguments. Per-step commitments and proofs are folded with Merkle trees, yielding a final commitment and a succinct proof whose size and verification time are independent of the number of iterations.

SUMMER offers (i) the first end-to-end zkPoT for RNN training, including forward, backward, and parameter updates; (ii) the first use of LogUp for nonlinear operations with a batched interface; and (iii) efficient recursive composition of lookup and sumcheck proofs. On a Mini-Char-RNN with 12M parameters, the prover runs in 70.1 seconds per iteration, 8.5× faster and 11.6× more memory efficient than the IVC baseline, with 165 kilobyte proofs verified in 20 milliseconds.

1 Introduction

Artificial intelligence has advanced rapidly with large language models (LLMs), which power applications from chatbots to code generation. These models build on earlier architectures such as recurrent neural networks (RNNs), the first to capture sequential dependencies by processing inputs step by step. While modern LLMs use Transformers for efficiency and scale, RNNs remain the fundamental precursor to today's generative AI.

Yet both RNNs and LLMs face a trust problem: users cannot easily verify how models were trained, whether data sources were valid, or if outputs are faithful to the claimed computation. This black-box nature raises concerns around integrity, reproducibility, and accountability, especially in high-stakes domains like medicine or finance.

Zero-knowledge proofs (ZKPs) offer a path forward. Originally developed to prove statements without revealing secrets, ZKPs now power applications from blockchain scalability to privacy-preserving authentication. Applied to machine learning, they can prove the correctness of training and inference without exposing data or proprietary weights. For RNNs, ZKPs can certify forward passes, activations, and gradient updates, and—through recursion—yield succinct proofs that cover entire training runs. In this way, ZKPs help bridge the trust gap, enabling verifiable AI systems that are not only powerful but also provably reliable.

While much of the current research on verifiable AI focuses on proof of inference, the challenge of proof of training is both harder and more critical. Training involves multiple iterative steps: forward passes through the network, non-linear activations, loss evaluations, and gradient-based updates to model parameters. Each of these stages introduces additional complexity that must be faithfully proven, and when repeated across thousands of batches, the proof system must scale without substansive increase in size. Verifying training is essential because it establishes trust in the origin of the model itself. Without proof of training, one can only check outputs from a black box; with it, we can ensure that the entire learning process was conducted honestly, on the claimed data, and without hidden manipulations. This shift from verifying predictions to verifying the full lifecycle of model creation represents a crucial step toward trustworthy AI.

1.1 Related Works

With performance improving in ZK proof generation, the community has witnessed an enormous amount of work on verifiable AI. We start from the more related verifiable training line of work. In zkDL [24], the authors introduced how to handle the Relu function and more generally the non-linear

activation function. Their system relies on Perdersen commitment scheme and applies the GKR proof system layer-by-layer, showing the feasibility result but leaving scalability issues open for large models. zkPoT [9] expands the scope of zkDL work, but its construction cannot be naturally extended to deep neural networks due to fundamental limitations in handling complex nonlinear functions and iterative updates.

The second aspect of verifiable AI is about integrity in inference. zkCNN [16] proposed one of the first approaches to verifiable convolutional neural networks, introducing specialized sumcheck protocols tailored to two-dimensional convolutions as well as to FFT operations that arise in CNN spectral domain acceleration. After that, several work [4, 25, 28] improves on several sectors of verifiable inference, such as optimized Boolean representation, advanced constraint representation, and expansion to large-scale models. The most recent work [21] represents the next step forward, in delivering a system explicitly designed to prove the correctness of GPT-scale architectures.

The most related work that we should consider is Kaizen [1], where they designed and implemented a zkPoT framework targeted at deep neural networks (DNNs). In their construction, they used a two-step process: First, a GKR system is deployed to prove traning result of each iteration, and then these step-proofs are recursively composed to obtain a succinct proof. However, as their construction heavily relies on bit-decomposition for nonlinear computation, such as exponentiation, Relu and softmax, which incurs significant increase in the number of constraints and thus the proving time. After carefully investing in their code [1], we found several issues in their implementation. Kaizen falls short of its claims. It only supports ReLU activations, omitting common functions like and tanh, and does not provide a true end-to-end training proof—its design repeats per-iteration step proofs within a recursive wrapper, without handling parameter updates across iterations. Moreover, the implementation shows engineering and correctness flaws: in aggregate commitments, only the last matrix is kept instead of forming proper combinations; inconsistent quantization schemes introduce ambiguity; and the benchmarks contain repeated or missing entries, raising concerns about result reliability.

1.2 Our Contributions

Our system constructs an end-to-end prover and verifier pipeline that encompasses every stage of the training process. This includes the forward pass, the handling of non-linear activations, the calculation of loss, and the backpropagation steps required to update gradients and weights. By capturing the complete training loop, SUMMER makes it possible to prove correctness and integrity of the training procedure itself, ensuring that the final model parameters were derived honestly.

[1] https://github.com/zkPoTs/kaizen

Our work introduces several novel contributions to the design of zero-knowledge proofs of training. First of all, we present the *first ZKP system for RNN training*, not merely inference. SUMMER builds a unified prover/verifier pipeline that covers the entire training loop, including the forward pass, nonlinear activations, loss computation, gradient derivation, and parameter updates, thereby certifying that the final model parameters arise from an honest execution of BPTT. In the implementation, we are the first to apply the LogUp lookup argument [11] to training, proving nonlinearities like tanh and Softmax/exp, and introduce a batching interface (ExpBatch) with sumcheck checks for efficient activations. We also design a recursive composition framework that folds iterations into a single succinct proof, binding all lookup claims and updates while keeping proof size and verification time constant. Finally, we evaluate SUMMER on RNNs such as Mini-Char-RNN [13], showing scalability: for a 12M-parameter model, proofs take 70.1s per iteration, are 165KB verified in 20ms, and improve proving speed 8.5× and memory use 11.6× over the state-of-the-art [15].

2 Preliminaries

Notation. In this work, we use λ to denote the security parameter and negl to denote a negligible function. We use $\mathbb{F}$ to denote a finite field. "PPT" stands for probabilistic polynomial time. We use bold lowercase letters (e.g., $\mathbf{v}, \mathbf{w}$) to denote vectors and bold uppercase letters (e.g., $\mathbf{A}, \mathbf{B}$) to denote matrices. For a vector $\mathbf{v} \in \mathbb{F}^\ell$, we use lowercase letters to refer to its components; that is, we write $\mathbf{v} = (v_1, \ldots, v_\ell)$.

2.1 RNN and Training

Recurrent Neural Networks (RNNs) [7] are deep learning models that capture the dynamics of sequences via recurrent connections, which can be thought of as cycles in the network of nodes.

An RNN operates on an input space and an internal state space, which maintains a memory trace of previously processed inputs. The state space enables the representation of sequentially extended dependencies over unspecified intervals according to

$$\mathbf{y}^{(t)} = G(\mathbf{s}^{(t)}) \quad \mathbf{s}^{(t)} = F(\mathbf{s}^{(t-1)}, \mathbf{x}^{(t)}).$$

We consider a standard Recurrent Neural Network (RNN) with an input layer, one hidden layer, and an output layer. For notational convenience, we assume that the input at time step t is already an *embedded* vector: let $\mathbf{x}^{(t)} \in \mathbb{R}^d$ denote the token embedding of token $w^{(t)}$, obtained via an embedding matrix $\mathbf{E} \in \mathbb{R}^{|\mathcal{V}| \times d}$. For a length-$T$ sequence we stack the inputs as $\mathbf{X} = [\mathbf{x}^{(1)}; \ldots; \mathbf{x}^{(T)}] \in \mathbb{R}^{T \times d}$.

Forward Pass. The model directly consumes the input vector $\mathbf{x}^{(t)}$ (no separate embedding step). Let $\mathbf{X}$ denote the stacked input matrix whose t-th row is $\mathbf{x}^{(t)}$. Using hidden state $\mathbf{h}^{(t)}$, weight matrices $\mathbf{W}_h, \mathbf{W}_e, \mathbf{W}_y$ for hidden, input, and output layers, respectively, and bias terms b_1, b_2, the forward update is

$$\mathbf{h}^{(t)} = \mathsf{Act}\big(\mathbf{W}_h \mathbf{h}^{(t-1)} + \mathbf{W}_e \mathbf{x}^{(t)} + b_1\big),$$
$$\hat{\mathbf{y}}^{(t)} = \mathsf{softmax}\big(\mathbf{W}_y \mathbf{h}^{(t)} + b_2\big).$$

Here, Act denotes the element-wise activation function, which introduces non-linearity into the model, enabling it to capture complex patterns. Common choices for activation functions include $\mathsf{ReLU}(x) = max(x, 0)$, $\mathsf{sigmoid}(x) = 1/(1+e^{-x})$, and $\tanh(x) = (e^{2x}-1)/(e^{2x}+1)$. For the output layer, we follow standard practice and apply the $\mathsf{Softmax}(x_i) = (e^{x_i})/(\sum_j e^{x_j})$ to normalize the logits into a probabilistic distribution over the possible output classes.

Let K be the number of output classes, and $y_k^{(t)}$ be the one-hot encoded ground-truth label for class k at time step t, with $\hat{y}_k^{(t)}$ denoting the predicted probability corresponding. The loss at time step t is computed using the cross-entropy loss: $\mathcal{L}^{(t)} = -\sum_{k=1}^{K} y_k^{(t)} \log\left(\hat{y}_k^{(t)}\right)$, while the total loss over T steps is: $\mathcal{L} = \frac{1}{T}\sum_{t=1}^{T} \mathcal{L}^{(t)}$.

Backward Pass. Due to the recursive nature of RNNs, gradients must be computed through time. This is achieved via BackPropagation Through Time (BPTT) [29]. Let $\delta^{(t)} = \hat{\mathbf{y}}^{(t)} - \mathbf{y}^{(t)}$ denote the output error at the time step t. Gradients for the output layer are

$$\frac{\partial \mathcal{L}}{\partial \mathbf{W}_y} = \delta^{(t)} \big(\mathbf{h}^{(t)}\big)^\top, \qquad \frac{\partial \mathcal{L}}{\partial b_2} = \delta^{(t)}.$$

The gradient is backpropagated through the hidden layer using the chain rule:

$$\frac{\partial \mathcal{L}}{\partial \mathbf{h}^{(t)}} = \mathbf{W}_y^\top \delta^{(t)}, \qquad \frac{\partial \mathcal{L}}{\partial \mathbf{a}^{(t)}} = \frac{\partial \mathcal{L}}{\partial \mathbf{h}^{(t)}} \circ \mathsf{Act}'\big(\mathbf{a}^{(t)}\big),$$

where the pre-activation is $\mathbf{a}^{(t)} = \mathbf{W}_h \mathbf{h}^{(t-1)} + \mathbf{W}_e \mathbf{x}^{(t)} + b_1$.

Gradients for the parameters of the hidden layer follow:

$$\frac{\partial \mathcal{L}}{\partial \mathbf{W}_h} = \left(\frac{\partial \mathcal{L}}{\partial \mathbf{a}^{(t)}}\right)\big(\mathbf{h}^{(t-1)}\big)^\top, \qquad \frac{\partial \mathcal{L}}{\partial \mathbf{W}_e} = \left(\frac{\partial \mathcal{L}}{\partial \mathbf{a}^{(t)}}\right)\big(\mathbf{x}^{(t)}\big)^\top,$$
$$\frac{\partial \mathcal{L}}{\partial b_1} = \frac{\partial \mathcal{L}}{\partial \mathbf{a}^{(t)}}.$$

Finally, gradients to the previous hidden state are:

$$\frac{\partial \mathcal{L}}{\partial \mathbf{h}^{(t-1)}} = \mathbf{W}_h^\top \left(\frac{\partial \mathcal{L}}{\partial \mathbf{a}^{(t)}}\right).$$

Parameter Update. After computing the gradients of the loss with respect to each trainable parameter using backpropagation through time (BPTT), the parameters are updated via (stochastic) gradient descent:

$$\mathbf{W}_y \leftarrow \mathbf{W}_y - \eta \cdot \frac{\partial \mathcal{L}}{\partial \mathbf{W}_y}, \qquad b_2 \leftarrow b_2 - \eta \cdot \frac{\partial \mathcal{L}}{\partial b_2},$$
$$\mathbf{W}_h \leftarrow \mathbf{W}_h - \eta \cdot \frac{\partial \mathcal{L}}{\partial \mathbf{W}_h}, \qquad \mathbf{W}_e \leftarrow \mathbf{W}_e - \eta \cdot \frac{\partial \mathcal{L}}{\partial \mathbf{W}_e},$$
$$b_1 \leftarrow b_1 - \eta \cdot \frac{\partial \mathcal{L}}{\partial b_1}.$$

where $\eta > 0$ is the *learning rate* that controls the step size, a hyperparameter that controls the size of the update step.

In practice, adaptive optimizers like Adam, RMSProp, or Adagrad [6, 12, 14] are often used to adjust the effective learning rate per-parameter; gradient clipping [20] is also commonly applied in RNNs to mitigate exploding gradients. The detailed algorithm are presented in Algorithm 1 of Appendix A.1.

2.2 Proofs, Arguments, and Zero-Knowledge

Interactive Proof. An interactive proof allows a prover $\mathcal{P}$ to convince a verifier $\mathcal{V}$ of a statement's validity through multiple rounds, with $\mathcal{V}$'s queries depending on $\mathcal{P}$'s prior answers. Such a protocol satisfies completeness if an honest proof is always accepted, and soundness if a false proof convinces the verifier only with negligible probability. Knowledge soundness further ensures a prover cannot succeed without holding a valid witness. When soundness holds only against computationally bounded provers, the system is called an argument. A proof/argument is zero-knowledge if it reveals nothing about the witness beyond the truth of the statement, and succinct if $\mathcal{V}$'s runtime and communication are only $\mathrm{poly}(\lambda, |x|, \log|w|)$.

A protocol is public-coin if the verifier's messages are independent of the prover's. Via the Fiat–Shamir transformation [8], such protocols can be converted into non-interactive proofs in the random oracle model.

Polynomial Commitment Schemes. A polynomial commitment scheme (PCS) lets a prover $\mathcal{P}$ commit to a polynomial and later prove its evaluation at chosen points. A PCS is hiding if the commitment leaks no information about the polynomial, and binding if a prover cannot open one commitment to different values at the same point. Knowledge soundness ensures that any valid evaluation proof implies knowledge of the committed polynomial, while zero-knowledge guarantees the proof reveals nothing beyond the claimed evaluation. Formal definitions appear in Appendix A.2.

Incrementally Verifiable Computation (IVC). Incrementally Verifiable Computation (IVC) [27] certifies long computations by proving each step while keeping verification cost

independent of the total number of steps. The computation is modeled as $z_{i+1} = F(z_i, \omega_i)$, with z_i as public state and ω_i as the private witness.

Each step is wrapped in an augment function [15] F_A, which, given $(i{+}1, z_0, z_i, \omega_i, \pi_i, s_i)$, outputs $(\mathsf{verdict}i, zi+1, s_{i+1})$. The in-circuit verifier $\mathcal{V}_0$ checks the prior proof, enforcing $\mathsf{verdict}i = 1$ before advancing. The accumulator $si+1$ stores minimal data (e.g., counters, digests) needed for recursion.

At each round, the prover evaluates F_A in-circuit and produces a succinct proof Π_{i+1} showing the prior proof was valid, the state transitioned correctly, and the accumulator updated consistently. The external verifier need only check the final (or any chosen) succinct proof, plus a constant number of deferred openings, rather than re-verifying all steps.

2.3 GKR-based Proof Systems

Multilinear Extension. Let $f : \{0,1\}^\ell \to \mathbb{F}$ be a function. The *multilinear extension* of f is defined as the unique polynomial $\widetilde{f} : \mathbb{F}^\ell \to \mathbb{F}$ such that $\widetilde{f}(x_1, \ldots, x_\ell) = f(x_1, \ldots, x_\ell)$ for all $x_1, \ldots, x_\ell \in \{0,1\}$.

$\widetilde{f}$ can be expressed as:

$$\widetilde{f}(x_1, \ldots, x_\ell) = \sum_{b \in \{0,1\}^\ell} L_b(x_1, \ldots, x_\ell) \cdot f(b).$$

Here, L_b is called the *Lagrange kernel* and is defined as

$$L_b(x_1, \ldots, x_\ell) = \prod_{i=1}^{\ell} (b_i x_i + (1 - b_i)(1 - x_i)),$$

where b_i is the i-th bit of b. The crucial property of the Lagrange kernel is that if $b, b' \in \{0,1\}^\ell$, $L_b(b') = 1$ if and only if $b' = b$, otherwise $L_b(b') = 0$.

Merkle Tree. Merkle tree [18] is a cryptographic primitive that is used to commit to a vector and open it at an index with logarithmic proof size and verification time. It consists of three algorithms:

- $\mathsf{root} \leftarrow \mathsf{MT.Commit}(x; r)$: on input a vector x and randomness r, the commitment algorithm returns the root root of the Merkle tree as a commitment to x.
- $(x_i, \pi_i) \leftarrow \mathsf{MT.Open}(i, x; r)$: on input a vector x, an opening index i and randomness r, the opening algorithm returns a vector element x_i and a proof π_i.
- $(1, 0) \leftarrow \mathsf{MT.Verify}(\mathsf{root}, i, x_i, \pi_i)$: on input a commitment root, an index i, an element x_i, and a proof π_i, the verification algorithm returns 1 if the proof is valid, otherwise returns 0.

Sumcheck Protocol. The *sumcheck protocol* [17] is an interactive proof system that enables a verifier to check whether the sum of a multivariate polynomial over an exponentially large domain is equal to the claimed target value, without evaluating all terms directly.

Given an ℓ-variate polynomial $g : \mathbb{F}^\ell \to \mathbb{F}$ over a finite field $\mathbb{F}$, define the total sum

$$T := \sum_{b_1 \in \{0,1\}} \sum_{b_2 \in \{0,1\}} \cdots \sum_{b_\ell \in \{0,1\}} g(b_1, \ldots, b_\ell),$$

which can equivalently be written as $T := \sum_{x \in \{0,1\}^\ell} g(x)$.

The protocol proceeds in ℓ rounds. In each round, the prover sends a univariate polynomial, and the verifier responds with a random challenge. At the end of the protocol, the verifier checks a single evaluation of g at a random point $r \in \mathbb{F}^\ell$. For completeness, we describe the detailed protocol in Protocol A.3 of Appendix A.3.

GKR Protocol. Goldwasser, Kalai, and Rothblum introduced the celebrated GKR [10] protocol, a highly efficient interactive proof system to verify computations represented by layered arithmetic circuits. Let $\mathcal{C} : \mathbb{F}^n \to \mathbb{F}^k$ be a layered arithmetic circuit with d layers over a finite field $\mathbb{F}$. Let layer 0 be the output layer and layer d the input layer. The prover $\mathcal{P}$ wants to convince a verifier $\mathcal{V}$ that $\mathsf{out} = \mathcal{C}(\mathsf{input})$ where in is the input from $\mathcal{V}$ and out is the corresponding output computed by the circuit.

Let S_i be the number of gates in the i-th layer and let $s_i = \lceil \log S_i \rceil$. We define a function $V_i : \{0,1\}^{s_i} \to \mathbb{F}$ that takes binary inputs $b \in \{0,1\}^{s_i}$ and outputs the output of gate b in layer i, where b is called the *gate label*. Each gate in layer i is wired to two gates in layer $i+1$ through fixed wiring patterns, which specify the inputs used for each gate's computation. These wires can be described using *wire predicates*, which are Boolean functions that indicate whether a tuple of gate indices corresponds to a valid connection. The multiplicative wiring predicate $\mathsf{mult}_i(u, v, w) : \{0,1\}^{s_i + 2s_{i+1}} \to \{0,1\}$ outputs 1 if gate u in layer i computes the product of gates v and w from layer $i+1$. Similarly, the additive wiring predicate $\mathsf{add}_i(u, v, w)$ captures the addition gates.

We then extend V_i to its multilinear extension $\widetilde{V}_i : \mathbb{F}^{s_i} \to \mathbb{F}$. Then $\widetilde{V}_i$ can be computed as:

$$\begin{aligned} \widetilde{V}_i(u) = \sum_{v,w \in \{0,1\}^{s_{i+1}}} & (\widetilde{\mathsf{add}}_i(u, v, w) \cdot (\widetilde{V}_{i+1}(v) + \widetilde{V}_{i+1}(w)) \\ & + (\widetilde{\mathsf{mult}}_i(u, v, w) \cdot (\widetilde{V}_{i+1}(v) \cdot \widetilde{V}_{i+1}(w)) \end{aligned} \tag{1}$$

for any $u \in \{0,1\}^{s_i}$.

Once $\mathcal{V}$ receives the claimed output from $\mathcal{P}$, $\mathcal{V}$ samples a random point r_0 and evaluates $\widetilde{V}_0(r_0)$. In the first round, $\mathcal{V}$ and $\mathcal{P}$ run the sumcheck protocol on Equation 1. At the end of the protocol $\mathcal{V}$ receives two evaluations $\widetilde{V}_1(r_{1,0}), \widetilde{V}_1(r_{1,1})$ of $\widetilde{V}_1$ at two distinct random point $r_{1,0}, r_{1,1}$. $\mathcal{V}$ randomly samples

$\alpha, \beta \in \mathbb{F}$ and computes $\alpha\widetilde{V}_1(r_{1,0}) + \beta\widetilde{V}_1(r_{1,1})$. Then $\mathcal{V}$ and $\mathcal{P}$ run sumcheck protocol on

$$\alpha\widetilde{V}_1(r_{1,0}) + \beta\widetilde{V}_1(r_{1,1}) = \sum_{x,y\in\{0,1\}^{s_2}} \big(\alpha \cdot \tilde{\mathrm{mult}}_1(r_{1,0},x,y) + \beta \cdot \tilde{\mathrm{mult}}_1(r_{1,1},x,y)\big) \cdot \tilde{V}_2(x) \cdot \tilde{V}_2(y) + \big(\alpha \cdot \tilde{\mathrm{add}}_1(r_{1,0},x,y) + \beta \cdot \tilde{\mathrm{add}}_1(r_{1,1},x,y)\big) \cdot (\tilde{V}_2(x) + \tilde{V}_2(y)).$$

At the end of the protocol, $\mathcal{V}$ receives two claims about $\widetilde{V}_2$, computes their random linear combination, and proceeds to an above layer recursively until the input layer. At input layer d, $\mathcal{V}$ can check the evaluations of $\widetilde{V}_d$ by querying the oracle. The detailed protocol is shown in Protocol A.4 of Appendix A.4.

2.4 Lookup Arguments

Lookup arguments are a powerful cryptographic primitive for proving statements about every element of a *private* vector $\mathbf{v}$ is contained in a *public*, usually pre-computed table T. This is particularly useful in contexts where a computation involves repeated accesses to a set of predetermined values. Formally, the statement proven in a lookup argument is defined as:

Definition 1 (Lookup Arugment [23])**.** *Given a commitment* cm_a *and a public set* T *of* N *field elements, represented as vector* $\mathbf{t} = (t_0, \ldots, t_{N-1}) \in \mathbb{F}^N$ *to which the verifier has provided a commitment* cm_t*, the prover knows an opening* $a = (a_0, \ldots, a_{m-1}) \in \mathbb{F}^m$ *of* cm_a *such that for each* $i = 0, \ldots, m-1$*, there is a* $j \in \{0, \ldots, N-1\}$ *such that* $a_i = t_j$*.*

Modern ZKP systems rely on lookup arguments to capture non-linear relations compactly. In RNN trainings, numerous non-linear operations, such as activation functions (e.g., $\mathsf{ReLU}, \mathsf{tanh}, \mathsf{softmax}$), range check, and comparison relations are naturally encoded as lookups.

We highlight two complementary lines that are widely used in practice. The *Lasso-style* [22, 23] approach decomposes the *structured* public table T into much smaller subtables, so that each query $\mathbf{v}[i] \in T$ reduces to consistent sub-queries. The other approach, known as *LogUp-style* [11, 19], utilizes a logarithmic derivative to certify membership through a single rational identity in a random challenge c:

$$\sum_{i=0}^{m-1} \frac{1}{c - \mathbf{v}[i]} = \sum_{j=0}^{n-1} \frac{\mathsf{mult}[j]}{c - T[j]}$$

where mult denotes the multiplicity that counts the number of times each element $T[j]$ is included in the queries.

In our work, we adopt *Tassle* [5], which synthesizes Lasso's decomposability with a LogUp-style rational sumcheck, thereby significantly reducing the commitment costs. We precompute and commit shared activation and auxiliary tables and batch all per-time-step queries across the unrolled sequence, achieving linear prover cost in the number of lookups and few commitment openings for the verifier. For technical details of TaSSLE, please refer to [5].

3 Technical Overview

3.1 Quantization

Neural training uses floating-point arithmetic that is incompatible with arithmetic circuits in zero-knowledge proof. While several schemes exist (e.g., uniform affine quantization [16, 21]), SUMMER adopts a power-of-two fixed point-encoding over a large prime field to keep constraints simple, avoid per-tensor zero-points, and eliminate bit-decomposition during rescaling.

Let $\mathbb{F}_{p^2}$ denote a quadratic extension prime field with a Mersenne prime p. An element is represented as $a + bu$ with $u^2 = v$ a quadratic non-residue. For the training computation itself, we embed all quantized values in the base subfield by setting the imaginary part to zero, i.e. $(b = 0)$, so arithmetic behaves exactly like $\mathbb{F}_p$ while the extension field remains available for FFT-friendly commitments and recursive composition.

Quantization and Dequantization. We quantize real numbers with Q fractional bits by scaling with 2^Q and rounding *toward zero*: $s = \mathsf{trunc}(r \cdot 2^Q) \in \mathbb{Z}$. Before embedding, we apply symmetric clipping to avoid wrap-around:

$$s \leftarrow \mathsf{clip}_q(s) \in \big[-(2^{q-1}), 2^{q-1} - 1\big],$$

and store $\mathsf{quantize}(r) = s \bmod p \in \mathbb{F}_p$. Positive codes are kept as-is; negative codes $s < 0$ are represented as $p + s$ (the high half of $\mathbb{F}_p$). Decoding lifts $a \in \mathbb{F}_p$ to its signed representative $\tilde{s} \in (-\frac{p}{2}, \frac{p}{2}]$ and returns $\mathsf{dequantize}(a, depth) = \tilde{s}/(2^Q)^{depth}$. We use no per-tensor zero-points; symmetric clipping ensures sign disambiguation and prevents rare overflows without changing the arithmetic in the base subfield.

Integer arithmetic after quantization. Addition and subtraction operate directly in $\mathbb{F}_p$ and preserve the Q-fraction scale. For multiplication, if s_a, s_b are the encoded integers, the product $z = s_a s_b$ has scale $2Q$; we downscale by a power of two using shift (Euclidean division by 2^Q) and take the returned quotient as the rescaled result at Q fractional bits.

3.2 Sumcheck-based Proofs for Backpropagation Through Time (PBPTT)

Our PBPTT protocol attests to the correctness of the RNN training by seperating the computation into two parts: (i) *linear operations*, including matrix-matrix/vector products, Hadamard products, and summations; (ii) nonlinear maps, including activations such as $\exp$ ($\tanh$, $\mathsf{Softmax}$) and $\max$ (ReLU). For composability and verifier efficiency, we use a sumcheck-based approach throughout. All linear operations are reduced to matrix multiplication and proved with a time

optimal matrix multiplication sumcheck protocol, and all nonlinearities are verified via lookup arguments over fixed point tables produced in the quantization step.

Linear Operations. We reduce every linear subroutine to matrix multiplication and invoke Thaler's time optimal sumcheck protocol for matrix multiplication [26]. We denote this subprotocol by $\mathsf{Sum}_{\mathsf{MatMul}}$, and apply it uniformly to rectangular shapes $\mathbf{M}_1 \in \mathbb{F}^{p\times t}, \mathbf{M}_2 \in \mathbb{F}^{q\times t}$, and $\mathbf{M}_1\mathbf{M}_2^\top \in \mathbb{F}^{p\times q}$. In particular, for $n \times n$ inputs the prover runs in $\widetilde{O}(n^2)$ time.

Matrix–vector multiplication is treated as a special case of this canonical form.

$$\text{Matrix vector:} \quad \mathbf{y} = \mathbf{M}\mathbf{x} \quad (\mathbf{M} \in \mathbb{F}^{p\times q},\ \mathbf{x} \in \mathbb{F}^q,\ \mathbf{y} \in \mathbb{F}^p)$$

Hadamard multiplication is represented as multiplication by a diagonal operator.

$$\text{Hadamard:} \quad \mathbf{x}\circ\mathbf{y} = \operatorname{diag}(\mathbf{x})\,\mathbf{y} \qquad (\mathbf{x},\mathbf{y} \in \mathbb{F}^q,\ \operatorname{diag}(\mathbf{x}) \in \mathbb{F}^{q\times q},\ \mathbf{x}\odot\mathbf{y} \in \mathbb{F}^q)$$

Summation is handled by stacking and concatenation so that the total becomes a single multiplication.

$$\text{Summation:} \quad \sum_{t=1}^{T} \mathbf{M}^{(t)}\mathbf{x}^{(t)} = \begin{bmatrix}\mathbf{M}^{(1)} & \cdots & \mathbf{M}^{(T)}\end{bmatrix} \begin{bmatrix}\mathbf{x}^{(1)}\\ \vdots \\ \mathbf{x}^{(T)}\end{bmatrix}$$
$$(\mathbf{M}^{(t)} \in \mathbb{F}^{p\times q},\ \mathbf{x}^{(t)} \in \mathbb{F}^q,\ \sum_t \mathbf{M}^{(t)}\mathbf{x}^{(t)} \in \mathbb{F}^p)$$

Under these reductions, $\mathsf{Sum}_{\mathsf{MatMul}}$ proves the matrix multiplication claims that arise from all linear layers in the forward and backward passes. Henceforth, whenever such a linear operation appears, we simply invoke $\mathsf{Sum}_{\mathsf{MatMul}}$ on its canonical reduction.

Non-linear operations. Nonlinear computations in RNNs include max and exponentiations. These are costly to arithmetize inside a zero-knowledge arithmetic circuit. As an example, one $\mathsf{softmax}$ requires exponentiations and a normalization, which leads to many constraints if expanded directly. We avoid this expansion by using lookup arguments into public pre-committed tables under fixed-point quantization. For activations we certify membership of input–output pairs in the corresponding table.

$$\begin{aligned}(x,y) \in T_{\mathsf{ReLU}} \quad &\text{encodes} \quad y = \max\{x,0\}\\ (z,u) \in T_{\exp} \quad &\text{encodes} \quad u = \exp(z)\\ p_j = \frac{u_j}{\sum_\ell u_\ell} \quad &\text{with} \quad (z_j,u_j) \in T_{\exp}\end{aligned}$$

We instantiate the lookup argument with a LogUp-style scheme, TaSSLE [5]. Crucially, the resulting batched lookup proof composes via sumcheck, so all nonlinear claims in an iteration are aggregated together and then *merged* with the sumcheck proofs of linear operations. After transcript folding, the verifier checks only a constant number of polynomial openings, and the same lookup commitments are reused across time steps.

3.3 Recursive Composition of Proofs

In each iteration, the PBPTT protocol emits many artifacts, including commitment opening proofs for model parameters and lookup tables, sumcheck transcripts for linear operations, and Fiat–Shamir challenges and digests. As the number of iterations grows, the verification time, proof size, and prover memory overhead increase linearly. We address this with an incrementally verifiable computation (IVC) scheme.

We adopt the multivariate commitment aggregation method of Abbaszadeh et al. [1]. Concretely, we commit to an aggregated codeword $\mathbf{M}^* = \sum_i \alpha_i \mathbf{M}_i$ and check the linear relation at $\Theta(\lambda)$ randomly sampled codeword positions. Then we open Merkle-tree paths for $\mathbf{M}^*$ and each $\mathbf{M}_i$ to verify $\mathbf{M}^*[r,c] = \sum_i \alpha_i \mathbf{M}_i[r,c]$ using the same sumcheck framework, which composes cleanly with our other sumcheck proofs and the recursive composition.

In each iteration, the prover generates a set of evaluation claims from $\mathsf{Sum}_{\mathsf{MatMul}}$ and simple vector operations, written as $\{(\sigma_j, r_j, v_j)\}_{j=1}^q$ where σ_j commits to a polynomial, r_j is the verifier challenge, and v_j is the claimed evaluation. We compress these claims by a sumcheck-based evaluation reduction followed by random linear combination, and we aggregate the corresponding commitments using the method of Abbaszadeh et al.. Across iterations we append only the aggregated pair $(\sigma_{\mathsf{agg}}, v_{\mathsf{agg}})$ to the recursive accumulator. At the end of IVC we sample fresh challenges and apply one final evaluation reduction so that all remaining claims are bound to a constant number of points, then we perform a constant number of batched openings. This yields a constant number of commitment openings for all matrix and vector evaluations, while the intermediate reductions use the same sumcheck framework as the rest of PBPTT.

For security and efficiency, we use a k-ary recursion tree in the sense of Bitansky et al. [3]. Leave nodes are per iteration PBPTT step proofs together with their matrix/vector evaluation claims and lookup commitments. At each internal node, the parent invokes the verifier circuit on its k child proofs and, once they pass, it invokes the commitment aggregation scheme together with evaluation reduction to combine the k children into a single parent proof and commitment. This procedure repeats level by level until the root is formed (depth $\lceil \log_k T \rceil$). At the root, the external verifier checks one recursive proof and a constant number of aggregated openings. The design reduces the number of IVC folds, keeps verifier time and proof size logarithmic in T, and preserves soundness via the IVC properties, commitment binding, and the sumcheck-based subprotocols.

For security and efficiency, we use a k-ary recursion tree in the sense of Bitansky et al. [3]. Leaf nodes are per-iteration PBPTT step proofs together with their matrix/vector evaluation claims and lookup commitments. At each internal node, the parent invokes the verifier circuit on its k child proofs and, once they pass, it invokes the commitment aggregation scheme together with evaluation reduction to combine the k children into a single parent proof and commitment. This procedure repeats level by level until the root is formed (depth $\lceil \log_k I \rceil$), where I is the number of iterations. At the root, the external verifier checks one recursive proof and a constant number of aggregated openings. The design reduces the number of IVC folds, keeps verifier time and proof size logarithmic in I, and preserves soundness via the IVC properties, commitment binding, and the sumcheck-based subprotocols.

4 Proofs for Backpropagation Through Time for Vanilla RNN

Recall a standard Recurrent Neural Network (RNN) with an input layer, a single hidden layer, and an output layer. At iteration i, the input-label sequences are $\{\mathbf{x}_i^{(t)}\}_{t=1}^T$ and $\{\mathbf{y}_i^{(t)}\}_{t=1}^T$. At each time step t,

$$\mathbf{h}_i^{(t)} = \mathsf{Act}\big(\mathbf{W}_x \mathbf{x}_i^{(t)} + \mathbf{W}_h \mathbf{h}_i^{(t-1)} + b_1\big), \hat{\mathbf{y}}_i^{(t)} = \mathsf{softmax}\big(\mathbf{W}_y \mathbf{h}_i^{(t)} + b_2\big),$$

where Act denotes the activation function. The model is trained by minimizing cross-entropy loss per step $\mathcal{L}_i^{(t)}$, and gradients are computed via Backpropagation Through Time (BPTT). The weights $\mathbf{W}_y, \mathbf{W}_h, \mathbf{W}_x$ and the biases b_1, b_2 are updated by gradient descent with the learning rate η.

Our Proof of BPTT (PBPTT) protocol utilizes sumcheck protocol to attest the crossness of the training process and the gradient-based update equations. Concretely, the prover $\mathcal{P}$ employs a polynomial commitment scheme (PCS) to give verifier $\mathcal{V}$ oracle access to the inputs and outputs of the BPTT algorithm (Algorithm 1), together with auxiliary commitments supporting TaSSLE-based lookup arguments for non-linearities and range/compare relations.

We organize the PBPTT proofs into four stages, proceeding from output to input, with an initialization stage that fixes the committed instance.

Stage 0 (initialization). $\mathcal{P}$ executes Algorithm 1 locally for iteration i on inputs $(\{\mathbf{x}_i^{(t)}, \mathbf{y}_i^{(t)}\}_{t=1}^T, \theta^{(i)})$, recording the complete training trace (hidden states, logits, per-step gradients and updated parameters $\theta^{(i+1)}$). It encodes these arrays as low degree polynomials in the index domain (e.g., time $\times$ dimension) and invokes PCS to commit: (i) the *inputs* (previous parameters) $\mathsf{cm}_\theta^{(i)}$, (ii) the *outputs* (current parameters) $\mathsf{cm}_\theta^{(i+1)}$, and (iii) the TaSSLE *lookup witnesses*. $\mathcal{P}$ sends these commitments to $\mathcal{V}$.

Stage I (parameter-update). For each parameter block $\theta \in \{\mathbf{W}_y, \mathbf{W}_h, \mathbf{W}_x, b_1, b_2\}$, $\mathcal{V}$ samples an random challenge r_θ in the index domain of the multilinear extension of θ and requests from $\mathcal{P}$ the evaluation $\widetilde{\theta}_{i+1,\theta}$. $\mathcal{V}$ and $\mathcal{P}$ then run a (sumcheck-based) GKR protocol to verify, for each θ,

$$\widetilde{\theta}_{i+1,\theta}(r_\theta) \;=\; \widetilde{\theta}_{i,\theta}(r_\theta) \;-\; \eta \cdot \sum_{t=1}^{T} \widetilde{G}_\theta^{(t)}(r_\theta),$$

where $\widetilde{\theta}_{i,\theta}$ is the multilinear extension of the input parameters and $\widetilde{G}_\theta^{(t)}$ is the contribution of the gradient per time step to θ. At the end of the protocol, $\mathcal{V}$ obtains the random evaluations $\widetilde{\theta}_{i,\theta}(r_\theta)$ and $\{\widetilde{G}_\theta^{(t)}(r_\theta)\}_{t=1}^T$ (or their batched sum), which will be consumed in later stages.

Stage II (backward pass). $\mathcal{V}$ samples fresh challenges in the backward trace domain and, by $\mathsf{Sum}_{\mathsf{MatMul}}$, certifying BPTT recurrences and per-time-step gradient identities that justify the projected terms used in Stage 1:

$$\nabla_{\mathbf{W}_{x,i}} = \mathbf{a}_i'^{(t)} (\mathbf{x}_i^{(t)})^\top, \quad \nabla_{\mathbf{W}_{h,i}}^{(t)} = \mathbf{a}_i'^{(t)} (\mathbf{h}_i^{(t-1)})^\top$$
$$\nabla_{b_{1,i}}^{(t)} = \mathbf{a}_i'^{(t)}.$$

At the end of these subprotocols, $\mathcal{V}$ obtains the random evaluations of the MLEs $\widetilde{\mathbf{a}_i'^{(t)}}, \widetilde{\mathbf{x}_i^{(t)}}$ and $\widetilde{\mathbf{h}_i'^{(t-1)}}$ at the challenge points of the protocol. With $\widetilde{\mathbf{a}_i'}$, $\mathcal{V}$ and $\mathcal{P}$ check the nonlinear relation $\mathbf{a}_i'^{(t)} = \mathsf{Act}'(\mathbf{h}_i'^{(t)})$ using the TaSSLE lookup argument against the witnesses committed in Stage 0.

$\mathcal{V}$ then continues to verify, still via $\mathsf{Sum}_{\mathsf{MatMul}}$:

$$\mathbf{h}_i'^{(t)} = \begin{cases} \mathbf{W}_y^\top \delta_i^{(t)}, & t = T, \\ \mathbf{W}_y^\top \delta_i^{(t)} + \mathbf{W}_h^\top \mathbf{a}_i'^{(t+1)}, & t < T, \end{cases}$$

and the remaining gradient identities

$$\nabla_{\mathbf{W}_{y,i}}^{(t)} = \delta_i^{(t)} (\mathbf{h}_i^{(t)})^\top, \quad \nabla_{b_{2,i}}^{(t)} = \delta_i^{(t)}$$
$$\delta_i^{(t)} = \hat{\mathbf{y}}_i^{(t)} - \mathbf{y}_i^{(t)}.$$

This stage yields the required random evaluations of $\widetilde{\delta_i^{(t)}}$, $\widetilde{\mathbf{a}_i'^{(t)}}$, $\widetilde{\mathbf{h}_i'^{(t)}}$, consistently tying them to the projected gradient terms $\widetilde{G}_\theta^{(t)}(r_\theta)$ obtained in Stage 1.

Stage III (forward pass). With fresh challenges in the forward trace domain, $\mathcal{V}$ and $\mathcal{P}$ verify the forward equations, checking linear relations via $\mathsf{Sum}_{\mathsf{MatMul}}$ and certifying nonlinearities through TaSSLE lookups.

$$\hat{\mathbf{y}}_i^{(t)} = \mathsf{Softmax}\big(\mathbf{z}_i^{(t)}\big), \quad \mathbf{z}_i^{(t)} = \mathbf{W}_{y,i-1}\, \mathbf{h}_i^{(t)} + b_{2,i-1}$$
$$\mathbf{h}_i^{(t)} = \mathsf{Act}\big(\mathbf{a}_i^{(t)}\big), \quad \mathbf{a}_i^{(t)} = \mathbf{W}_{h,i-1}\, \mathbf{h}_i^{(t-1)} + \mathbf{W}_{x,i-1}\, \mathbf{x}_i^{(t)} + b_{1,i-1}$$

At the end of this stage, $\mathcal{V}$ obtains random evaluations of $\widetilde{\mathbf{h}_i^{(t)}}$, $\widetilde{\mathbf{z}_i^{(t)}}$ and $\widetilde{\hat{\mathbf{y}}_i^{(t)}}$ (and, when necessary, $\widetilde{\mathbf{a}_i^{(t)}}$), which are cross-checked against Stage II to ensure end-to-end consistency.

Stage IV (evaluation reduction and openings). $\mathcal{V}$ and $\mathcal{P}$ fold all evaluations emitted across Stages I–III into a *constant* number of evaluations per committed matrix using random linear combinations, producing single-point challenges $r_{\mathsf{in},*}$ for input and $r_{\mathsf{out},*}$ for output. $\mathcal{P}$ then invokes the PCS opening algorithm for each relevant commitment (including lookup-witness commitments).

$$\begin{aligned}
&\mathsf{PCS.Open}\big(\mathsf{cm}_{\mathbf{W}_{*,i-1}}, r_{\mathsf{in},\mathbf{W}_{*,i-1}}\big), \quad \mathsf{PCS.Open}\big(\mathsf{cm}_{\mathbf{W}_{*,i}}, r_{\mathsf{out},\mathbf{W}_{*,i}}\big)\\
&\mathsf{PCS.Open}\big(\mathsf{cm}_{\mathbf{b}_{*,i-1}}, r_{\mathsf{in},\mathbf{b}_{*,i-1}}\big), \quad \mathsf{PCS.Open}\big(\mathsf{cm}_{\mathbf{b}_{*,i}}, r_{\mathsf{out},\mathbf{b}_{*,i}}\big)\\
&\mathsf{PCS.Open}\big(\mathsf{cm}_{\mathbf{x}_i}, r_{\mathsf{in},\mathbf{x}_i}\big), \quad \mathsf{PCS.Open}\big(\mathsf{cm}_{\mathbf{y}_i}, r_{\mathsf{in},\mathbf{y}_i}\big)\\
&\mathsf{PCS.Open}\big(\mathsf{cm}_{L_i}, r_{\mathsf{in},L_i}\big)
\end{aligned}$$

$\mathcal{V}$ verifies these openings together with the $\mathsf{Sum}_{\mathsf{MatMul}}$/GKR transcripts to accept or reject.

Theorem 1. *The PBPTT protocol (Protocol B) satisfies completeness, knowledge soundness, and zero-knowledge.*

Completeness follows from the correctness of GKR, $\mathsf{Sum}_{\mathsf{MatMul}}$, and LogUp, an honest prover always produces an accepting transcript. Knowledge soundness holds since any forged parameter or gradient would be accepted with negligible probability via the Schwartz-lemma and the binding of PCS. Zero-knowledge follows by invoking the simulators of each component, yielding transcripts indistinguishable from real proofs. We show the formal details in Protocol B and provide the security proof in Appendix B.

Cost analysis. Fix iteration i. Let T be the number of time steps, $s_{\mathsf{in},i}$ and $s_{\mathsf{out},i}$ the sizes of the committed inputs and outputs, and q the bit-length of quantized field elements. Write $N_i = s_{\mathsf{in},i} + s_{\mathsf{out},i} + s_{\mathsf{param}} + s_{\mathsf{LogUp}}$ which captures the algebraic size of one PBPTT iteration, including inputs, outputs, parameters, and batched LogUp lookup witnesses. Since the time of PBPTT is dominated by commitments, we adopt Orion as the polynomial commitment scheme: its prover runs in $O(qN_i)$ field operations, while the verifier time and proof size are both $O(\log^2 qN_i)$. Evaluation reduction ensures only $O(1)$ openings per iteration.

5 Recursive Proof Composition

Each PBPTT iteration produces multiple subproofs, commitments, and hash digests: sumcheck transcripts for linear operations and LogUp-based lookups, and PCS opening proofs for model and lookup commitments, together with Fiat–Shamir challenges and transcript digests. These subproofs, if verified sequentially, would incur verification time, proof size, and memory overhead that grow linearly with the number of iterations. We address this with incrementally verifiable computation (IVC).

We rely on the IVC step relation from Section 2.2. For PBPTT, we instantiate the state with the model parameters: we treat the parameter commitment from iteration i, $\mathsf{cm}_\theta^{(i)}$, as the input z_i to F_A. Given $(i{+}1, z_0, \mathsf{cm}_\theta^{(i)}, \omega_i, \pi_i, s_i)$, F_A rechecks π_i *in circuit*, enforces one round of BPTT (forward, backward, and update $\theta^{(i+1)} = \theta^{(i)} - \eta\nabla\theta^{(i)}$) using $\mathsf{Sum}_{\mathsf{MatMul}}$ for linear operations and batched lookups for non-linear maps, compresses the resulting evaluation claims, and outputs the commitment of the new parameter $\mathsf{cm}_\theta^{(i+1)}$ together with the updated accumulator s_{i+1}all certified by a succinct proof.

Evaluation reduction and aggregated openings. We unify the treatment of sumcheck evaluations and PCS openings in one pipeline. After enforcing the current PBPTT constraints, the step collects evaluation claims $\{(\sigma_j, r_j, v_j)\}$ from $\mathsf{Sum}_{\mathsf{MatMul}}$ (linear kernels) and from batched lookups (nonlinear maps). We first perform a *evaluation reduction* to bind all claims to a common point (or a few points) and compress the values using the Fiat–Shamir coefficients: $v_{\mathsf{agg}} = \sum_j \lambda_j v_j$.

Because our PCS is non-homomorphic, we do not combine commitments algebraically. Instead, using the *same* coefficients $\{\lambda_j\}$ we build a fresh aggregated codeword $\mathbf{M}^* = \sum_j \lambda_j \mathbf{M}_j$, commit to it (yielding σ^*), and at $\Theta(\lambda)$ sampled coordinates open Merkle-tree paths for $\mathbf{M}^*$ and for each $\mathbf{M}_j$ to certify the linear relation $\mathbf{M}^*[r,c] = \sum_j \lambda_j \mathbf{M}_j[r,c]$. These checks reuse the same sumcheck-friendly constraints and link v_{agg} to σ^* without relying on homomorphism. Both linear and non-linear families flow through this pipeline, so the accumulator carries only $(\sigma^*, v_{\mathsf{agg}})$ and a transcript digest; in the end, we discharge them with a constant number of batched openings. The prover's work is linear in the total encoded size, while verifier time and proof size are logarithmic in the number of aggregated items [1].

Sumcheck-friendly hashing and challenges. All challenges within the step (for sumcheck and aggregation) are derived via the Fiat–Shamir transformation from a transcript that binds $(i{+}1, z_0, z_i, h_i)$ and the relevant commitments. Following [1], we instantiate a sumcheck-friendly permutation (e.g., MiMC), so that the in-circuit verification of hash rounds decomposes into simple linear maps.

Tree-based organization. To reduce recursion depth, we can organize steps in a k-ary tree: leaves are per-iteration step proofs; each internal node verifies its k children in-circuit and reapplies the reduction/aggregation pipeline to emit one parent proof. The depth is $\lceil \log_k T \rceil$, and the root proof, together with the same constant number of batched openings, is the final object of verification [3].

5.1 Putting Everything Together

At the parent invocation of the augment function F_A over k child steps, we use the following *inputs*: (i) the k child transcripts and public I/O $\{(\pi_j, \mathrm{io}_j, h_j)\}_{j=1}^{k}$, where each io_j contains the pre–aggregation commitments for that step (parameter and lookup commitments), the *committed* next-iteration parameters (e.g., $\mathsf{cm}_\theta^{(i_j+1)}$), and that step's aggregated triple $(\sigma_j^*, r_{\mathsf{agg},j}, v_{\mathsf{agg},j})$; (ii) Fiat–Shamir challenge $\{\lambda_j\}$, the sampled coordinates $\mathcal{S}$ of size $\Theta(\lambda)$, and the Merkle authentication paths for the constituent codewords at $\mathcal{S}$; (iii) hash advice for the k PBPTT transcripts and for the current aggregation (to be checked by sumcheck).

Inside F_A, the value side is first aligned by an evaluation reduction that binds all per–kernel claims to a common point r_{agg} (or a small set of points) and compresses them by a random linear combination, $v_{\mathsf{agg}} = \sum_{j=1}^{k} \lambda_j v_j$. Because the underlying PCS is non-homomorphic, the commitment side is handled by constructing a fresh aggregated codeword with the *same* coefficients,

$$\mathbf{M}^* = \sum_{j=1}^{k} \lambda_j \mathbf{M}_j, \qquad \sigma^* \leftarrow \mathsf{PCS.Commit}(\mathbf{M}^*),$$

and certifying, *in circuit and without calling* PCS.Open, that for every $(r,c) \in \mathcal{S}$ the relation $\mathbf{M}^*[r,c] = \sum_{j=1}^{k} \lambda_j \mathbf{M}_j[r,c]$, holds via the provided Merkle paths. This links the reduced value v_{agg} to the new commitment σ^* while deferring openings.

The step verifier then rechecks all k child proofs in circuit and *enforces* acceptance before proceeding. For a node, the step additionally enforces the PBPTT constraints (forward, backward, and the parameter update) that relate the input commitment $\mathsf{cm}_\theta^{(i)}$ to the *committed* output $\mathsf{cm}_\theta^{(i+1)}$, producing the per-kernel evaluation claims consumed by the reduction above. For an internal node, no new PBPTT computation is introduced—only child-proof verification and aggregation are performed.

Next, the hash permutations that feed Fiat–Shamir are verified by sumcheck: one family covers the k PBPTT transcripts, and another covers the current aggregation, using a sumcheck-friendly algebraic hash (e.g., MiMC) so that the permutation rounds arithmetize to simple linear maps.

Finally, F_A records the aggregated triple $(\sigma^*, r_{\mathsf{agg}}, v_{\mathsf{agg}})$, updates a domain–separated transcript digest, and produces the next accumulator with a fixed tiny schema (counter, digest, aggregated pair, minimal metadata). For leaves, the output parameter commitment $\mathsf{cm}_\theta^{(i+1)}$ simultaneously becomes the next step's input. The outputs of this invocation are the combined verifier result (enforced to be 1 in circuit), the sumcheck proofs for the k PBPTT hashes and for the current aggregation hash, the aggregated triple $(\sigma^*, r_{\mathsf{agg}}, v_{\mathsf{agg}})$, and the updated accumulator. Crucially, no per-iteration PCS openings are performed inside the recursion; because each step publishes only the aggregated triple and the list of constituent commitments as public I/O, all evaluation openings are deferred to the end and discharged once with a constant number of batched openings.

6 SUMMER: A Recursive Framework of RNN Training

In this section, we present SUMMER, the first end-to-end verifiable training system for RNNs. We first propose a model-agnostic definition for time series models that covers our RNN instantiation and can be extended to transformer-based large language models (LLM), and then we describe the concrete design and implementation of SUMMER.

6.1 A Generic Definition

We adopt the zero-knowledge proof-of-training interface of Abbaszadeh et al. [1] and generalize it to our framework. Let the following step function $z_{i+1} = F(z_i, \omega_i)$

describe training, where the public state z_i includes model parameters θ_i and ω_i is the private per-step witness derived from the dataset and randomness.

Definition 2. *Given a time-series model, a generic state function $\mathcal{F}$ is defined as a deterministic mapping that updates the model parameters $\theta_{i+1} = \mathcal{F}(\theta_i, \omega_i)$, a zero-knowledge proof of training is a tuple of PPT algorithms* $(\mathsf{KeyGen}, \mathsf{DataCom}, \mathsf{ParCom}, \mathsf{Prove}, \mathsf{Verify}$. *Among them,* $\mathsf{pp} \leftarrow \mathsf{KeyGen}(1^\lambda)$ *computes the public parameters,* $\sigma_{\mathcal{D}} \leftarrow \mathsf{DataCom}(\mathcal{D}, r_{\mathcal{D}})$ *commits to the (entire) training dataset $\mathcal{D}$,* $\sigma_\theta \leftarrow \mathsf{ParCom}(\theta; r_\theta)$*: commit to the updated model parameters.* $(\pi_i, \sigma_{\theta_i}) \leftarrow \mathsf{Prove}(\mathsf{pp}, i, \sigma_{\mathcal{D}}, \sigma_{\theta_0}, \sigma_{\theta_{i-1}}, \theta_{i-1}, \pi_{i-1})$ *computes proof a* single *training step is correct. The prover outputs a new commitment σ_{θ_i} and a proof π_i, and* $\{0,1\} \leftarrow \mathsf{Verify}(i, \sigma_{\mathcal{D}}, \sigma_{\theta_0}, \sigma_{\theta_i}, \pi_i)$*: accepts if and only if the above step relation holds with respect to the commitments.*

This definition satisfies the essential properties of completeness, knowledge soundness and zero-knowledge as defined in [1] Section E.1.

Iteration-succinct end-to-end proof. The above definition is for single-step training. An end-to-end zkPoT for the T steps provides a composition mechanism via IVC that yields a final *single* proof Π_T for $(\sigma_{\mathcal{D}}, \sigma_{\theta_0}, \sigma_{\theta_T})$ with a verification cost and a proof size independent of T. We use this generalized form in SUMMER with time-indexed witnesses and commitments.

6.2 Implementation of SUMMER

SUMMER implements an end-to-end verifiable training pipeline that turns each PBPTT into a succinct step proof

and then composes all steps recursively. This implementation follows the technical components proposed in Section B and the IVC interface in Section 5, and instantiates the design of the time series model with RNNs as the running case.

Data/model commitments and Fiat-Shamir transformation. At initialization, we commit to the training inputs and labels, the initial model parameters, and the precomputed lookup tables for nonlinearities through a PCS. Each PBPTT iteration then packs the iteration's witness into a single vector that includes the lookup indices/values used by the activations and the updated parameters; this vector is also committed. All prover challenges are derived from a single Fiat-Shamir transcript driven by a sumcheck-friendly hash so that every random evaluation point, Merkle query coordinate, and aggregation coefficient is bound to the public transcript and ensures consistency across the circuit.

Per-iteration proving. After running PBPTT (Protocol B), the prover generates the step proof. ALL linear relations are reduced to matrix multiplication and are proved by the $\mathsf{Sum}_{\mathsf{MatMul}}$ subprotocol. Nonlinear constraints from activation functions are enforced with TaSSLE-style lookups *against the committed lookup tables*: the prover provides indexed queries and provers them with a logarithmic-derivative lookup that composes with sumcheck proofs, while deferring commitment to the recursive aggregation stage. Throughout the step we collect only a small set of random evaluation claims (σ, r, v) for matrices/vectors, the verifier will verify the corresponding openings once at the end of IVC.

Recursive Composition. Every k step, SUMMER invokes the augment function F_A to fold the child steps into one parent. Values are combined by reducing the evaluation to a common point and a random linear combination; commitments are combined by the multivariate commitment aggregation method [1]. The parent also invokes the k child verification in-circuit and verifiers the Fiat-Shamir hash permutations through small sumcheck instances over a MiMC-style [2] hash function. The parent outputs a tiny accumulator that carries a counter, a transcript digest, and one aggregated $(\sigma^*, r_{\text{agg}}, v_{\text{agg}})$. No openings occur within the recursion; all openings are batched once at the root.

Optimized Engineering. In the IVC process, lookup table commitments are cached and their Merkle roots reused across iterations. Commitment encoding and sumcheck instances use parallel loops, while intermediate tensors are released per-iteration to keep memory low. The prover maintains a compact transcript of step proofs and aggregation proofs and records only the minimal advice required by the parent F_A.

Outputs. The output of SUMMER consists of a final commitment to the trained parameters, one recursive proof that certifies to all iterations, and a constant number of batched opening proofs that validate all deferred evaluation claims. Verifier time and proof size are independent of the number of iterations, while the prover time at each step is dominated by the parameters commitment.

Combining all the ingredients discussed above, we have the full description of SUMMER as in Protocol 6.2

Theorem 2. SUMMER *(Protocol 6.2) is a zero-knowledge proof of training for time-series model and satisifes completeness, knowledge soundness, and zero-knowledge.*

Completeness follows directly since each PBPTT.Prove call produces valid sumcheck transcripts for the linear and nonlinear relations, and the aggregation step F_A only folds already verified children and Merkle-certified codewords; by induction the final root proof certifies the correctness of the entire training. For knowledge soundness, if an adversary outputs an accepting proof, then by the extractability of PCS and the knowledge soundness of $\mathsf{Sum}_{\mathsf{MatMul}}$, LogUp, and GKR, one can extract the committed parameters and lookup witnesses for each iteration. The recursive aggregation preserves extractability, so running the extractors iteratively recovers the full training trace, ensuring that no accepting proof can exist without knowledge of a valid BPTT execution. Zero-knowledge is immediate from the hiding property of PCS and Merkle commitments, and from the zero-knowledge simulators of $\mathsf{Sum}_{\mathsf{MatMul}}$, LogUp, and GKR; the aggregation adds only randomized linear combinations, so the entire proof transcript leaks nothing beyond the public commitments. The security proof can be found in Appendix C.

7 Implementation and Evaluation

7.1 Implementation

Framework Setup. We implemented SUMMER mainly in C++. The sumcheck and GKR protocols are built on open-source libraries [16]. We use Orion [31] as our multivariate polynomial commitment scheme due to its linear prover time and polylogarithmic verifier time and proof size. We adopt TaSSLE [5] as our look-up argument, which provides minimal commitment costs. All protocols work on arbitrary finite fields; in our implementation, we instantiate them in the quadratic extension field $\mathbb{F}_{p^2}$ with $p = 2^{61} - 1$ a Mersenne prime. For cryptographic primitives, we use MiMC [2] with 80 rounds and SHA-256 as our hash functions, which together provide a security level of $\lambda = 100$ bits.

All experiments were run on a Linux server with an AMD EPYC 9374F 3.85GHz 32-Core Processor and 1.5TB RAM. We set the quantization parameter to $Q = 32$, which quantized each real number as a signed 32-bit fixed-point integer. After symmetric clipping, the quantized values lie in

Protocol 6.2. SUMMER Zero-knowledge Proof of Training for RNN

Parameters. Let $\mathsf{PCS} = (\mathsf{KeyGen}, \mathsf{Commit}, \mathsf{Open}, \mathsf{Verify})$ be a (non-homomorphic) multivariate polynomial commitment scheme; $\mathsf{Sum}_{\mathsf{MatMul}}$ denote the time–optimal sumcheck subprotocol for matrix multiplication; LogUp denote the logarithmic–derivative lookup argument; GKR be a generic sumcheck–based zk system; $\mathsf{PBPTT} = (\mathsf{Prove}, \mathsf{Verify})$ be the per–iteration step protocol (Protocol B, using $\mathsf{Sum}_{\mathsf{MatMul}} + \mathsf{LogUp}$); FS be a Fiat–Shamir transcript over a sumcheck–friendly hash. Fix recursion arity $k \geq 2$.

Preprocessing. *(i)* $\mathsf{pp} \leftarrow \mathsf{PCS.KeyGen}(1^\lambda)$. *(ii)* Dataset commitment: $\sigma_{\mathcal{D}} \leftarrow \mathsf{DataCom}(\mathcal{D}; r_{\mathcal{D}})$. *(iii)* Initial parameters: $\sigma_{\theta_0} \leftarrow \mathsf{ParCom}(\theta_0; r_{\theta_0})$. *(iv)* One–time commitments for activation lookup tables used by LogUp: $\sigma_{T,\mathsf{act}}, \sigma_{T,\mathsf{Softmax}} \leftarrow \mathsf{ParCom}(T_{\mathsf{act}} \,\|\, T_{\mathsf{Softmax}}; r_T)$. *(v)* Initialize FS with public commitments/metadata; set base proof Π_0 and accumulator $\mathsf{acc}_0 = (\mathsf{ctr} = 0, \mathsf{digest}, \perp)$.

$\big(\sigma_{\theta_i}, \Pi_i\big) \leftarrow \mathsf{Prove}\big(i, \mathsf{pp}, \sigma_{\mathcal{D}}, \sigma_{\theta_0}, \sigma_{\theta_{i-1}}, \theta_{i-1}, \Pi_{i-1}\big)$.

1. Run one BPTT iteration on $(\theta_{i-1}, \mathcal{D})$ to obtain θ_i and the packed step witness wit_i. Commit:
$$\sigma_{\theta_i} \leftarrow \mathsf{ParCom}(\theta_i; r_{\theta_i}), \qquad \sigma_{\mathsf{wit},i} \leftarrow \mathsf{ParCom}(\mathsf{wit}_i; r_{\mathsf{wit},i}).$$
2. Invoke $\mathsf{PBPTT.Prove}$ using $\mathsf{Sum}_{\mathsf{MatMul}}$ for all linear maps and LogUp for $\mathsf{act}/\mathsf{Softmax}$ lookups *against* $(\sigma_{T,\mathsf{act}}, \sigma_{T,\mathsf{Softmax}})$. This yields a sumcheck bundle that exposes only random evaluation claims $\{(\sigma_j, r_j, v_j)\}_j$ for the matrices/vectors that appeared during the checks.
3. When k fresh children are buffered, run the augment function F_A: derive FS challenges, batch–verify the k child step transcripts *in–circuit* via $\mathsf{PBPTT.Verify}$; align value claims to a common point r_{agg} and compress $v_{\mathsf{agg}} = \sum_{j=1}^{k} \lambda_j v_j$; build an aggregated codeword $M^\star = \sum_{j=1}^{k} \lambda_j M_j$ and set $\sigma^\star \leftarrow \mathsf{PCS.Commit}(M^\star)$; certify on a public sample set $\mathcal{S}$ that $M^\star[r,c] = \sum_j \lambda_j M_j[r,c]$ via Merkle paths; verify the hash–permutation rounds that feed FS by small sumchecks; update the accumulator to $\mathsf{acc}_i = (\mathsf{ctr}{+}1, \mathsf{digest}', (\sigma^\star, r_{\mathsf{agg}}, v_{\mathsf{agg}}))$.
4. Assemble
$$\Pi_i \;=\; \big(\Pi_i^{\mathsf{sc}},\, \Pi_i^{\mathsf{agg}},\, \Pi_i^{\mathsf{open}},\, \mathsf{acc}_i\big),$$
where Π_i^{sc} is produced by $\mathsf{PBPTT.Prove}$ (linear layers via $\mathsf{Sum}_{\mathsf{MatMul}}$, nonlinearities via LogUp against $\sigma_{T,\mathsf{act}}, \sigma_{T,\mathsf{Softmax}}$, plus the sumchecks for the FS permutations); Π_i^{agg} is produced by the parent fold F_A over k children (evaluation reduction to r_{agg}, new $\sigma^\star$, Merkle checks on $\mathcal{S}$, and hash–permutation sumchecks); Π_i^{open} is empty for internal nodes and becomes the final batched openings at the root.

$\{0,1\} \leftarrow \mathsf{Verify}\big(i, \mathsf{pp}, \sigma_{\mathcal{D}}, \sigma_{\theta_0}, \sigma_{\theta_i}, \Pi_i\big)$.

1. Parse $\Pi_i = (\Pi_i^{\mathsf{sc}}, \Pi_i^{\mathsf{agg}}, \Pi_i^{\mathsf{open}}, \mathsf{acc}_i)$ and recompute the Fiat–Shamir and Merkle hashes from the public inputs and acc_i.
2. Run $\mathsf{PBPTT.Verify}$ (and any auxiliary $\mathsf{GKR.Verify}$) on Π_i^{sc} with respect to $(\sigma_{\mathcal{D}}, \sigma_{\theta_0}, \sigma_{\theta_i})$ to obtain the transcript–bound commitment–evaluation claims $\{(\sigma_j, x_j, y_j)\}_j$.
3. Verify Π_i^{agg} against $\{(\sigma_j, x_j, y_j)\}_j$ to check the current fold and recover the next aggregated instance encoded in acc_i. If i is the recursion root, additionally verify the batched evaluation openings in Π_i^{open} via $\mathsf{PCS.Verify}$. Accept iff all checks pass.

$\big[-2^{31}, 2^{31}-1\big]$ and are then embedded in $\mathbb{F}_p$. For memory and fair-comparison considerations, we configure the recursion tree with arity $k = 10$ and depth $d = 2$, yield $10^2 = 100$ training iterations. This setting is configurable: It extends naturally to larger recursion parameters, e.g., $k = 20$ and $d = 4$, which correspond to $20^4 = 160{,}000$ iterations.

Scalable Design. Our framework is designed to be scalable and general and supports all the computations required by existing RNN models. In particular, SUMMER handles standard matrix-martix, matrix-vector multiplications, as well as element-wise operations for arbitrary model dimensions. Beyond linear calculations, SUMMER also implements nonlinear activation functions such as ReLU, tanh and Softmax. For each of these operators, we provide both forward evaluation and backward gradient computation, together with the corresponding proof-generation interfaces, enabling users to flexibly compose and verify complete model executions, thereby achieving true end-to-end model verification.

Model Choice. For evaluation, we select the Minimal Character Level Language Model with a Vanilla Recurrent Neural Network (Min-Char-RNN) [13] as our benchmark. This is a classic RNN model consisting of an input layer, a single hidden layer, and an output layer, with activation functions tanh and Softmax. The model is highly scalable: all the dimensions of the layer and the training time steps can be customized. In our experiments, we trained Min-Char-RNN under multiple parameter settings of increasing dimension and time steps (cd. Table 1, Table 2), which allow us to evaluate both the scalability and efficiency of our framework.

7.2 Performance of SUMMER

Performance of PBPTT. We first benchmark the prover time of PBPTT, which represents the cost of generating a complete step proof for a single iteration. We report the average prover time for 100 training iterations. As shown in Table 1, the proof is divided into three stages: *Update*, *Backward*, and *Forward*. We evaluated models ranging from small (50k parameters) to large (12M parameters). The results show that most of the PBPTT prover time comes from the *Forward* stage, mainly because it includes the proof of activation functions via lookup arguments. Notably, even for the largest model with 12M parameters, generating a single iteration proof takes only 11.136 seconds. This shows that SUMMER scales well with model dimension and remains practical even for multi-million parameter RNNs.

Overall Performance. Beyond the per-step analysis, we also evaluate the end-to-end proving pipeline of SUMMER, as reported in Table 2. At the beginning of each iteration, the prover calls the PBPTT prover, which involves generating parameter commitments, witness commitments, and LogUpcommitments. These initialization costs are averaged and reported as *Commitment* time. The PBPTT prover then executes the RNN training computation and produces a step proof, which is reflected in *PBPTT* time. The commitment and proof transcript generated for each step are treated as leaves in the recursion tree. In every k steps, the prover invokes the recursion tree's augment function to aggregate child proofs. This process includes forming a linear combination of evaluation claims and issuing queries, measured as the *Aggregate* cost. The *Prove aggr.* stage corresponds to proving this aggregation, while the *IVC* stage accounts for intermediate hash-based sumchecks and the in-circuit verification of child proofs. All of these aggregation-related costs are amortized over the total number of iterations. Finally, the entire pipeline outputs the total prover time, along with constant-size proof transcripts whose verification time and proof size remain independent of the number of iterations.

From a systems perspective, the overall proving pipeline exhibits remarkable stability. The *Total prover time* remains small even for very large models: for instance, the 12M-parameter RNN completes a full iteration proof in only 75.8 seconds. The intermediate recursion stage, implemented through the augment function (i.e. *Aggregate*, *Prove aggr.*, and *IVC*), contributes only a minor and stable overhead across all model sizes, typically less than 10% of the total prover time. The *Commitment* phase consistently dominates the prover cost. This overhead is inherent and unavoidable, as every iteration must commit to fresh parameters and witness data. Importantly, the verifier time (≈ 20 ms) and proof size (≈ 164 KB) remain constant, independent of the model size and the number of iterations. Finally, memory usage grows with parameter size but under control: even at the largest scale, peak memory reaches about 324GB, which is well within the capacity of modern servers relative to the massive underlying RNN computations. These results confirm that SUMMER achieves scalable proving performance while ensuring succinct and stable verification.

7.3 Comparison with Baseline

We compare SUMMER with Nova [15], where computations must be compiled into R1CS, causing constraints—and thus proving time, setup, folding, verifier cost, and memory—to grow linearly with model size and sequence length. By contrast, SUMMER shows only mild, stable growth in prover time as sequences extend, with per-iteration costs varying only slightly across time steps.

Metric	Time Steps (T)					
	2^4	2^5	2^6	2^7	2^8	2^9
Total Prover (s)						
Nova	11.57	18.01	32.31	60.90	107.71	226.03
SUMMER	18.76	19.09	19.71	20.39	22.20	26.52
Total Verifier (s)						
Nova	2.19	2.17	4.96	7.47	20.13	34.20
SUMMER	0.02	0.02	0.02	0.02	0.02	0.02
Proof size (KB)						
Nova	14.82	14.83	15.19	15.53	15.53	16.24
SUMMER	164.59	164.59	164.59	164.59	164.59	164.59

Table 3: Comparison of Nova and SUMMER (dimension $256 \times 512 \times 256$, model size 525,056).

Table 3 summarizes the detailed comparison between Nova and SUMMER in the same 500k parameter setting. Nova requires a one-time preprocessing step to sample its public parameters, which can be expensive in practice. Beyond this preprocessing, Nova's per-iteration proving time, folding time, verifier time, and peak memory all scale linearly in T. In contrast, SUMMER maintains stable verifier cost and succinct proofs, with only mild growth in prover time. At $T = 512$, SUMMER achieves an increase in $8.5\times$ total prover time and reduces memory usage by $11.6\times$ compared to Nova.

Figures 1 and 2 visualize the comparison: while Nova's prover time and memory usage grow sharply with T, SUMMER remains efficient and memory-stable across all evaluated settings.

Metric	50k		500k		1.5M		3M		12M	
	T=64	T=256	T=64	T=256	T=64	T=256	T=64	T=256	T=64	T=256
Update	0.005	0.005	0.051	0.050	0.154	0.154	0.298	0.298	1.399	1.402
Backward	0.008	0.024	0.046	0.106	0.129	0.253	0.201	0.335	0.841	1.120
Forward	1.679	2.835	2.717	4.540	3.282	4.575	3.908	5.353	5.574	8.614
Total Prover	1.692	2.864	2.814	4.696	3.566	4.982	4.407	5.987	7.815	11.136

Table 1: PBPTT Prover time for Min-Char-RNN (seconds).

Metric	50k		500k		1.5M		3M		12M	
	T=64	T=256	T=64	T=256	T=64	T=256	T=64	T=256	T=64	T=256
Commitment	5.82	6.48	8.44	9.04	11.31	17.69	16.96	29.55	50.82	54.56
PBPTT	1.69	2.86	2.81	4.70	3.57	4.98	4.41	5.99	7.82	11.14
Aggregate	0.13	0.12	0.16	0.16	0.22	0.33	0.33	0.53	0.99	1.01
Prove aggr.	7.81	8.00	8.01	7.99	8.03	8.07	8.05	8.08	7.98	7.96
IVC	0.25	0.24	0.28	0.29	0.34	0.45	0.45	0.66	1.12	1.14
Total Prover	15.71	17.70	19.71	22.17	23.47	31.52	30.19	44.81	68.72	75.80
Total Verifier	0.02	0.02	0.02	0.02	0.02	0.02	0.02	0.02	0.02	0.03
Proof size (KB)	164.59	164.59	164.59	164.59	164.59	164.59	164.59	164.59	164.59	164.59
Peak Mem. (GB)	81.76	84.65	94.51	98.06	108.98	142.19	137.53	201.61	304.86	323.68

Table 2: End-to-end proving statistics for Min-Char-RNN (seconds unless noted).

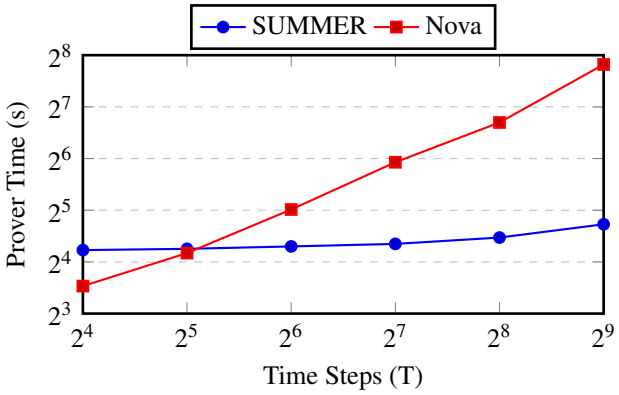

Figure 1: Prover time vs. time steps.

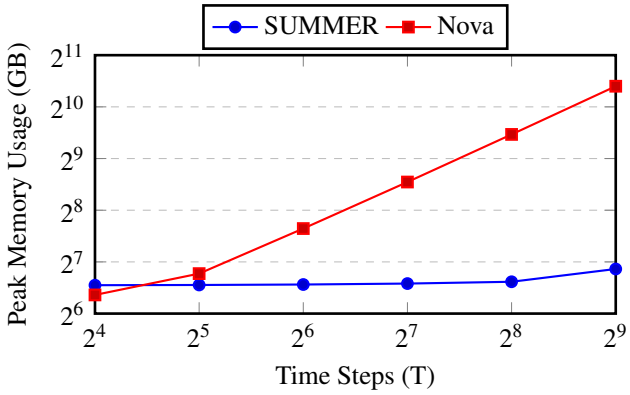

Figure 2: Peak memory usage vs. time steps.

Open Science

We provide the source code of our work in the anonymous repository https://anonymous.4open.science/r/zkRNN-CE8F/.

References

[1] Kasra Abbaszadeh, Christodoulos Pappas, Jonathan Katz, and Dimitrios Papadopoulos. Zero-knowledge proofs of training for deep neural networks. In *Proceedings of the 2024 on ACM SIGSAC Conference on Computer and Communications Security*, pages 4316–4330, 2024.

[2] Martin Albrecht, Lorenzo Grassi, Christian Rechberger, Arnab Roy, and Tyge Tiessen. Mimc: Efficient encryption and cryptographic hashing with minimal multiplicative complexity. In *International Conference on the Theory and Application of Cryptology and Information Security*, pages 191–219. Springer, 2016.

[3] Nir Bitansky, Ran Canetti, Alessandro Chiesa, and Eran Tromer. Recursive composition and bootstrapping for snarks and proof-carrying data. In *Proceedings of the forty-fifth annual ACM symposium on Theory of computing*, pages 111–120, 2013.

[4] Bing-Jyue Chen, Suppakit Waiwitlikhit, Ion Stoica, and Daniel Kang. Zkml: An optimizing system for ml inference in zero-knowledge proofs. In *Proceedings of the Nineteenth European Conference on Computer Systems*, pages 560–574, 2024.

[5] Daniel Dore. Tassle: Lasso for the commitment-phobic. *Cryptology ePrint Archive*, 2024.

[6] John Duchi, Elad Hazan, and Yoram Singer. Adaptive subgradient methods for online learning and stochastic optimization. *Journal of machine learning research*, 12(7), 2011.

[7] Jeffrey L Elman. Finding structure in time. *Cognitive science*, 14(2):179–211, 1990.

[8] Amos Fiat and Adi Shamir. How to prove yourself: Practical solutions to identification and signature problems. In *Conference on the theory and application of cryptographic techniques*, pages 186–194. Springer, 1986.

[9] Sanjam Garg, Aarushi Goel, Somesh Jha, Saeed Mahloujifar, Mohammad Mahmoody, Guru-Vamsi Policharla, and Mingyuan Wang. Experimenting with zero-knowledge proofs of training. In *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security*, pages 1880–1894, 2023.

[10] Shafi Goldwasser, Yael Tauman Kalai, and Guy N Rothblum. Delegating computation: interactive proofs for muggles. *Journal of the ACM (JACM)*, 62(4):1–64, 2015.

[11] Ulrich Haböck. Multivariate lookups based on logarithmic derivatives. *Cryptology ePrint Archive*, 2022.

[12] Geoffrey Hinton. Neural networks for machine learning, lecture 6e: Rmsprop. Coursera video lectures, 2012. URL: http://www.cs.toronto.edu/~tijmen/csc321/slides/lecture_slides_lec6.pdf.

[13] Andrej Karpathy. The unreasonable effectiveness of recurrent neural networks. https://karpathy.github.io/2015/05/21/rnn/, 2015.

[14] Diederik P Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

[15] Abhiram Kothapalli, Srinath Setty, and Ioanna Tzialla. Nova: Recursive zero-knowledge arguments from folding schemes. In *Annual International Cryptology Conference*, pages 359–388. Springer, 2022.

[16] Tianyi Liu, Xiang Xie, and Yupeng Zhang. zkcnn: Zero knowledge proofs for convolutional neural network predictions and accuracy. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*, pages 2968–2985, 2021.

[17] Carsten Lund, Lance Fortnow, Howard Karloff, and Noam Nisan. Algebraic methods for interactive proof systems. *Journal of the ACM (JACM)*, 39(4):859–868, 1992.

[18] Ralph C Merkle. A digital signature based on a conventional encryption function. In *Conference on the theory and application of cryptographic techniques*, pages 369–378. Springer, 1987.

[19] Shahar Papini and Ulrich Haböck. Improving logarithmic derivative lookups using gkr. *Cryptology ePrint Archive*, 2023.

[20] Razvan Pascanu, Tomas Mikolov, and Yoshua Bengio. On the difficulty of training recurrent neural networks. In *Proceedings of the 30th International Conference on Machine Learning (ICML)*, pages 1310–1318, 2013.

[21] Wenjie Qu, Yijun Sun, Xuanming Liu, Tao Lu, Yanpei Guo, Kai Chen, and Jiaheng Zhang. zkgpt: An efficient non-interactive zero-knowledge proof framework for llm inference. In *34st USENIX Security Symposium (USENIX Security 25)*, 2025.

[22] Srinath Setty and Justin Thaler. Twist and shout: Faster memory checking arguments via one-hot addressing and increments. *Cryptology ePrint Archive*, 2025.

[23] Srinath Setty, Justin Thaler, and Riad Wahby. Unlocking the lookup singularity with lasso. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, pages 180–209. Springer, 2024.

[24] Haochen Sun, Tonghe Bai, Jason Li, and Hongyang Zhang. Zkdl: Efficient zero-knowledge proofs of deep learning training. *IEEE Transactions on Information Forensics and Security*, 2024.

[25] Haochen Sun, Jason Li, and Hongyang Zhang. zkllm: Zero knowledge proofs for large language models. In *Proceedings of the 2024 on ACM SIGSAC Conference on Computer and Communications Security*, pages 4405–4419, 2024.

[26] Justin Thaler. Time-optimal interactive proofs for circuit evaluation. In *Annual Cryptology Conference*, pages 71–89. Springer, 2013.

[27] Paul Valiant. Incrementally verifiable computation or proofs of knowledge imply time/space efficiency. In *Theory of Cryptography: Fifth Theory of Cryptography Conference, TCC 2008, New York, USA, March 19-21, 2008. Proceedings 5*, pages 1–18. Springer, 2008.

[28] Chenkai Weng, Kang Yang, Xiang Xie, Jonathan Katz, and Xiao Wang. Mystique: Efficient conversions for {Zero-Knowledge} proofs with applications to machine learning. In *30th USENIX Security Symposium (USENIX Security 21)*, pages 501–518, 2021.

[29] Paul J Werbos. Backpropagation through time: what it does and how to do it. *Proceedings of the IEEE*, 78(10):1550–1560, 2002.

[30] Tiacheng Xie, Jiaheng Zhang, Yupeng Zhang, Charalampos Papamanthou, and Dawn Song. Libra: Succinct zero-knowledge proofs with optimal prover computation. In *Advances in Cryptology–CRYPTO 2019: 39th Annual International Cryptology Conference, Santa Barbara, CA, USA, August 18–22, 2019, Proceedings, Part III 39*, pages 733–764. Springer, 2019.

[31] Tiancheng Xie, Yupeng Zhang, and Dawn Song. Orion: Zero knowledge proof with linear prover time. In *Annual International Cryptology Conference*, pages 299–328. Springer, 2022.

A Additional Preliminaries

In this section we introduce the foundational background used throughout the paper. We first present the full Backpropagation Through Time (BPTT) algorithm. We then give formal definitions of interactive proofs/arguments, polynomial commitment schemes (PCS), and incrementally verifiable computation (IVC). Finally, we provide detailed protocols for the sumcheck and GKR frameworks.

A.1 RNNs and BPTT

The Backpropagation Through Time (BPTT) algorithm is shown in Algorithm 1. Here Act is an element-wise nonlinearity, "$\circ$" denotes the Hadamard product, and we average gradients over the sequence length before the SGD update. With softmax and cross-entropy, the output-layer residual simplifies to $\delta_i^{(t)} = \hat{\mathbf{y}}_i^{(t)} - \mathbf{y}_i^{(t)}$.

Notation and dimensions. We use n for the input size, h for the hidden size, k for the output size, and T for the sequence length (time steps). Vectors are column vectors; biases broadcast along the feature dimension. At training iteration i and time step t,

$$\mathbf{x}_i^{(t)} \in \mathbb{R}^n, \quad \mathbf{h}_i^{(t)} \in \mathbb{R}^h \ (\mathbf{h}_i^{(0)} \text{ given}),$$
$$\mathbf{a}_i^{(t)} \in \mathbb{R}^h, \quad z_i^{(t)} \in \mathbb{R}^k, \quad \hat{\mathbf{y}}_i^{(t)} \in \mathbb{R}^k.$$

Parameters and their shapes are

$$\mathbf{W}_x \in \mathbb{R}^{h\times n}, \quad \mathbf{W}_h \in \mathbb{R}^{h\times h},$$
$$\mathbf{W}_y \in \mathbb{R}^{k\times h}, \quad b_1 \in \mathbb{R}^h, \quad b_2 \in \mathbb{R}^k.$$

In the backward pass we use

$$\delta_i^{(t)} = \hat{\mathbf{y}}_i^{(t)} - \mathbf{y}_i^{(t)} \in \mathbb{R}^k,$$
$$\mathbf{h}_i'^{(t)} \in \mathbb{R}^h, \quad \mathbf{a}_i'^{(t)} \in \mathbb{R}^h.$$

Gradients satisfy

$$\nabla_{\mathbf{W}_y} \in \mathbb{R}^{k\times h}, \quad \nabla_{\mathbf{W}_h} \in \mathbb{R}^{h\times h}, \quad \nabla_{\mathbf{W}_x} \in \mathbb{R}^{h\times n},$$
$$\nabla_{b_1} \in \mathbb{R}^h, \quad \nabla_{b_2} \in \mathbb{R}^k,$$

and we use a scalar learning rate η with averaging over T before the update.

Algorithm 1 Backpropagation Through Time (BPTT)

Input: Weights $\mathbf{W}_{y,i-1}, \mathbf{W}_{h,i-1}, \mathbf{W}_{e,i-1}$; biases $b_{1,i-1}, b_{2,i-1}$; initial hidden state $\mathbf{h}_i^{(0)}$; inputs $\{\mathbf{x}_i^{(t)}\}_{t=1}^T$ and labels $\{\mathbf{y}_i^{(t)}\}_{t=1}^T$
Output: Updated weights $\mathbf{W}_{y,i}, \mathbf{W}_{h,i}, \mathbf{W}_{x,i}$ and $b_{1,i}$, $b_{2,i}$

INITIALIZE:

$\nabla_{\mathbf{W}_{y,i}} \leftarrow 0,\ \nabla_{\mathbf{W}_{h,i}} \leftarrow 0,\ \nabla_{\mathbf{W}_{e,i}} \leftarrow 0,\ \nabla_{b_{1,i}} \leftarrow 0,\ \nabla_{b_{2,i}} \leftarrow 0$
$\mathcal{L} \leftarrow 0$

FORWARD PASS:

for $t = 1$ to T **do**
  $\mathbf{a}_i^{(t)} \leftarrow \mathbf{W}_{h,i-1}\mathbf{h}_i^{(t-1)} + \mathbf{W}_{e,i-1}\mathbf{x}_i^{(t)} + b_{1,i-1}$ ▷ $\mathbf{a}_i^{(t)} \in \mathbb{R}^h$
  $\mathbf{h}_i^{(t)} \leftarrow \mathsf{Act}(\mathbf{a}_i^{(t)})$
  $z_i^{(t)} \leftarrow \mathbf{W}_{y,i-1}\mathbf{h}_i^{(t)} + b_{2,i-1}$
  $\hat{\mathbf{y}}_i^{(t)} \leftarrow \mathsf{softmax}(z_i^{(t)})$
  $\mathcal{L}^{(t)} \leftarrow -\sum_k y_{i,k}^{(t)} \log(\hat{y}_{i,k}^{(t)})$
  $\mathcal{L} \mathrel{+}= \mathcal{L}^{(t)}$
end for

BACKWARD PASS:

for $t = T$ **down to** 1 **do**
  $\delta_i^{(t)} \leftarrow \hat{\mathbf{y}}_i^{(t)} - \mathbf{y}_i^{(t)}$ ▷ $\delta_i^{(t)} \in \mathbb{R}^K$
  $\nabla_{\mathbf{W}_{y,i}} \mathrel{+}= \delta_i^{(t)} (\mathbf{h}_i^{(t)})^\top$
  $\nabla_{b_{2,i}} \mathrel{+}= \delta_i^{(t)}$
  if $t == T$ **then**
    $\mathbf{h}_i'^{(t)} \leftarrow \mathbf{W}_{y,i-1}^\top \delta_i^{(t)}$
  else
    $\mathbf{h}_i'^{(t)} \leftarrow \mathbf{W}_{y,i-1}^\top \delta_i^{(t)} + \mathbf{W}_{h,i-1}^\top \mathbf{a}_i'^{(t+1)}$
  end if
  $\mathbf{a}_i'^{(t)} \leftarrow \mathbf{h}_i'^{(t)} \circ \mathsf{Act}'(\mathbf{a}_i^{(t)})$ ▷ $\mathbf{a}_i'^{(t)} \in \mathbb{R}^h$
  $\nabla_{\mathbf{W}_{h,i}} \mathrel{+}= \mathbf{a}_i'^{(t)} (\mathbf{h}_i^{(t-1)})^\top$
  $\nabla_{\mathbf{W}_{e,i}} \mathrel{+}= \mathbf{a}_i'^{(t)} (\mathbf{x}_i^{(t)})^\top$
  $\nabla_{b_{1,i}} \mathrel{+}= \mathbf{a}_i'^{(t)}$
end for

UPDATE WEIGHTS:

$\mathbf{W}_{y,i} \leftarrow \mathbf{W}_{y,i-1} - \eta \cdot \frac{1}{T} \nabla_{\mathbf{W}_{y,i}}$
$\mathbf{W}_{h,i} \leftarrow \mathbf{W}_{h,i-1} - \eta \cdot \frac{1}{T} \nabla_{\mathbf{W}_{h,i}}$
$\mathbf{W}_{e,i} \leftarrow \mathbf{W}_{e,i-1} - \eta \cdot \frac{1}{T} \nabla_{\mathbf{W}_{e,i}}$
$b_{1,i} \leftarrow b_{1,i-1} - \eta \cdot \frac{1}{T} \nabla_{b_{1,i}}$
$b_{2,i} \leftarrow b_{2,i-1} - \eta \cdot \frac{1}{T} \nabla_{b_{2,i}}$

A.2 Proofs, Arguments and Commitment

Zero-knowledge Arguments of Knowledge. An argument system for an NP relation $\mathcal{R}$ is a protocol that enables a computationally-bounded prover $\mathcal{P}$ to convince a verifier $\mathcal{V}$ that there exists a witness ω such that $(x, \omega) \in \mathcal{R}$ for some input x. If there exists an extractor $\mathcal{E}$ that extracts the witness ω from $\mathcal{P}$'s communication with $\mathcal{V}$, then we say that the argument is a zero-knowledge argument of knowledge. Let $\mathcal{G}$ denote the generator used to generate the public parameter pp.

Definition 3. *Given an* NP *relation* $\mathcal{R}$*. A tuple of algorithm* $(\mathcal{G}, \mathcal{P}, \mathcal{V})$ *is a zero-knowledge argument of knowledge for* $\mathcal{R}$ *if the following conditions hold:*

- ***Correctness.*** *For any* $\mathsf{pp} \leftarrow \mathcal{G}(1^\lambda)$ *and* $(x, \omega) \in \mathcal{R}$,

$$\langle \mathcal{P}(\mathsf{pp}, \omega), \mathcal{V}(\mathsf{pp}) \rangle(x) = 1.$$

- ***Knowledge Soundness.*** *For any PPT adversary* $\mathcal{A}$*, there exists a PPT extractor* $\mathcal{E}$ *such that given the access to* $\mathcal{A}$*'s messages and randomness*

$$\Pr\left[\begin{array}{c} \mathcal{V}(\mathsf{pp}, x, \pi^*) = 1 \wedge \\ (x, \omega) \notin \mathcal{R} \end{array} \middle| \begin{array}{c} \mathsf{pp} \leftarrow \mathcal{G}(1^\lambda), \\ \pi^* \leftarrow \mathcal{A}(x, \mathsf{pp}) \\ \omega \leftarrow \mathcal{E}(\mathsf{pp}, x, \pi^*) \end{array} \right] \leq \mathsf{negl}(\lambda)$$

- ***Zero-knowledge.*** *There exists a PPT simulator* $\mathcal{S}$ *such that for all PPT adversary* $\mathcal{A}$*, auxiliary input* $z \in \{0,1\}^*$*,* $(x, \omega) \in \mathcal{R}$*, and* $\mathsf{pp} \leftarrow \mathcal{G}(1^\lambda)$*, it holds that*

$$\mathsf{View}(\langle \mathcal{P}(\mathsf{pp}, \omega), \mathcal{A}(\mathsf{pp}, z) \rangle(x)) \approx \mathcal{S}^{\mathcal{A}}(x, z)$$

Polynomial Commitment Schemes.

Definition 4 (Polynomial Commitment Schemes.)**.** *Given a finite field* $\mathbb{F}$*, a polynomial commitment scheme* PCS *is a tuple of PPT algorithms* $(\mathsf{KeyGen}, \mathsf{Commit}, \mathsf{Open}, \mathsf{Verify})$ *where:*

- $\mathsf{pp} \leftarrow \mathsf{KeyGen}(1^\lambda)$
- $\sigma \leftarrow \mathsf{Commit}(\mathsf{pp}, f, r)$
- $(y, \pi) \leftarrow \mathsf{Open}(\mathsf{pp}, f, r, x)$
- $\{0,1\} \leftarrow \mathsf{Verify}(\mathsf{pp}, \sigma, \pi, x, y)$

which satisfies the following properties:

- ***Correctness.*** *For any* $x \in \mathbb{F}$*,*

$$\Pr\left[\mathsf{Verify}(\mathsf{pp}, \sigma, \pi, x, y) = 1 \middle| \begin{array}{c} \mathsf{pp} \leftarrow \mathsf{KeyGen}(1^\lambda), \\ \sigma \leftarrow \mathsf{Commit}(\mathsf{pp}, f, r) \\ (y, \pi) \leftarrow \mathsf{Open}(\mathsf{pp}, f, r, x) \end{array} \right] = 1$$

- ***Knowledge Soundness.*** *For any PPT adversary* $\mathcal{A}$*, there exists a PPT extractor* $\mathcal{E}$ *such that given the access to* $\mathcal{A}$*'s messages and randomness, the following probability is* $\mathsf{negl}(\lambda)$

$$\Pr\left[\begin{array}{c} \mathsf{Verify}(\mathsf{pp}, \sigma, \pi, x, y) = 1 \\ \wedge \sigma \leftarrow \mathsf{Commit}(\mathsf{pp}, f, r) \\ \wedge f(x) \neq y \end{array} \middle| \begin{array}{c} \mathsf{pp} \leftarrow \mathsf{KeyGen}(1^\lambda), \\ (x, y) \leftarrow \mathcal{A}(\mathsf{pp}), \\ (\sigma, \pi) \leftarrow \mathcal{A}(\mathsf{pp}), \\ f \leftarrow \mathcal{E}^{\mathcal{A}}(\mathsf{pp}) \end{array} \right]$$

- ***Binding.*** *For any PPT adversary $\mathcal{A}$, the following probability is* $\mathsf{negl}(\lambda)$

$$\Pr\left[\begin{array}{c} f \neq f' \\ \mathsf{Commit}(pp,f,r) = \\ \mathsf{Commit}(pp,f',r') \end{array} \middle| \begin{array}{c} \mathsf{pp} \leftarrow \mathsf{KeyGen}(1^\lambda), \\ (f,r) \leftarrow \mathcal{A}(\mathsf{pp}), \\ (f',r') \leftarrow \mathcal{A}(\mathsf{pp}) \end{array}\right]$$

- *For any* $\mathsf{pp} \leftarrow \mathsf{KeyGen}(1^\lambda)$ *and for all* f, f'*, the following distributions are statistically indistinguishable:*

$$\mathsf{Commit}(\mathsf{pp}, f, r) \approx \mathsf{Commit}(\mathsf{pp}, f', r')$$

Incrementally Verifiable Computation (IVC).

Definition 5 (IVC [15])**.** *An incrementally verifiable computation (IVC) scheme is defined by PPT algorithms $(\mathcal{G}, \mathcal{P}, \mathcal{V})$ and deterministic $\mathcal{K}$ denoting the generator, the prover, the verifier, and the encoder respectively. An IVC scheme $(\mathcal{G}, \mathcal{K}, \mathcal{P}, \mathcal{V})$ satisfies completeness if for any PPT adversary $\mathcal{A}$, the following probability is 1.*

$$\Pr\left[\mathcal{V}(\mathsf{vk}, i, z_0, \pi_i) = 1 \middle| \begin{array}{c} \mathsf{pp} \leftarrow \mathcal{G}(1^\lambda), \\ F(i, z_0, z_i, z_{i-1}, \omega_{i-1}, \pi_{i-1}) \leftarrow \mathcal{A}(\mathsf{pp}), \\ (\mathsf{pk}, \mathsf{vk}) \leftarrow \mathcal{K}(\mathsf{pp}, F), \\ z_i = F(z_{i-1}, \omega_{i-1}), \\ \mathcal{V}(\mathsf{vk}, i-1, z_0, z_{i-1}, \pi_{i-1}) \\ \pi_i \leftarrow \mathcal{P}(\mathsf{pk}, i, z_0, z_i; z_{i-1}, \omega_{i-1}, \pi_{i-1}) \end{array}\right]$$

where F is a polynomial function. An IVC scheme satisfies knowledge-soundness if for any constant $n \in \mathbb{N}$, any PPT adversaries $\mathcal{P}^*$, there exists PPT extractor $\mathcal{E}$, such that for any input randomness ρ, the following probability is $\mathsf{negl}(\lambda)$

$$\Pr\left[\begin{array}{c} z_n \neq z, \\ \mathcal{V}(\mathsf{vk}, n, z_0, z, \pi) = 1 \end{array} \middle| \begin{array}{c} \mathsf{pp} \leftarrow \mathcal{G}(1^\lambda), \\ F(z_0, z, \pi_{i-1}) \leftarrow \mathcal{P}^*(\mathsf{pp}, \rho), \\ (\mathsf{pk}, \mathsf{vk}) \leftarrow \mathcal{K}(\mathsf{pp}, F), \\ (\omega_0, \ldots, \omega_{n-1}) \leftarrow \mathcal{E}(\mathsf{pp}, z_0, z; \rho) \\ z_i \leftarrow F(z_{i-1}, \omega_{i-1}) \forall i \in \{1, \ldots, n\} \end{array}\right]$$

An IVC scheme satisfies succinctness if the size of the IVC proof π does not grow with the number of applications n.

A.3 Sumcheck Protocol

Theorem 3. *[17] Let $\mathbb{F}$ be a finite field and let $g : \mathbb{F}^\ell \to \mathbb{F}$ be an ℓ-variate polynomial with degree at most d. Then Protocol A.3 is an interactive proof for the statement*

$$T = \sum_{x \in \{0,1\}^\ell} g(x),$$

with soundness error at most $O(d\ell/|\mathbb{F}|)$. The protocol uses ℓ rounds. The verifier runs in $O(d\ell)$ field operations plus a single oracle evaluation of g at a random point $g(r_1, \ldots, r_\ell)$. The prover time can be reduced to $O(2^\ell)$ using techniques of [30].

Protocol A.3. Sumcheck Protocol

- Round 1: $\mathcal{P}$ sends a univariate polynomial

$$g_1(x_1) := \sum_{b_2, \ldots, b_\ell \in \{0,1\}} g(x_1, b_2, \ldots, b_\ell).$$

  $\mathcal{V}$ checks $g_1(0) + g_1(1) \stackrel{?}{=} T$ and sends a random challenge $r_1 \in \mathbb{F}$.

- Round i ($2 \leq i \leq \ell - 1$): $\mathcal{P}$ sends

$$g_i(x_i) := \sum_{b_{i+1}, \ldots, b_\ell \in \{0,1\}} g(r_1, \ldots, r_{i-1}, x_i, b_{i+1}, \ldots, b_\ell).$$

  $\mathcal{V}$ checks $g_{i-1}(r_{i-1}) \stackrel{?}{=} g_i(0) + g_i(1)$, then sends a random challenge $r_i \in \mathbb{F}$.

- Final round ($i = \ell$): $\mathcal{P}$ sends

$$g_\ell(x_\ell) := g(r_1, \ldots, r_{\ell-1}, x_\ell),$$

  and $\mathcal{V}$ checks $g_{\ell-1}(r_{\ell-1}) \stackrel{?}{=} g_\ell(0) + g_\ell(1)$. Finally, $\mathcal{V}$ samples $r_\ell \in \mathbb{F}$ and checks

$$g_\ell(r_\ell) \stackrel{?}{=} g(r_1, \ldots, r_\ell),$$

  using oracle access to g. $\mathcal{V}$ accepts if and only if all checks pass.

A.4 GKR Protocol

Theorem 4. *[30] Let $\mathcal{C} \in \mathbb{F}^n \to \mathbb{F}^k$ be a layered arithmetic circuit with d layers. Protocol A.4 is an interactive proof for the function computed by $\mathcal{C}$ with soundness $O(d \log |\mathcal{C}|/|\mathbb{F}|)$. The protocol uses $O(d \log |\mathcal{C}|)$ rounds. The prover time is $O(|\mathcal{C}| \log |\mathcal{C}|)$. For log-space uniform circuits $T = \mathsf{polylog}|\mathcal{C}|$. The verifier time is $O(n + k + d \log |\mathcal{C}| + T)$. The prover time can be reduced to $O(|\mathcal{C}|)$.*

B PBPTT Protocol

In this section, we present our complete PBPTT protocol of RNN in Protocol B.

We formally prove the security of Protocol B.

Proof of Theorem 1. Correctness. The correctness is straightforward by the correctness of GKR, $\mathsf{Sum}_{\mathsf{MatMul}}$, and LogUp protocols.

Knowledge Soundness. Suppose a malicious prover $\mathcal{P}^*$ produces an accepting transcript for iteration i. Consider Stage I, e.g. $\mathbf{W}_y$. If $\mathcal{P}^*$ provides an incorrect value $\mathbf{W}^*_{y,i} \neq \mathbf{W}_{y,i}$, then by the Schwartz–Zippel lemma and the soundness of GKR and $\mathsf{Sum}_{\mathsf{MatMul}}$, this will be detected with high probability unless either: (i) the corresponding input parameter $\mathbf{W}^*_{y,i-1}$ is incorrect, or (ii) the gradient contribution $\nabla^*_{\mathbf{W}_{y,i}}$ is incorrect. Case (ii) reduces to the backward-pass check: if $\nabla^*_{\mathbf{W}_{y,i}} \neq \nabla_{\mathbf{W}_{y,i}}$, then with high probability one of the underlying relations

Protocol A.4. GKR Protocol
Let $\mathbb{F}$ be a finite field. Let $\mathcal{C}: \mathbb{F}^n \to \mathbb{F}^k$ be a layered arithmetic circuit with d layers. A prover $\mathcal{P}$ wants to convince a verifier $\mathcal{V}$ that $\mathsf{out} = C(\mathsf{in})$ where in is the input from $\mathcal{V}$ and out is the corresponding output computed by the circuit. WLOG, assume n and k are both powers of 2.

1. Define the multilinear extension of array out as $\tilde{V}_0$. $\mathcal{V}$ samples a random $g^{(0)} \in \mathbb{F}^{s_0}$ and sent it to $\mathcal{P}$. Both parties compute $\tilde{V}_0(g)$.
2. $\mathcal{P}$ and $\mathcal{V}$ run a sumcheck protocol on
$$\tilde{V}_0(g^{(0)}) = \sum_{x,y\in\{0,1\}^{s_1}} \tilde{\mathsf{mult}}_1(g^{(0)},x,y)\cdot\left(\tilde{V}_1(x)\tilde{V}_1(y)\right) + \tilde{\mathsf{add}}_1(g^{(0)},x,y)\cdot\left(\tilde{V}_1(x)+\tilde{V}_1(y)\right)$$
At the end of the sumcheck, $\mathcal{V}$ receives $\tilde{V}_1(u^{(1)})$ and $\tilde{V}_1(v^{(1)})$. $\mathcal{V}$ computes $\mathsf{mult}_1(g^{(0)},u^{(1)},y^{(1)})$ and $\mathsf{add}_1(g^{(0)},u^{(1)},y^{(1)})$. $\mathcal{V}$ checks that
$$\mathsf{mult}_1(g^{(0)},u^{(1)},v^{(1)})\tilde{V}_1(u^{(1)})\tilde{V}_1(v^{(1)}) + \mathsf{add}_1(g^{(0)},u^{(1)},v^{(1)})\tilde{V}_1(u^{(1)}) + \tilde{V}_1(v^{(1)})$$
equals to the last message of the sumcheck.
3. for $i = 1,\ldots,d-1$:
   - $\mathcal{V}$ randomly samples $\alpha^{(i)},\beta^{(i)} \in \mathbb{F}$ and sends them to $\mathcal{P}$.
   - $\mathcal{P}$ and $\mathcal{V}$ run a sumcheck protocol on the equation:
$$\begin{aligned}\alpha^{(i)}\tilde{V}_i(u^{(i)}) + \beta^{(i)}\tilde{V}_i(v^{(i)}) = \sum_{x,y\in\{0,1\}^{s_{i+1}}} &((\alpha^{(i)}\tilde{\mathsf{mult}}_{i+1}(u^{(i)},x,y) + \beta^{(i)}\tilde{\mathsf{mult}}_{i+1}(v^{(i)},x,y))\cdot\tilde{V}_{i+1}(x)\tilde{V}_{i+1}(y)\\ &+\left(\alpha^{(i)}\tilde{\mathsf{add}}_{i+1}(u^{(i)},x,y) + \beta^{(i)}\tilde{\mathsf{add}}_{i+1}(v^{(i)},x,y)\right)\cdot(\tilde{V}_{i+1}(x)+\tilde{V}_{i+1}(y))\end{aligned}$$
   - At the end of the sumcheck, $\mathcal{P}$ sends to $\mathcal{V}$ $\tilde{V}_{i+1}(u^{(i+1)})$ and $\tilde{V}_{i+1}(v^{(i+1)})$.
   - $\mathcal{V}$ computes $Mult_{i+1}(x) = \tilde{\mathsf{mult}}_{i+1}(x,u^{(i+1)},v^{(i+1)})$ and $Add_{i+1}(x) = \tilde{\mathsf{add}}_{i+1}(x,u^{(i+1)},v^{(i+1)})$. $\mathcal{V}$ checks that if the following equals to the last message of the sum check
$$\begin{aligned}&\left(\alpha^{(i)}\mathsf{Mult}_{i+1}(u^{(i)}) + \beta^{(i)}\mathsf{Mult}_{i+1}(v^{(i)})\right)\cdot\tilde{V}_{i+1}(u^{(i+1)})\tilde{V}_{i+1}(v^{(i+1)})\\ &+\left(\alpha^{(i)}\mathsf{Add}_{i+1}(u^{(i)}) + \beta^{(i)}\mathsf{Add}_{i+1}(v^{(i)})\right)\cdot\left(\tilde{V}_{i+1}(u^{(i+1)}) + \tilde{V}_{i+1}(v^{(i+1)})\right)\end{aligned}$$
   - If the check passes, $\mathcal{V}$ uses $\tilde{V}_{i+1}(u^{(i+1)})$ and $\tilde{V}_{i+1}(v^{(i+1)})$ as new claims and proceeds to the next layer. Otherwise, $\mathcal{V}$ aborts and rejects.
4. At the input layer d, $\mathcal{V}$ has two claims: $\tilde{V}_d(u^{(d)})$ and $\tilde{V}_d(v^{(d)})$. $\mathcal{V}$ queries the oracle of evaluations of $\tilde{V}_d$ at both $u^{(d)}$ and $v^{(d)}$ and checks whether the evaluations match the claimed values. If both checks succeed, $\mathcal{V}$ accepts; otherwise, it rejects.

(e.g. $\delta^{(t)}(\mathbf{h}^{(t)})^\top$) will fail a $\mathsf{Sum}_{\mathsf{MatMul}}$ verification. Similarly, checking $\delta^{(t)} = \hat{\mathbf{y}}^{(t)} - \mathbf{y}^{(t)}$ guarantees that any incorrect value in $\delta^{(t)}$ propagates to a violation with respect to the committed lookup witnesses or parameters. By induction across stages, any incorrect value in intermediate traces (hidden states, activations, gradients) reduces to an inconsistency in the committed parameters (θ_{i-1},θ_i) or in the lookup witnesses. At the end of Stage IV, all random evaluation claims are reduced to a constant number of PCS openings, whose binding property ensures uniqueness. Therefore, an extractor can invoke the extractor of PCS to recover the unique committed parameters and lookup witnesses, establishing knowledge soundness.

Zero-knowledge. The zero-knowledge property follows by invoking the simulators of each components to produce commitments, evaluation claims, and transcript that are indistinguishable from a real execution. □

C Construction of SUMMER

We formally prove the security of Protocol 6.2.

Proof of Theorem 2. Correctness. Suppose $\mathcal{P}$ honestly executes iteration i with inputs $(\theta_{i-1},\mathcal{D})$. By construction, PBPTT.Prove produces valid transcripts for all linear relations via $\mathsf{Sum}_{\mathsf{MatMul}}$ and nonlinear relations via LogUp, and returns the correct updated parameters θ_i. Since the child proofs Π_{i-1} are already accepted, the augment function F_A folds only valid instances and Merkle-certified codewords. By induction, the verifier $\mathcal{V}$ will accept every Π_i, and at the recursion root, PCS.Verify ensures that the committed inputs/outputs are consistent. Thus, every honestly generated proof is accepted with probability 1.

Knowledge Soundness. We construct an extractor $\mathcal{E}$ in the Fiat-Shamir random oracle model.

By invoking the PCS extractor with Π_i^{open}, $\mathcal{E}$ extracts the opened values for the constant number of batched commitments at the root. By binding of PCS, these are uniquely determined. From Π_i^{agg},$\mathcal{E}$ obtains the k child step transcripts that were batch verified *in-circuit* and the linear-combination coefficients $\{\lambda_j\}$ derived from the Fiat–Shamir transcript, and verifies that the parent's value/evaluation claims are the linear combination of children and that the aggregated codeword matches Merkle samples on a public random set. By the Merkle sampling check and Schwartz–Zippel lemma, any incorrect equality is rejected with high probability; thus $\mathcal{E}$ obtains the next accumulator and the k child instances. For each child step, $\mathcal{E}$ invokes the extractor of PBPTT to obtain the per-iteration committed objects: the parameter commitments $(\sigma_{\theta_{i-1}}, \sigma_{\theta_i})$ and the per-iteration lookup witness commitment $\sigma_{Q,i}$. Any incorrect parameter, gradient, or intermediate value is already detected with high probability by PBPTT's soundness. Applying (ii)–(iii) recursively over the recursion tree reaches all leaves. The root's constant number of PCS openings from (i) binds the concrete values needed to certify all transcript-bound claims. By binding and extractability of PCS, the extracted parameters and lookup witnesses are unique and consistent.

Zero-knowledge. The zero-knowledge property follows from each component: PCS (and Merkle) are hiding; PBPTT admits a simulator; the augment function F_A adds only randomized linear combinations and simulatable Merkle checks; the root invokes the PCS simulator for batched openings. The simulated transcript is indistinguishable from a real proof. □

Protocol B. Proof of Gradient Descent of RNN Protocol

Parameters: Let GKR be a generic GKR-style zero-knowledge proof system, $\mathsf{Sum}_{\mathsf{MatMul}}$ be the time-optimal sumcheck protocol for matrix multiplication [26]. Let LogUp be a logarithmic-derivative TaSSLE-style lookup argument. Throughout the protocol, every linear relation listed under *Linear Operations (Section 3.2)* (matrix–matrix/vector products, Hadamard product, and summations) is reduced to a single matrix multiplication instance and proved by invoking $\mathsf{Sum}_{\mathsf{MatMul}}$. Let $\mathsf{PCS} = (\mathsf{KeyGen}, \mathsf{Commit}, \mathsf{Open}, \mathsf{Verify})$ be the multivariate polynomial commitment scheme with public parameter pp. Let the training window have T time steps.

Initialization:

1. On input sequence $\{\mathbf{x}^{(t)}\}_{t=1}^{T}$ with labels $\{\mathbf{y}^{(t)}\}_{t=1}^{T}$, initial state $\mathbf{h}^{(0)}$, and parameters $(\mathbf{W}_{x,i-1}, \mathbf{W}_{h,i-1}, \mathbf{W}_{y,i-1}, \mathbf{b}_{1,i-1}, \mathbf{b}_{2,i-1})$, the prover $\mathcal{P}$ runs BPTT (Algorithm 1) to obtain forward traces $\{\mathbf{a}^{(t)}, \mathbf{h}^{(t)}, \mathbf{z}^{(t)}, \hat{\mathbf{y}}^{(t)}\}$, backward traces $\{\delta^{(t)}, \mathbf{h}'^{(t)}, \mathbf{a}'^{(t)}\}$, the gradients $(\nabla_{\mathbf{W}_{x,i}}, \nabla_{\mathbf{W}_{h,i}}, \nabla_{\mathbf{W}_{y,i}}, \nabla_{\mathbf{b}_{1,i}}, \nabla_{\mathbf{b}_{2,i}})$, and updated parameters $(\mathbf{W}_{x,i}, \mathbf{W}_{h,i}, \mathbf{W}_{y,i}, \mathbf{b}_{1,i}, \mathbf{b}_{2,i})$.
2. $\mathcal{P}$ concatenates the initial parameters and the updated parameters, and runs $\sigma_{\theta,i-1} \leftarrow \mathsf{PCS.Commit}\big(\mathsf{concat}(\mathbf{W}_{x,i-1}, \widetilde{\mathbf{W}_{h,i-1}}, \mathbf{W}_{y,i-1}, \mathbf{b}_{1,i-1}, \mathbf{b}_{2,i-1})\big)$, $\sigma_{\theta,i} \leftarrow \mathsf{PCS.Commit}\big(\mathsf{concat}(\mathbf{W}_{x,i}, \widetilde{\mathbf{W}_{h,i}}, \mathbf{W}_{y,i}, \mathbf{b}_{1,i}, \mathbf{b}_{2,i})\big)$. $\mathcal{P}$ sends $\sigma_{\theta,i-1}, \sigma_{\theta,i}$ to the verifier $\mathcal{V}$.
3. $\mathcal{P}$ runs $\sigma_{T,\mathsf{act}} \leftarrow \mathsf{PCS.Commit}(\widetilde{T_{\mathsf{act}}}), \sigma_{T,\mathsf{sm}} \leftarrow \mathsf{PCS.Commit}(\widetilde{T_{\mathsf{sm}}})$ *once* (in the first iteration) to the precomputed lookup tables and runs $\sigma_{Q,i} \leftarrow \mathsf{PCS.Commit}(\widetilde{Q_i})$ to the per-round lookup queries (indices and any auxiliary values) packed as a vector Q_i. $\mathcal{P}$ sends $\sigma_{T,\mathsf{act}}, \sigma_{T,\mathsf{sm}}, \sigma_{Q,i}$ to $\mathcal{V}$.

Stage I: Proof of Update.

1. $\mathcal{V}$ samples r_{out} and sends it to $\mathcal{P}$.
2. $\mathcal{P}$ evaluates $\widetilde{\mathbf{W}}_{y,i}(r_{\mathsf{out}})$, $\widetilde{\mathbf{W}}_{h,i}(r_{\mathsf{out}})$, $\widetilde{\mathbf{W}}_{x,i}(r_{\mathsf{out}})$, $\widetilde{\mathbf{b}}_{1,i}(r_{\mathsf{out}})$, $\widetilde{\mathbf{b}}_{2,i}(r_{\mathsf{out}})$ and sends these values to $\mathcal{V}$.
3. $\mathcal{V}$ and $\mathcal{P}$ run GKR on:

$$\mathbf{W}_{y,i} = \mathbf{W}_{y,i-1} - \eta \cdot \nabla_{\mathbf{W}_{y,i}} \quad \mathbf{W}_{h,i} = \mathbf{W}_{h,i-1} - \eta \cdot \nabla_{\mathbf{W}_{h,i}} \quad \mathbf{W}_{x,i} = \mathbf{W}_{x,i-1} - \eta \cdot \nabla_{\mathbf{W}_{x,i}}$$
$$\mathbf{b}_{1,i} = \mathbf{b}_{1,i-1} - \eta \cdot \nabla_{\mathbf{b}_{1,i}} \quad \mathbf{b}_{2,i} = \mathbf{b}_{2,i-1} - \eta \cdot \nabla_{\mathbf{b}_{2,i}}$$

   At the end of this protocol, $\mathcal{V}$ receives the evaluations of $\big(\widetilde{\mathbf{W}}_{*,i-1}(r_{1,\mathbf{W}_*}), \widetilde{\nabla_{\mathbf{W}_{*,i}}}(r_{1,\nabla_{\mathbf{W}_*}})\big)$ and $\big(\widetilde{\mathbf{b}}_{*,i-1}(r_{1,\mathbf{b}_*}), \widetilde{\nabla_{\mathbf{b}_{*,i}}}(r_{1,\nabla_{\mathbf{b}_*}})\big)$.

Stage II: Proof of Backward Pass.

1. With $\widetilde{\nabla_{\mathbf{W}_{x,i}}}(r_{1,\nabla_{\mathbf{W}_{x,i}}})$ from Stage I, $\mathcal{V}$ and $\mathcal{P}$ run $\mathsf{Sum}_{\mathsf{MatMul}}$ on

$$\nabla_{\mathbf{W}_{x,i}} = \frac{1}{T}\sum_{t=1}^{T} \mathbf{a}'^{(t)} \cdot (\mathbf{x}^{(t)})^{\top}$$

   At the end of the protocol, $\mathcal{V}$ receives evaluations of $\widetilde{\mathbf{a}'^{(t)}}(r^{(1)}_{2,a'^{(t)}})$ and $\widetilde{\mathbf{x}^{(t)}}(r_{2,x^{(t)}})$ for all $t \in [1,T]$.
2. With $\widetilde{\nabla_{\mathbf{W}_{h,i}}}(r_{1,\nabla_{\mathbf{W}_{h,i}}})$ from Stage I, $\mathcal{V}$ and $\mathcal{P}$ run $\mathsf{Sum}_{\mathsf{MatMul}}$ on

$$\nabla_{\mathbf{W}_{h,i}} = \frac{1}{T}\sum_{t=1}^{T} \mathbf{a}'^{(t)} \cdot (\mathbf{h}^{(t-1)})^{\top}$$

   At the end of the protocol, $\mathcal{V}$ receives evaluations of $\widetilde{\mathbf{a}'^{(t)}}(r^{(2)}_{2,a'^{(t)}})$ and $\widetilde{\mathbf{h}^{(t-1)}}(r^{(1)}_{2,h^{(t-1)}})$ for all $t \in [1,T]$.
3. With $\widetilde{\nabla_{\mathbf{b}_{1,i}}}(r_{1,\nabla_{\mathbf{b}_{1,i}}})$ from Stage I, $\mathcal{V}$ and $\mathcal{P}$ run $\mathsf{Sum}_{\mathsf{MatMul}}$ on

$$\nabla_{\mathbf{b}_{1,i}} = \frac{1}{T}\sum_{t=1}^{T} \mathbf{a}_i'^{(t)}$$

   At the end of the protocol, $\mathcal{V}$ receives evaluations of $\widetilde{\mathbf{a}'^{(t)}}(r^{(3)}_{2,a'^{(t)}})$ for all $t \in [1,T]$.
4. $\mathcal{V}$ and $\mathcal{P}$ combine evaluations $\widetilde{\mathbf{a}'^{(t)}}(r^{(1)}_{2,a'^{(t)}})$, $\widetilde{\mathbf{a}'^{(t)}}(r^{(2)}_{2,a'^{(t)}})$, and $\widetilde{\mathbf{a}'^{(t)}}(r^{(3)}_{2,a'^{(t)}})$ for all $t \in [1,T]$ by running an evaluation reduction sumcheck. Have the combination, along with the LogUp proof against the committed Act table, $\mathcal{V}$ and $\mathcal{P}$ prove

$$\mathbf{a}'^{(t)} \;=\; \mathbf{h}'^{(t)} \circ \mathsf{Act}'(\mathbf{a}^{(t)}) \quad (\text{e.g., } \mathsf{tanh}' : 1 - \mathbf{h}^{(t)} \circ \mathbf{h}^{(t)}).$$

   At the end of the protocol, $\mathcal{V}$ receives evaluations of $\widetilde{\mathbf{h}'^{(t)}}(r^{(2)}_{2,h'^{(t)}})$ for all $t \in [1,T]$.
5. With $\widetilde{\mathbf{h}'^{(t)}}(r^{(2)}_{2,h'^{(t)}})$, for $t = T$, $\mathcal{V}$ and $\mathcal{P}$ run $\mathsf{Sum}_{\mathsf{MatMul}}$ on

$$\mathbf{h}'^{(t)} = \mathbf{W}_{y,i-1}^{\top} \cdot \delta^{(t)}$$

   At the end of the protocol, $\mathcal{V}$ receives evaluations of $\widetilde{\mathbf{W}_{y,i-1}}(r_{2,W_{y,i-1}})$ and $\widetilde{\delta^{(t)}}(r^{(1)}_{2,\delta^{(t)}})$ for all $t \in [1,T]$.

   For $t \neq T$, $\mathcal{V}$ and $\mathcal{P}$ run $\mathsf{Sum}_{\mathsf{MatMul}}$ on

$$\mathbf{h}'^{(t)} = \mathbf{W}_{y,i-1}^{\top} \cdot \delta^{(t)} + \mathbf{W}_{h,i-1}^{\top} \cdot \mathbf{a}'^{(t+1)}$$

   At the end of the protocol, $\mathcal{V}$ receives evaluations of $\widetilde{\mathbf{W}_{y,i-1}}(r_{2,\mathbf{W}_{y,i-1}})$, $\widetilde{\mathbf{W}_{h,i-1}}(r_{2,\mathbf{W}_{h,i-1}})$, $\widetilde{\mathbf{a}'^{(t+1)}}(r^{(4)}_{2,a'^{(t+1)}})$, and $\widetilde{\delta^{(t)}}(r^{(1)}_{2,\delta^{(t)}})$ for all $t \in [1,T]$.

6. With $\widetilde{\nabla_{\mathbf{b}_{2,i}}}(r_{1,\nabla_{\mathbf{b}_{2,i}}})$ from Stage I, $\mathcal{V}$ and $\mathcal{P}$ run $\mathsf{Sum}_{\mathsf{MatMul}}$ on

$$\nabla_{\mathbf{b}_{2,i}} = \frac{1}{T}\sum_{t=1}^{T}\delta^{(t)}$$

At the end of the protocol, $\mathcal{V}$ receives evaluations of $\widetilde{\delta^{(t)}}(r^{(2)}_{2,\delta^{(t)}})$ for all $t \in [1,T]$.

7. With $\widetilde{\nabla_{\mathbf{W}_{y,i}}}(r_{1,\nabla_{\mathbf{W}_{y,i}}})$ from Stage I, $\mathcal{V}$ and $\mathcal{P}$ run $\mathsf{Sum}_{\mathsf{MatMul}}$ on

$$\nabla_{\mathbf{W}_{y,i}} = \frac{1}{T}\sum_{t=1}^{T}\delta^{(t)}\cdot(\mathbf{h}^{(t)})^{\top}$$

At the end of the protocol, $\mathcal{V}$ receives evaluations of $\widetilde{\delta^{(t)}}(r^{(3)}_{2,\delta^{(t)}})$ and $\widetilde{\mathbf{h}^{(t)}}(r^{(2)}_{2,h^{(t)}})$ for all $t \in [1,T]$.

8. $\mathcal{V}$ and $\mathcal{P}$ combine evaluations $\widetilde{\delta^{(t)}}(r^{(1)}_{2,\delta^{(t)}})$,$\widetilde{\delta^{(t)}}(r^{(2)}_{2,\delta^{(t)}})$, and $\widetilde{\delta^{(t)}}(r^{(3)}_{2,\delta^{(t)}})$ for all $t \in [1,T]$ by running an evaluation reduction sumcheck. Having the combination, $\mathcal{V}$ and $\mathcal{P}$ run GKR on

$$\delta_i^{(t)} = \hat{\mathbf{y}}^{(t)} - \mathbf{y}^{(t)}$$

At the end of the protocol, $\mathcal{V}$ receives evaluations of $\widetilde{\hat{\mathbf{y}}^{(t)}}(r_{2,\hat{y}^{(t)}})$ and $\widetilde{\mathbf{y}^{(t)}}(r_{2,y^{(t)}})$ for all $t \in [1,T]$.

Stage III: Proof of Forward Pass.

1. With $\widetilde{\hat{\mathbf{y}}^{(t)}}(r_{2,\hat{y}^{(t)}})$ from Stage II, along with the LogUp proof against the committed $\mathsf{softmax}$ table, $\mathcal{V}$ and $\mathcal{P}$ prove

$$\hat{\mathbf{y}}^{(t)} = \mathsf{Softmax}(\mathbf{z}^{(t)})$$

At the end of the protocol, $\mathcal{V}$ receives evaluations of $\widetilde{\mathbf{z}^{(t)}}(r_{3,z^{(t)}})$ for all $t \in [1,T]$.

2. With $\widetilde{\mathbf{z}^{(t)}}(r_{3,z_i^{(t)}})$, $\mathcal{V}$ and $\mathcal{P}$ run $\mathsf{Sum}_{\mathsf{MatMul}}$ on

$$\mathbf{z}^{(t)} \leftarrow \mathbf{W}_{y,i-1}\cdot\mathbf{h}^{(t)} + \mathbf{b}_{2,i-1}$$

At the end of the protocol, $\mathcal{V}$ receives evaluations of $\widetilde{\mathbf{W}_{y,i-1}}(r_{3,\mathbf{W}_{y,i-1}})$, $\widetilde{\mathbf{h}^{(t)}}(r_{3,h^{(t)}})$ for all $t \in [1,T]$, and $\widetilde{\mathbf{b}_{2,i-1}}(r_{3,\mathbf{b}_{2,i-1}})$.

3. With $\widetilde{\mathbf{h}^{(t)}}(r_{3,h^{(t)}})$, along with the LogUp proof against the committed Act table, $\mathcal{V}$ and $\mathcal{P}$ prove

$$\mathbf{h}^{(t)} = \mathsf{Act}(\mathbf{a}^{(t)})$$

At the end of the protocol, $\mathcal{V}$ receives evaluations of $\widetilde{\mathbf{a}^{(t)}}(r_{3,a^{(t)}})$ for all $t \in [1,T]$.

4. With $\widetilde{\mathbf{a}^{(t)}}(r_{3,a^{(t)}})$, $\mathcal{V}$ and $\mathcal{P}$ run $\mathsf{Sum}_{\mathsf{MatMul}}$ on

$$\mathbf{a}^{(t)} = \mathbf{W}_{h,i-1}\cdot\mathbf{h}^{(t-1)} + \mathbf{W}_{x,i-1}\cdot\mathbf{x}^{(t)} + \mathbf{b}_{1,i-1}$$

At the end of the protocol, $\mathcal{V}$ receives evaluations of $\widetilde{\mathbf{W}_{h,i-1}}(r_{3,\mathbf{W}_{h,i-1}})$, $\widetilde{\mathbf{W}_{x,i-1}}(r_{3,\mathbf{W}_{x,i-1}})$, $\widetilde{\mathbf{b}_{1,i-1}}(r_{3,\mathbf{b}_{1,i-1}})$, $\widetilde{\mathbf{h}^{(t-1)}}(r'_{3,h^{(t-1)}})$, $\widetilde{\mathbf{x}^{(t)}}(r_{3,x^{(t)}})$ for all $t \in [1,T]$.

Stage IV: Proof of evaluation reduction.

1. $\mathcal{V}$ and $\mathcal{P}$ combine evaluations of form $\widetilde{\mathbf{W}_{*,i-1}}(r_{1,\mathbf{W}_{*,i-1}})$, $\widetilde{\mathbf{W}_{*,i-1}}(r_{2,\mathbf{W}_{*,i-1}})$, $\widetilde{\mathbf{W}_{*,i-1}}(r_{3,\mathbf{W}_{*,i-1}})$ into a single evaluation $\widetilde{\mathbf{W}_{*,i-1}}(r_{\mathsf{in},\mathbf{W}_{*,i-1}})$, evaluations of form $\widetilde{\mathbf{b}_{*,i-1}}(r_{1,\mathbf{b}_{*,i-1}})$, $\widetilde{\mathbf{b}_{*,i-1}}(r_{2,\mathbf{b}_{*,i-1}})$, $\widetilde{\mathbf{b}_{*,i-1}}(r_{3,\mathbf{b}_{*,i-1}})$ into a single evaluation $\widetilde{\mathbf{b}_{*,i-1}}(r_{\mathsf{in},\mathbf{b}_{*,i-1}})$, evaluations of form $\widetilde{\mathbf{x}^{(t)}}(r_{2,x^{(t)}})$ and $\widetilde{\mathbf{x}^{(t)}}(r_{3,x^{(t)}})$ for all $t \in [1,T]$ into a single evaluation $\widetilde{\mathbf{x}}(r_{\mathsf{in},x})$, evaluations of $\widetilde{\mathbf{y}^{(t)}}(r_{2,y^{(t)}})$ for all $t \in [1,T]$ into a single evaluations $\widetilde{\mathbf{y}}(r_{\mathsf{in},y})$.
2. The combined evaluations can be verified by using the evaluation opening algorithm of commitment scheme $\mathsf{PCS.Open}(\mathbf{W}_{*,i-1})$, $\mathsf{PCS.Open}(\mathbf{W}_{*,i})$, $\mathsf{PCS.Open}(\mathbf{b}_{*,i-1})$,$\mathsf{PCS.Open}(\mathbf{b}_{*,i})$, $\mathsf{PCS.Open}(\mathbf{x})$, $\mathsf{PCS.Open}(\mathbf{y})$.