An Efficient and Accurate Gated RNN for Execution under TFHE

Rickard Brännvall[1], **Andrei Stoian**[2]

[1]RISE, Research Institutes of Sweden, Luleå, Sweden
[2]Zama, Machine Learning Group, Paris, France
rickard.brannvall@ri.se, andrei.stoian@zama.ai

Abstract

We implement and test the recently proposed Inhibitor gate for Recurrent Neural Networks (RNNs) that is both efficient and accurate under Homomorphic Encryption. Gated RNNs such as LSTM and GRU have applications in numerous real-world use cases for sequential data, such as time-series analysis and natural language processing, due to their ability to capture long-term dependencies. Conventionally, the update based on current inputs and the previous state history is each multiplied with the dynamic gate values and combined to compute the next state. However, as it is a multiplication between two variables that depend on the input, it can be computationally expensive, especially under homomorphic encryption, where it involves a multiplication between two cipher text variables. Therefore, the novel gating mechanism replaces the multiplication and sigmoid function of the conventional RNN with addition and ReLU activation. Numerical experiments on three synthetic benchmark tests demonstrate that our algorithm outperforms a conventional gated RNN showing at least an order of magnitude faster execution already at 4-bit quantization. At a fixed computational budget, we get 7-bit precision with the novel gate that executes faster than the 4-bit conventional gate.

The gating mechanism employed in this paper may enable privacy-preserving AI applications based on gated RNNs operating under homomorphic encryption by avoiding the multiplication of encrypted variables.

Introduction

Recurrent Neural Networks (RNNs) have gained widespread adoption for learning from sequential data due to their ability to capture temporal dependencies. Although they today are superceeded by Transformers (Vaswani et al. 2017) for SotA performance on many tasks, they remain an important arcitecture, for example, in resource constrained settings (Agarwal and Alam 2020).

Simple RNNs are challenged by gradient-related problems, specifically exploding gradients and vanishing gradients. Exploding gradients leads to unstable training processes, but it can be solved by gradient clipping or norm penalties (Mikolov 2012; Pascanu, Mikolov, and Bengio 2013). On the other hand, vanishing gradients occur when the gradients of recurrent weights become extremely small during backpropagation, impeding the learning of long-term dependencies (Hochreiter 1991; Bengio, Simard, and Frasconi 1994; Hochreiter et al. 2009). To address this issue, gated mechanisms such as Long Short-Term Memory (LSTM) and Gated Recurrent Unit (GRU), have been introduced (Hochreiter and Schmidhuber 1997; Chung et al. 2014). Gated RNNs have been used successfully across various domains, including natural language processing (Bahdanau, Cho, and Bengio 2015), handwriting recognition (Graves et al. 2009), audio- and video processing, and other time series analysis.

The success of gated RNNs lies in their ability to selectively update and retain information over time. For the purpose of illustration, we take the set-up for the GRU (Chung et al. 2014) which in each time-step forms a weighted combinations of the previous state's summarized history, h_{t-1}, and an update proposal, $\hat{h}_t = \phi_h(h_{t-1}, x_t)$, that is a function, ϕ_h, of the previous state and the current input, x_t.

$$z_t = \sigma(W_z x_t + U_z h_{t-1} + b_z) \tag{1}$$

$$h_t = z_t \odot h_{t-1} + (1 - z_t) \odot \hat{h}_t \tag{2}$$

Here W_z, U_z, and b_z are weight matrices and bias, and the symbol $\odot$ denotes the element-wise (Hadamard) multiplication of two vector-valued variables, the previous state and a sigmoid term, $z_t = \sigma(h_{t-1}, x_t)$.

Note that multiplication plays two distinct roles in gated RNNs: Firstly, as in equation 1, matrix multiplication is utilized to weigh the input and state information at each time step, enabling the calculation of gate values. These weights are learned during training but are fixed and known in advance at inference. Secondly, as in equation 2, element-wise multiplication regulates the flow of information within the network. The gates in LSTMs and GRUs employ sigmoid functions to produce gate values that range between 0 and 1. These gate values are multiplied element-wise with the precious state and proposed update information, determining their relative influence. Notably, a gate value of 0 effectively suppresses certain information, while a value of 1 allows it to pass through entirely.

While element-wise multiplication is crucial for controlling information flow, it can be computationally more challenging than applying weights, which only multiply a literal matrix with a variable. The multiplication in applying

the gates instead depends on two variables, both of which rely on input values and are thus undetermined. The computers used for much of modern machine learning activities are equipped with hardware accelerators, such as Graphics Processing Units (GPUs), which have been optimized to efficiently perform floating point multiplication at scale. Even so, the computational complexities of gated RNNs can, however, be important to consider, especially on specific hardware architectures (Parhami 2010) or alternative arithmetic systems like homomorphic encryption (Gentry 2010) that facilitate computation on encrypted data.

Homomorphic encryption introduces a steep cost difference between addition and multiplication operations. In fully homomorphic encryption (FHE) schemes like TFHE (Chillotti et al. 2016), adding two encrypted variables can be efficiently executed with minimal computational overhead. For multiplication, it becomes important to differentiate between literals and variables, where literal multiplication can be optimized using techniques like repeated addition or bit-shifting, resulting in faster execution and lower cost compared to encrypted variable multiplication. In TFHE, the multiplication of two encrypted variables cannot be directly performed and requires two Programmable Bootstrap (PBS) operations, which is an operation that distinguished TFHE (Chillotti et al. 2019) since it allows the simultaneous noise reduction and evaluation of non-linear functions. It, however, significantly increases computational costs and can potentially dominate overall expenses. Also, for activation functions, evaluating non-linear functions like sigmoid and tanh requires the Programmable Bootstrap operation in TFHE and incurs substantial computational overhead. Understanding these complexities is vital for optimizing the efficiency of FHE in practical applications.

Quantization and the use of integer arithmetics can reduce the memory footprint and computational complexity of deep neural networks (Polino, Pascanu, and Alistarh 2018). The application of such techniques may become necessary for execution under FHE (Bourse et al. 2018) that only supports limited precision integers and face higher computational constraints.

Contributions. Motivated by these observations, this paper examines the recently proposed (Brännvall et al. 2023) Inhibitor gating mechanism that is based on addition and the ReLU activation function in place of element-wise multiplication. Our contribution is to examine the use of this novel gate for non-interactive privacy-preserving applications of gated RNNs under TFHE through numerical experiments, which show that it is both accurate and efficient compared to the conventional gate. The novel gated RNN is more economical in the sense that it requires fewer bit circuits at a given precision compared to conventional gated RNNs. Table 2 shows, for example, that the novel gate is at least 200 times faster at 4-bit precision (for input and state) compared to the conventional gate (at 4-bit quantization of the sigmoid function). Alternatively, we see that 7-bit precision with the novel gate executes faster than the 4-bit conventional gate.

Limitations. Computational efficiency is examined in experiments running the gated RNNs for inference under encryption. Only three synthetic tasks executed under TFHE as implemented in the Concrete Python (Zama 2022) are considered. A more comprehensive numerical study should also consider more realistic real-world tasks, which is something that we intend to return to in future work.

The numerical experiments are designed to capture the execution time of the algorithms under encrypted inference; but other aspects of computational efficiency, such as power consumption, are not explicitly measured.

Model training under FHE is not assessed in this work. However, the training capacity of Inhibitor gated RNNs on clear text data was investigated in our previous work (Brännvall et al. 2023).

In parallel work we have proposed the Inhibitor mechanism also for Transformers (Brännvall 2024). We investigate the numerical performance of this construct under FHE in a forthcoming publication (Brännvall and Stoian 2024).

Paper Outline. The next section discusses some preliminaries to this work, including the notations. This is followed by the Method section which describes the main algorithm in. Performance results for the algorithm on three synthetic tasks are reported in Experiments section, followed by discussion of the results and conclutions. The appendix contains supplementary material with equations and algorithms for the Relu and addition-based GRU and LSTM, as well as implementation details and parameters.

Preliminaries

Notations. Variables are denoted by small Roman letters and are generally vector-valued but in our examples often scalar. Sometimes subscript t is used to distinguish order in a sequence and j for individual elements in a vector variable, such as $h_{t,j}$ for the j-th element of the hidden state at time t. Functions are denoted by Greek letters, where special functions $\sigma(\cdot)$ are the sigmoid function, and $\phi_h(\cdot)$ are an activation function. Capitalized Roman letters are used for (constant) matrices, e.g., W_h and U_h, with the exception of the bias term b_h, which is a small letter. Note that subscript decoration is used for functions and constants to distinguish which variable they are associated with, such as for h in the examples above. There is no special notation to indicate that a quantity is encrypted. This will be clear by the context, but generally, the model weights of the neural networks are assumed to be clear text, while input variables, intermediate states, and outputs can be encrypted.

Homomorphic Encryption. Homomorphic Encryption is a cryptographic method that enables calculations on encrypted data by parties possessing only the public key. This technique facilitates applications where user data can be processed by a cloud service without the need for prior decryption. The computed results are then transmitted back to the secret key holder, who can decrypt the results. Throughout this process, the original data, intermediate computations, and final results remain indistinguishable from random noise to the computing cloud.

The term "homomorphic" signifies the support for a structure-preserving map between operations on encrypted

and unencrypted data. For arithmetic operations, we have:

$$\mathrm{enc}(x+y) = \mathrm{enc}(x) + \mathrm{enc}(y) \tag{3}$$

$$\mathrm{enc}(xy) = \mathrm{enc}(x)\mathrm{enc}(y) \tag{4}$$

where $\mathrm{enc}(x)$ represents the encryption operation applied to the clear text variable x. In this context, "clear text" refers to unencrypted data, while "cipher text" is used for encrypted data. Decryption is possible only with the private key and is considered highly secure, even against hypothetical quantum computer-based attacks (Augot et al. 2015). The term "Fully Homomorphic Encryption" distinguishes modern schemes supporting a universal set of operations such that any function can be computed. Earlier "Partially Homomorphic Encryption" schemes could only handle a limited set of operations, for example, either addition or multiplication.

One prominent scheme is CKKS (Cheon-Kim-Kim-Song) (Cheon et al. 2017), which encrypts complex numbers and performs fixed-point arithmetic. Like many other modern FHE schemes, its security is based on the Ring Learning With Errors (RLWE) problem (Regev 2009; Lyubashevsky, Peikert, and Regev 2013).

Practical Challenges Precision loss or cryptographic noise accumulates with each non-trivial operation for a modern Fully Homomorphic Encryption scheme. This can potentially render the results of large computations meaningless even after decryption unless mitigating measures like bootstrapping are employed. Bootstrapping involves decrypting and re-encrypting the data while staying within the ciphertext space. Although CKKS supports bootstrapping (Cheon et al. 2018), it is computationally expensive and often avoided by adopting a "leveled" approach.

The leveled approach requires careful management of the cryptographic noise budget, which determines the number of arithmetic operations that can be performed before bootstrapping becomes necessary. In practice, this translates to limiting the number of multiplications and additions. The noise budget is controlled by encryption parameters, which have to be carefully selected.

It is worth noting that arithmetic homomorphic encryption schemes, such as CKKS, support the evaluation of polynomial functions with a degree limited by the maximum multiplicative depth. Common mathematical functions like exponentials, sigmoids, ReLU, and trigonometric functions are only available to the extent that they can be approximated by limited-degree polynomials.

Ducas and Micciancio (Ducas and Micciancio 2015) introduced a significantly faster scheme that reduced bootstrapping time to less than a second. This technique was further refined into the TFHE (Fully Homomorphic Encryption on the Torus) scheme by Chillotti et al. (Chillotti et al. 2019), which is distinguished from other FHE schemes because it proposes a special bootstrapping, so-called Programmable Bootstrapping (PBS), which is comparably fast and able to evaluate a univariate function at the same time as it reduces the noise.

PBS in TFHE The TFHE bootstrap (Chillotti et al. 2019) does a blind rotation over a predefined table of all the permittable values in a message space, e.g $[-N, N-1]$, landing on the coefficient whose index is closest to the original cipher text x. By filling the lookup table with values of a function evaluated over the original message space, $f(x)$, one instead obtains the PBS which both reduces noise and evaluates a univariate function on a discrete set of points.

TFHE does not natively have multiplication between ciphertexts, but it can be constructed by the application of two PBS operations

$$a\,b = \mathrm{PBS}\,(f; a+b) - \mathrm{PBS}\,(f; a-b) \tag{5}$$

where by $\mathrm{PBS}\,(f; x)$, we mean applying a table that corresponds to the function f with argument x, which is

$$f(x) = \frac{x^2}{4} \tag{6}$$

in this case of multiplication by PBS.

Homomorphic Encryption imposes restrictions on the available operations for data processing compared to the free program flow often assumed in modern algorithm development for clear text data. Operations that break loops or selectively execute branching code conditional on cipher text are not feasible, as the encrypted data looks random to the entity performing the computation.

Parameter Selection. In order for the FHE scheme to be able to evaluate a circuit correctly, all the possible values that can occur for each component must be inside the allotted message space. The TFHE scheme operates on several different cipher-texts, namely LWE (Regev 2009), RLWE (Lyubashevsky, Peikert, and Regev 2013), GLWE (Brakerski, Gentry, and Vaikuntanathan 2012a), and generalizations of GSW (Gentry, Sahai, and Waters 2013), each governed by specific parameters. For LWE ciphertexts, a dimension parameter is required, while GLWE ciphertexts rely on both polynomial size and dimension parameters.

A framework for parameter optimization in TFHE proposed by (Bergerat et al. 2023) distinguishes such macro-parameters from the micro-parameters used exclusively within FHE operators, such as the decomposition base and the number of decomposition levels required for a PBS. To optimize FHE operators, both noise and cost models come into play. The noise model simulates noise evolution across an operator, while the cost model serves as a metric to minimize, encompassing factors like execution time, power consumption, or price. As the selection of micro and macro parameters significantly impacts cost and noise growth for each operation, they demand prudent and deliberate choices. For all numerical experiments in this paper, we use parameters automatically selected by the Concrete Python Compiler (Zama 2022) based on the TFHE parameter optimization framework by (Bergerat et al. 2023).

Privacy Preserving Machine Learning. Despite the computational challenges, variants of Fully Homomorphic Encryption have been successfully applied to achieve Privacy Preserving Machine Learning (PPML). Such efforts include hyperplane decision-based classification, Naive Bayes

classification, decision tree classification (Bost et al. 2015; Sun et al. 2020), classification and regression using Random Forests as demonstrated in Cryptotrees (Huynh 2020), and more recent developments of efficient nearest neighbor classifiers (Chakraborty and Zuber 2022).

Early approaches for Neural Networks like Cryptonets (Dowlin et al. 2016) focused on achieving blind, noninteractive classification by applying a leveled homomorphic encryption scheme (Brakerski, Gentry, and Vaikuntanathan 2012b) to the network inputs and used the square function as an activation function replacement to accommodate the limitations of the underlying encryption scheme. Subsequent works (Zhang, Yang, and Chen 2016) adopted higher-degree polynomials that approximate conventional activation functions like sigmoid and tanh on a range. However, leveled homomorphic approaches have limited scalability, and as the number of layers in a neural network increases, the overall performance of homomorphic classification becomes prohibitive. This is especially challenging for Recurrent Neural Networks, whose depth scales with the length of the sequence processed. To overcome these challenges, repeated interaction was employed, such that HE was used for the linear operations of the RNN while intermediate encrypted results were returned to the client to perform non-linear operations (Bakshi and Last 2020; Podschwadt and Takabi 2020).

To overcome the limitations of previous approaches, a hybrid cryptographic scheme (Juvekar, Vaikuntanathan, and Chandrakasan 2018) was proposed that combined homomorphic encryption (HE) for processing linear layers with garbled circuits (GC) for computing activations in convolutional neural networks (CNNs). The garbled circuit is a cryptographic protocol that enables secure computation in which two parties can jointly evaluate a function over their private inputs (Yao 1982). This hybrid approach for recurrent neural networks (RNNs) suffered from long inference latency due to slow GC-based activations, especially for activation functions like tanh and sigmoid that required large circuits. To address the latency issue, a novel secure gated recurrent unit (GRU) network framework called CRYPTOGRU was introduced (Feng et al. 2021). It achieved lower inference latency by using SIMD HE kernel functions for linear operations in a GRU cell and interactive GC to compute activations, replacing the costly sigmoid and tanh activations with ReLU.

The emerging TFHE scheme (Chillotti et al. 2016) showed promise for noninteractive PPML because of its fast bootstrap procedure and capacity to evaluate look-up tables. Based on this scheme, FHE–DiNN, a novel framework for homomorphic evaluation of discretized neural networks, was proposed (Bourse et al. 2018). It featured linear complexity in the network depth and overcomes the scale limitations of leveled approaches, and applies the sign function for neuron activation by using bootstrapping. Empirical results demonstrate accurate classification by fully connected neural networks of encrypted MNIST dataset images with over 96% accuracy in under 1.7 seconds. Also using TFHE, a shift-accumulation-based convolutional deep neural network (SHE) (Lou and Jiang 2019) was proposed to overcome the limitations of using polynomial approximations by implementing ReLU activations and max pooling. It employed logarithmic quantization to replace expensive FHE multiplications with cheaper shifts and claim state-of-the-art inference accuracy and reduced inference latency on MNIST and CIFAR-10 image classification data sets.

Computational Efficiency. Deep neural networks demand substantial computational resources due to their large-scale architecture and the intensive computations involved in both the training and deployment stages. These networks can consist of numerous layers, each containing thousands or even millions of parameters. Their execution entails extensive matrix operations, such as multiplications and additions, as well as the evaluation of activation functions like sigmoid or tanh. Consequently, it consumes significant amounts of electric energy.

To enhance energy efficiency, it is essential to make AI and machine learning systems more cost-effective, environmentally sustainable, scalable, and compatible with a wider range of devices. A survey by Sze et al. (Sze et al. 2017) provides valuable insights and tutorials on various approaches to reduce energy consumption and increase throughput while maintaining accuracy.

Efficiency improvements are achieved by optimizing software or hardware implementations for specific neural architectures down to individual operations. Some techniques involve modifying the architecture, such as compression, pruning, and employing compact network architectures. Others focus on reducing the precision of operations and operands, such as utilizing fixed-point arithmetic, bit-width reduction, and quantization of weights and activations.

Different operations within neural networks have distinct power and energy requirements, which can vary based on the underlying computer architecture (Parhami 2010).

Addition is a relatively straightforward operation that can be executed in a single instruction. It is considered to be relatively inexpensive also under FHE. In contrast, multiplication is generally more complex and expensive even for clear text and may necessitate the use of intricate algorithms.

Literal vs. Variable multiplication: Literal constants are fixed values that can be encoded in program instructions, while variables are undetermined at compile time. Literal multiplication can be optimized by the compiler using techniques like precomputed values or bit-shifting, resulting in faster execution and lower cost compared to variable multiplication also in clear text. Variable multiplication entails additional memory access to retrieve values, which can significantly impact energy consumption, as RAM access can be several orders of magnitude more energy-intensive than computation (Horowitz 2014). While multiplication by literals is natively supported in TFHE, the multiplication of two encrypted variables is constructed using two Programmable Bootstrap (PBS) operations, which is much more expensive and can dominate the overall computational cost.

Activation Functions. Activation functions play a vital role in neural networks. Functions like sigmoid require complex mathematical operations even in clear text, including exponentials and divisions, which are computationally expensive. On the other hand, the ReLU (Rectified Linear

Unit) activation function have simpler implementations involving threshold comparisons, enabling more efficient execution on various hardware architectures.

Univariate non-linear activation functions can be evaluated in TFHE through the Programmable Bootstrap (PBS) operation, although it incurs significant computational costs. For other schemes, like the CKKS, one is limited to polynomial activation functions.

Quantization Quantization of neural networks is a technique used to reduce the memory footprint and computational complexity of deep learning models (Polino, Pascanu, and Alistarh 2018) making them more efficient for deployment. The primary motivation behind quantization is to reduce memory and computation requirements by representing the model's inputs, weights and/or activations using lower precision data types, such as 8-bit integers, instead of the standard 32-bit floating-point numbers. This allows for faster inference times and lower energy consumption, making it feasible to run complex models in environments with limited computational resources like mobile phones or embedded systems (Wang et al. 2019). Aggressive quantization can lead to significant accuracy loss, particularly for complex models. Careful consideration is necessary to achieve efficiency gains without compromising critical performance metrics (Banner, Nahshan, and Soudry 2018).

As the Concrete Python library for TFHE only supports integers natively, quantization becomes necessary when implementing neural network architectures: inputs, weights, and activations all need to be projected from floating-point values onto the integers. Furthermore, since any deep neural network (like a gated RNN) with non-trivial activation functions will require PBS, the maximum precision of the TFHE table look-up implementation will be a limiting factor. At the time of writing, this was $2^7 = 128$ different values, i.e. integer 7-bit precision. Precision also directly impacts execution time as larger table lookups imply slower inference.

Method

Recall that this work aims to construct an RNN with long-term memory that doesn't use element-wise multiplication between variables in the gating mechanism. For ease of exposition, we will discuss a minimal gated recurrent neural network with long-term memory, first in a conventional multiplication and sigmoid-based formulation and later using the alternative addition and ReLU-based form. It is obtained by removing the reset variable from the GRU (see equation 32 and 33 in the Appendix). We denote this minimal architecture as the Gated Nominal Unit (GNU).

Gated Nominal Unit. We take the GRU update gate,

$$z_t = \sigma(W_z x_t + U_z h_{t-1} + b_z), \tag{7}$$

where matrices W_z, W_z, and b_z are two weight matrices and one bias vector as before. The proposal for the next state is a simplified version of that for the GRU,

$$\hat{h}_t = \phi_h(W_h h_{t-1} + W_h x_t + b_h), \tag{8}$$

without the reset gate, which was dropped. The update proposal is then combined with the previous state to obtain the current state

$$h_t = z_t \odot h_{t-1} + (1 - z_t) \odot \hat{h}_t \tag{9}$$

with weights given (element-wise) according to the value of the individual elements of the gate.

The sigmoid function is saturating, such that it may take values very close to 0 or 1 also for moderate inputs. Equation 9, which is borrowed from the GRU, can thus be set to extinguish either the previous state or the proposal, as well as weighted combinations of these for values in between. Elements of z_t that are close to 1 will simply pass through the previous state – in such case, there is no repeated multiplication with W_h – which is how the vanishing gradient problem is avoided, and memory can thus be preserved over longer sequences. On the other hand, for elements of z_t close to zero, the previous state is immediately replaced with the corresponding values of the newly proposed state.

Recently, the Inhibitor gating mechanism based on ReLU and addition was proposed for GRU and LSTM (Brännvall et al. 2023). The resulting networks exhibits similar limit behavior as the conventional gated RNN, but avoids the evaluation of Sigmoid activation function and elementwise variable multiplication. Furthermore, the novel gated RNNs demonstrated clear text performance both for inference and training on some standard benchmark tasks at par with conventional GRU and LSTM (Brännvall et al. 2023). The similar Inhibitor attention mechanism based on ReLU and addition was later proposed for Transformers (Brännvall 2024).

i-GNU: A ReLU and addition-based alternative to the above Gated Nominal Unit RNN is obtained as follows. We replace the sigmoid from the conventional gate of equation 7 with a general univariate activation function ϕ_u,

$$u_t = \phi_u \left(W_u x_t + U_u h_{t-1} + b_u \right), \tag{10}$$

which in many cases is set to be the identity function $\phi_u(x) = x$.

We take a similar proposal function as for the conventional gate

$$\hat{h}_t = \phi_h^+(W_h x_t + U_h h_{t-1} + b_h), \tag{11}$$

with the restriction that ϕ_h^+ here is non-negative.

The proposal is now combined with the prevous state, h_{t-1}, according to a new update rule

$$h_t = (h_{t-1} + u_t^-)^+ + (\hat{h}_t - u_t^+)^+ \tag{12}$$

using ReLU function and addition in place of sigmoid and multiplication of equation 9.

Note that the above gated RNN cell has the same limit behavior for large and small u as the corresponding multiplicative formulation of the GNU. We can see this by setting $z_t = \sigma(u_t)$ and ϕ_u as identity to get the conventional gate of equation 7. The limits of equation 9 and equation 12 are now the same.

Remark: The ReLU and addition-based gated RNN do not require any modifications for integer quantization. It is sufficient to assume that all inputs, weights, states, and functions are integer-valued in Equations 10 to 12 to realize integer quantization. For the conventional gate of equation 9 we minimally have to scale and quantize the sigmoid function, which otherwise takes real values between 0 and 1.

Experiments

TFHE parameters automatically determined by the TFHE Python Compiler (Zama 2022) that was used for this work, based on the set of valid combinations of input values that must be provided at compile time. The underlying integer representation itself is based on a popular extension to the LWE encryption scheme, as presented, for example, in (Bourse et al. 2018), for use in discretized neural networks.

As the parameters are set by the compiler to ensure that the probability of noise-overflow is neglible, the results are accurate by construction (with high confidence). Our experiments thus focus on computational efficiency and assume full accuracy (which is confirmed by inspecting the results).

Task 1: Synthetic Adding Problem

This synthetic task by (Hochreiter and Schmidhuber 1997) is known to require long-term memory to solve. The problem takes two inputs: a sequence of random numbers, v, and a two-hot sequence, w. We take $v \in [0,1]^n$ to be in the positive unit-cube and w be a sequence with entries of one for index i in $[0, n/2-1]$ and j in $[n/2, n-1]$, and zeroes otherwise. The target output is $y_n = v_i + v_j$, which is also equivalent to the dot product of the two vectors, $v \cdot w$.

Take for example the two random sequences for $n = 20$,

$$v = [1,8,7,2,8,6,5,2,4,0,9,6,2,3,1,6,9,9,1,4] \quad (13)$$

$$w = [0,0,0,0,1,0,0,0,0,0,0,0,0,1,0,0,0,0,0,0] \quad (14)$$

for which the output would be $y_n = v_i + v_j = 8 + 3 = 11$.

Solution. The task can be solved by a gated RNN like GRU or LSTM, however, a minimal solution is provided by the Gated Nominal Unit with integer vector valued input $x_t = [v_t, w_t]^T$ and integer scalar state h_t. Gating and proposal variables of Equations 10 and 11, are both governed by linear activation functions (i.e., $\phi(x) = x$). In this experiment, we compare the two alternative state update rules:

Multiplicative gate rule

$$h_t = \sigma(u_t) \odot h_{t-1} + (1 - \sigma(u_t)) \odot \hat{h}_t \quad (15)$$

Additive gate rule

$$h_t = (h_{t-1} + u_t^-)^+ + (\hat{h}_t - u_t^+)^+ \quad (16)$$

Handcrafted weights Let the hidden state be scalar, that is a vector of length 1, and propose to update the state by simply adding the current input v_t to the previous state h_{t-1}

$$\hat{h}_t = 1 \cdot h_{t-1} + [1,0]x_t + 0 = h_{t-1} + v_t \quad (17)$$

for the gate control variable,

$$u_t = 0 \cdot h_{t-1} + [-2a, 0]x_t + a = -2aw_t + a \quad (18)$$

where $a = 30$ is set sufficiently large to realize a gating mechanism between the previous state and the proposed state update. The output is simply the final state $y_n = h_n$.

Quantization To quantize the sigmoid function with k bit precision, we take as a basis $B = 2^k - 1$ and rewrite equation 15 as

$$h_t^* = z_t^* h_{t-1} + (B - z_t^*)\hat{h}_t, \quad (19)$$

where we dropped the Hadamard product notation $\odot$ as the state is scalar, and, furthermore, introduced the scaled and integer-valued gating variable

$$z_t^* = \lfloor B\,\sigma(u_t) \rceil, \quad (20)$$

whereby $\lfloor x \rceil$ we mean the operation of rounding x to the closest integer. We finally obtain the updated state as

$$h_t = \lfloor h_t^* / B \rceil. \quad (21)$$

For the synthetic adding problem, a single bit is sufficient to quantize the sigmoid function, that is, we have the rather trivial case $B = 1$ for $k = 1$. However, with this experiment, we also want to evaluate higher precision quantizations up to 4 bits to examine the scaling. This is because we assume that many practical machine-learning tasks would require more than 1-bit precision. As noted in the remark of section the ReLU and addition-based gated RNN does not require any modifications – it is natively integer valued as long as inputs, weights, and activations functions all are.

The input v_t is an integer between 0 and 9, and can therefore always be stored as type `uint4`, that is, a 4-bit unsigned integer. Consequently, as the current state h_t is the sum of two inputs, it can, at most take the value 18. Similarly, the update proposal of equation 11, can at most be the sum of three inputs, which is at most 27. Therefore `uint5` is sufficient for the state variables. The highest precision, `uint6`, is required by the gating variable, u_t, of equation 10, which can be either -30 or +30 depending on whether input w_t takes the value 0 or 1.

Task 2: Copying Memory Problem

The second problem is also a standard benchmark task for RNNs, which has been used to examine long-term memory in seminal work by (Hochreiter and Schmidhuber 1997) among others. Here we have a set of $n = 8$ different symbols, let us say $\{1,2,3,4,5,6,7,8\}$, and additionally two control markers $\{0,9\}$ that are used to signal when to stop recording and when to start replay from memory.

First we receive a sequence of k random symbols followed by $T-1$ instances of the blank symbol 0. The sequence is ended by m instances of the start-recall symbol 9, for which our program should recall the ten first symbols. Take for example a random sequence for $k = 7$ and $T = 5$,

$$x = [1,2,8,7,2,8,6,0,0,0,0,0,9,9,9,9,9,9,9,9] \quad (22)$$

for which the correct output would be the sequence

$$y = [0,0,0,0,0,0,0,0,0,0,0,0,1,2,8,7,2,8,6]. \quad (23)$$

The task can be solved by a gated RNN like GRU or LSTM, however, a minimal solution is provided by the Gated Nominal Unit we also used for the previous task, however, this time with scalar input x_t and vector-valued gating variable u_t and state h_t, where we take the activation function ϕ_u to be linear, and ϕ_h to be ReLU. As for the previous

Table 1: Mean execution time on CPU over 100 trials for gated RNN solvers of the synthetic addition problem (task 1) and the copying memory problem (task 2). The table compares the proposed additive inhibitor gate (Add) with the conventional multiplicative (Mul) at different precision.

	Add 4b	Mul 1b	Mul 2b	Mul 3b	Mul 4b
Task 1	0.5	0.44	0.62	2.03	31.5
Task 2	1.13	1.61	2.33	10.1	104.

task we compare the performance of the addition-based gate of equation 16 and the conventional of equation 15.

Also for this problem we can handcraft the weights. These implement conditional copy and shift operations, but as the state is vector valued they are rather large matrices whose derivation we defer to the Appendix, together with the details on the quantization.

In summary, the quantization of the sigmoid function is carried out in the same way as for Task 1. The input x_t is an integer between 0 and 9, and can, therefore always be stored as type `uint4`, that is, a 4-bit unsigned integer. The update-proposal wraps a ReLU activation function, and is consequently non-negative. Its entries are used for memory and take values on the same range as the input (type `uint4`). The highest precision, `int5`, is required by the gating variable, u_t, which take values on the range -9 or +9.

Results for task 1 and 2

Table 1 reports execute time (in seconds) for evaluating one step of the gated RNN solutions to Task 1 and 2. The precision for quantization of the sigmoid function is noted in the column labels for the multiplication-based gated RNN (while there is no quantization required for the addition-based gate). It is evident that the computational cost quickly grows with the precision of the quantization for both tasks.

Our experiments for these task only use hand-crafted weights, as the focus is on execution time for inference under FHE and not model training. Earlier work confirmed that the training capacity of the proposed Inhibitor gated RNNs is comparable to the conventional (Brännvall et al. 2023).

We note that the conventional gate at the highest bit precision, $k = 4$, is 60 to 90 times slower than the addition-based gate. On the other hand, the difference in execution time is not so large at $k = 1$ bit quantization. It turns out that for both these problems one bit quantification of the sigmoid function is sufficient precision for an exact solution. The advantages of the addition-based gate should become important for real-world problems that require higher precision.

Table 3 and 4 in the Appendix reports compilation parameters and the circuit max bit precision required for both tasks. For Task 1, we can see that polynomial size ranges from 4096 for the addition-based gate up to 65536 for the 4-bit precision quantization of the multiplicative gate.

Scaling experiment

The experiment reported in this section examines a wider range of input, state, and weight settings. We consider the Gated Nominal Unit from the earlier experiments but with the precision of the input x_t variable and state h_t variable ranging from 2-bit to 8-bit.

Table 2: Mean execution time in seconds on CPU for one step of a scalar gated RNN with 1-bit quantized weights and n bit precision for input and state (100 trials), where the proposed additive gate (Add) is compared with the conventional multiplicative gate (Mul) at different precision.

n	Add 4b	Mul 1b	Mul 2b	Mul 3b	Mul 4b
2	0.13	0.13	0.21	0.26	0.63
3	0.18	0.19	0.26	0.6	2.02
4	0.22	0.22	0.62	2.1	5.57
5	0.52	0.53	2.03	5.56	25.58
6	1.76	1.79	5.48	25.94	
7	4.64	4.84	26.39		
8	21.06	22.49			

The activation functions ϕ_h and ϕ_h of equations 10 and 11 are selected to be a `clip` function that scales and clips the values of u_t and $\hat{h}_t$ to fall within an allowed numerical range, i.e. having the appropriate type to allow combination according to the addition based rule of equation 16 or the multiplication based rule of equation 15.

The experiments are repeated both for randomly sampled inputs, states, and weights, as well as for all combinations of minimum and maximum values according to their respective type and precision. The quantization of the sigmoid function is carried out in the same way as in Task 1, that is, through equations 19 – 21, for $k = 1 \dots 4$ bit precision, for each input and state quantization.

Results. The execution times reported in Table 2 agree with the trend we gleaned from the other two tasks. The addition-based gate shows a clear advantage by either offering faster execution at a given combined precision or higher precision at a given computation budget.

We note missing entries in the table for combinations with both high variable precision and sigmoid quantization. The Python kernel throws an error when attempting to evaluate these networks, which relies on extremely large GLWE polynomial size. Table 5 in the Appendix that reports captured compilation parameters reveals that the required polynomial size is 131072 at a combined maximum circuit precision of `uint10`, for example, for $n = 6$ and $k = 4$, which does not execute for the multiplicative gate. The corresponding addition-based gate at the same input and state precision level of $n = 6$ only requires a polynomial-size of 8192 and a maximum circuit precision level of `uint7`. The difference, in this case, is between completing the computation in 1.76 s for the addition-based gate or causing a memory error that crashes the Python kernel for the multiplication-based gate.

Conclusions

Our work examines the numerical performance of a novel RNN architecture based on the Inhibitor gate that uses ReLU and addition in place of sigmoid function and multiplication. It overcomes the challenges of cipher text to cipher

text multiplication that troubles conventional gated RNNs like LSTM and GRU when executed under homomorphic encryption. In addition, it natively supports integer quantization without the need for rescaling modifications of activation functions that are required by the conventional gate.

Numerical experiments indicate that the addition-based gated RNN provides an advantage over the conventional gated RNNs, by either offering faster execution at a given combined circuit precision or higher precision at a given computation budget. The implementations of gated RNNs such as LSTM and GRU that are efficient and accurate under TFHE can enable privacy-preserving applications in real-world use cases for sequential data such as time-series analysis and natural language processing due to their ability to capture long-term dependencies.

In parallel work we have proposed the Inhibitor mechanism for Transformers (Brännvall 2024) and examined its numerical performance under FHE (Brännvall and Stoian 2024). For future work, we aim to implement this construct for larger real-world problems also under encryption.

References

Agarwal, P.; and Alam, M. 2020. A Lightweight Deep Learning Model for Human Activity Recognition on Edge Devices. *Procedia Computer Science*, 167: 2364–2373. International Conference on Computational Intelligence and Data Science.

Augot, D.; et al. 2015. Initial recommendations of long-term secure post-quantum systems.

Bahdanau, D.; Cho, K.; and Bengio, Y. 2015. Neural machine translation by jointly learning to align and translate. 3rd International Conference on Learning Representations, ICLR 2015 ; Conference date: 07-05-2015 Through 09-05-2015.

Bakshi, M.; and Last, M. 2020. *CryptoRNN - Privacy-Preserving Recurrent Neural Networks Using Homomorphic Encryption*, 245–253. ISBN 978-3-030-49784-2.

Banner, R.; Nahshan, Y.; and Soudry, D. 2018. Post training 4-bit quantization of convolutional networks for rapid-deployment. In *Neural Information Processing Systems*.

Bengio, Y.; Simard, P.; and Frasconi, P. 1994. Learning long-term dependencies with gradient descent is difficult. *IEEE Transactions on Neural Networks*, 5(2): 157–166.

Bergerat, L.; Boudi, A.; Bourgerie, Q.; Chillotti, I.; Ligier, D.; Orfila, J.-B.; and Tap, S. 2023. Parameter Optimization and Larger Precision for (T)FHE. *J. Cryptol.*, 36(3).

Bost, R.; et al. 2015. Machine Learning Classification over Encrypted Data. *IACR Cryptol. ePrint Arch.*, 2014: 331.

Bourse, F.; Minelli, M.; Minihold, M.; and Paillier, P. 2018. Fast Homomorphic Evaluation of Deep Discretized Neural Networks. In *Advances in Cryptology – CRYPTO 2018: 38th Annual International Cryptology Conference, Santa Barbara, CA, USA, August 19–23, 2018, Proceedings, Part III*, 483–512. Berlin, Heidelberg: Springer-Verlag. ISBN 978-3-319-96877-3.

Brakerski, Z.; Gentry, C.; and Vaikuntanathan, V. 2012a. (Leveled) Fully Homomorphic Encryption without Bootstrapping. In *Proceedings of the 3rd Innovations in Theoretical Computer Science Conference*, ITCS '12, 309–325. New York, NY, USA: Association for Computing Machinery. ISBN 9781450311151.

Brakerski, Z.; Gentry, C.; and Vaikuntanathan, V. 2012b. (Leveled) fully homomorphic encryption without bootstrapping. In *Proceedings of the 3rd Innovations in Theoretical Computer Science Conference*. ACM.

Brännvall, R. 2024. The Inhibitor: ReLU and Addition-Based Attention for Efficient Transformers (Student Abstract). In *Proceedings of the 38th AAAI Conference on Artificial Intelligence (forthcoming)*.

Brännvall, R.; Forsgren, H.; Sandin, F.; and Liwicki, M. 2023. ReLU and Addition-based Gated RNN. arXiv:2308.05629.

Brännvall, R.; and Stoian, A. 2024. The Inhibitor: ReLU and Addition-Based Attention for Efficient Transformers under Fully Homomorphic Encryption on the Torus. To be presented at FHE.org Toronto 2024 Conference.

Chakraborty, O.; and Zuber, M. 2022. Efficient and Accurate Homomorphic Comparisons. In *Proceedings of the 10th Workshop on Encrypted Computing & Applied Homomorphic Cryptography*. ACM.

Cheon, J. H.; Han, K.; Kim, A.; Kim, M.; and Song, Y. 2018. Bootstrapping for Approximate Homomorphic Encryption. In *Advances in Cryptology – EUROCRYPT 2018*, 360–384. Springer Int. Publ.

Cheon, J. H.; Kim, A.; Kim, M.; and Song, Y. 2017. Homomorphic Encryption for Arithmetic of Approximate Numbers. In *Advances in Cryptology – ASIACRYPT 2017*, 409–437. Springer International Publishing.

Chillotti, I.; Gama, N.; Georgieva, M.; and Izabachène, M. 2016. Faster Fully Homomorphic Encryption: Bootstrapping in less than 0.1 Seconds. Cryptology ePrint Archive, Paper 2016/870. https://eprint.iacr.org/2016/870.

Chillotti, I.; et al. 2019. TFHE: Fast Fully Homomorphic Encryption Over the Torus. *Journal of Cryptology*, 33(1): 34–91.

Chung, J.; Gulcehre, C.; Cho, K.; and Bengio, Y. 2014. Empirical Evaluation of Gated Recurrent Neural Networks on Sequence Modeling.

Dowlin, N.; Gilad-Bachrach, R.; Laine, K.; Lauter, K.; Naehrig, M.; and Wernsing, J. 2016. CryptoNets : Applying Neural Networks to Encrypted Data with High Throughput and Accuracy.

Ducas, L.; and Micciancio, D. 2015. FHEW: Bootstrapping Homomorphic Encryption in Less Than a Second. In *Advances in Cryptology – EUROCRYPT 2015*, 617–640. Springer Berlin Heidelberg.

Feng, B.; Lou, Q.; Jiang, L.; and Fox, G. 2021. CRYPTOGRU: Low Latency Privacy-Preserving Text Analysis With GRU. In *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing*. Association for Computational Linguistics.

Gentry, C. 2010. Computing arbitrary functions of encrypted data. *Communications of the ACM*, 53(3): 97–105.

Gentry, C.; Sahai, A.; and Waters, B. 2013. Homomorphic Encryption from Learning with Errors: Conceptually-Simpler, Asymptotically-Faster, Attribute-Based. In Canetti, R.; and Garay, J. A., eds., *Advances in Cryptology – CRYPTO 2013*, 75–92. Berlin, Heidelberg: Springer Berlin Heidelberg. ISBN 978-3-642-40041-4.

Graves, A.; Liwicki, M.; Fernández, S.; Bertolami, R.; Bunke, H.; and Schmidhuber, J. 2009. A Novel Connectionist System for Unconstrained Handwriting Recognition. *IEEE transactions on pattern analysis and machine intelligence*, 31: 855–68.

Hochreiter, S. 1991. Untersuchungen zu dynamischen neuronalen Netzen. Diploma, Technische Universität München.

Hochreiter, S.; Bengio, Y.; Frasconi, P.; and Schmidhuber, J. 2009. Gradient Flow in Recurrent Nets: The Difficulty of Learning LongTerm Dependencies. In *A Field Guide to Dynamical Recurrent Networks*. IEEE.

Hochreiter, S.; and Schmidhuber, J. 1997. Long Short-Term Memory. *Neural Computation*, 9(8): 1735–1780.

Horowitz, M. 2014. 1.1 Computing's energy problem (and what we can do about it). volume 57, 10–14. ISBN 978-1-4799-0920-9.

Huynh, D. 2020. Cryptotree: fast and accurate predictions on encrypted structured data. arXiv:2006.08299.

Juvekar, C.; Vaikuntanathan, V.; and Chandrakasan, A. 2018. GAZELLE: A Low Latency Framework for Secure Neural Network Inference. In *27th USENIX Security Symposium (USENIX Security 18)*, 1651–1669. Baltimore, MD: USENIX Association. ISBN 978-1-939133-04-5.

Lou, Q.; and Jiang, L. 2019. *SHE: A Fast and Accurate Deep Neural Network for Encrypted Data*. Red Hook, NY, USA: Curran Associates Inc.

Lyubashevsky, V.; Peikert, C.; and Regev, O. 2013. On Ideal Lattices and Learning with Errors over Rings. *J. ACM*, 60(6).

Mikolov, T. 2012. Statistical Language Models Based on Neural Networks. PhD thesis, Brno University of Technology.

Parhami, B. 2010. *Computer Arithmetic: Algorithms and Hardware Designs*. Oxford University Press, New York. 2nd edition.

Pascanu, R.; Mikolov, T.; and Bengio, Y. 2013. On the Difficulty of Training Recurrent Neural Networks. In *Proceedings of the 30th International Conference on International Conference on Machine Learning - Volume 28*, ICML'13. JMLR.org.

Podschwadt, R.; and Takabi, D. 2020. Classification of Encrypted Word Embeddings using Recurrent Neural Networks. In Feyisetan, O.; Ghanavati, S.; Rokhlenko, O.; and Thaine, P., eds., *Proceedings of the PrivateNLP 2020: Workshop on Privacy in Natural Language Processing - Colocated with WSDM 2020, Houston, USA, Feb 7, 2020*, volume 2573 of *CEUR Workshop Proceedings*, 27–31. CEUR-WS.org.

Polino, A.; Pascanu, R.; and Alistarh, D. 2018. Model compression via distillation and quantization. In *International Conference on Learning Representations*.

Regev, O. 2009. On Lattices, Learning with Errors, Random Linear Codes, and Cryptography. *J. ACM*, 56(6).

Sun, X.; et al. 2020. Private Machine Learning Classification Based on Fully Homomorphic Encryption. *IEEE Transactions on Emerging Topics in Computing*, 8(2): 352–364.

Sze, V.; hsin Chen, Y.; Yang, T.-J.; and Emer, J. S. 2017. Efficient Processing of Deep Neural Networks: A Tutorial and Survey. *Proceedings of the IEEE*, 105: 2295–2329.

Vaswani, A.; Shazeer, N.; Parmar, N.; Uszkoreit, J.; Jones, L.; Gomez, A. N.; Kaiser, L. u.; and Polosukhin, I. 2017. Attention is All you Need. In Guyon, I.; Luxburg, U. V.; Bengio, S.; Wallach, H.; Fergus, R.; Vishwanathan, S.; and Garnett, R., eds., *Advances in Neural Information Processing Systems*, volume 30. Curran Associates, Inc.

Wang, K.; Liu, Z.; Lin, Y.; Lin, J.; and Han, S. 2019. HAQ: Hardware-Aware Automated Quantization With Mixed Precision. In *2019 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, 8604–8612.

Yao, A. C. 1982. Protocols for secure computations. In *23rd Annual Symposium on Foundations of Computer Science (sfcs 1982)*. IEEE.

Zama. 2022. Concrete: TFHE Compiler that converts python programs into FHE equivalent. https://github.com/zama-ai/concrete.

Zhang, Q.; Yang, L. T.; and Chen, Z. 2016. Privacy Preserving Deep Computation Model on Cloud for Big Data Feature Learning. *IEEE Transactions on Computers*, 65(5): 1351–1362.

Appendix

Recurrent Neural Networks

Recurrent Neural Networks (RNNs) are a class of neural networks specially designed for processing sequential data, such as speech recognition, language translation, and handwriting recognition. They possess the ability to incorporate information from previous inputs to influence future outputs, enabling them to capture dependencies and context over temporal sequences.

The simplest RNN architecture consists of a single repeating module. This module takes into account the previous hidden state, denoted as h_{t-1}, and the current input, denoted as x_t, to produce the current hidden state, h_t. It can be represented by the equation:

$$h_t = \phi_h(W_h x_t + U_h h_{t-1} + b_h) \tag{24}$$

where W_h and U_h are matrices that assign different weights to specific components of the input and state vectors, and b_h is a bias term. The activation function ϕ_h introduces non-linearity to the hidden state. By utilizing the hidden state, RNNs are capable of processing sequential data effectively.

However, the simple RNN suffers from the vanishing gradient problem, which limits its ability to learn long-term dependencies. To address this issue, gating mechanisms were

introduced. These mechanisms modify the state based on the combination of the previous state and an update proposal, allowing information to be selectively retained or forgotten.

LSTM. The Long Short-Term Memory (LSTM) proposed by Hochreiter and Schmidhuber (Hochreiter and Schmidhuber 1997) incorporates a memory cell, input gate, output gate, and forget gate to control the flow of information within the network. The forget gate determines what information should be discarded from the memory cell, while the input gate decides what new information should be stored. The output gate regulates the output based on the memory cell's content. The LSTM equations are as follows:

$$f_t = \sigma(W_f x_t + U_f h_{t-1} + b_f) \tag{25}$$
$$i_t = \sigma(W_i x_t + U_i h_{t-1} + b_i) \tag{26}$$
$$o_t = \sigma(W_o x_t + U_o h_{t-1} + b_o) \tag{27}$$
$$\hat{c}_t = \phi_c(W_c x_t + U_c h_{t-1} + b_c) \tag{28}$$
$$c_t = f_t \odot c_{t-1} + i_t \odot \hat{c}_t \tag{29}$$
$$h_t = o_t \odot \phi_h(c_t) \tag{30}$$

In Equations 25-30, σ denotes the sigmoid activation function, $\odot$ represents element-wise multiplication, and ϕ_c and ϕ_h denote activation functions for the memory cell and hidden state, respectively.

GRU. Another gated architecture known as the Gated Recurrent Unit (GRU) was introduced by Chung et al. (Chung et al. 2014). GRU combines the memory cell and gate mechanisms of LSTM into a single unit, making it computationally efficient and easier to train. GRU incorporates a reset gate and an update gate, which control the information flow and enable the capturing of long-term dependencies. The GRU equations are as follows:

$$z_t = \sigma(W_z x_t + U_z h_{t-1} + b_z) \tag{31}$$
$$r_t = \sigma(W_r x_t + U_r h_{t-1} + b_r) \tag{32}$$
$$\hat{h}_t = \phi_h(W_h x_t + U_h(r_t \odot h_{t-1}) + b_h) \tag{33}$$
$$h_t = z_t \odot h_{t-1} + (1 - z_t) \odot \hat{h}_t \tag{34}$$

In Equations 31-34, z_t represents the update gate, r_t is the reset gate, and $\hat{h}_t$ denotes the candidate proposal activation.

GNU. We obtain the minimal architecture (the Gated Nominal Unit, or GNU) used in the experiments by removing the reset variable of equation 32 and the corresponding element-wise product in the update proposal of equation 33 of the GRU architecture (i.e., setting $r_t \equiv 1$).

ReLU and Addition-based GRU and LSTM

The numerical experiments in the main part of this article employed a simplified version of the gated RNNs (here called the Gated Nominal Unit, or GNU), to examine the performance of the ReLU and addition-based gated RNNs. For completeness, we also present the full equations for the addition-based alternatives to the more common LSTM and GRU architectures below. These are, what in fact would be used for real-world problems testing and comparing gated RNNs. We denote these ReLU and addition-based gated RNNs as a-GRU and a-LSTM, where the *a* can be read as standing for alternative or addition-based version of the original.

i-GRU: To construct the ReLU and addition-based GRU, we first define the gating variable as

$$u_t = \phi_u\left(W_u x_t + U_u h_{t-1} + b_u\right), \tag{35}$$

which, for example, can be set to the identity function $\phi_u(x) = x$, as was done for the solutions to the synthetic tasks of Section . The GRU architecture additionally has a reset variable,

$$r_t = \phi_r^+(W_r x_t + U_r h_{t-1} + b_r), \tag{36}$$

which can be, for example, the ReLU function, or another non-negative function. For the proposal function, we take

$$\hat{h}_t = \phi_h^+(W_h x_t + U_h(h_{t-1} - r_t)^+ + b_h) \tag{37}$$

which we require to be non-negative to achieve the gating property. The newly proposed state is combined with the previous state, h_{t-1}, according to the ReLU and addition-based rule

$$h_t = (h_{t-1} + u_t^-)^+ + (\hat{h}_t - u_t^+)^+ \tag{38}$$

so that the combined equations now are a replacement to equations 31 to 34, that is, the original GRU where multiplicative gates are replaced with the ReLU and addition-based gate.

i-LSTM: For a ReLU and addition-based LSTM, we replace all of the original equations (25) to (30) with

$$f_t = \phi_f^+(W_f x_t + U_f h_{t-1} + b_f) \tag{39}$$
$$i_t = \phi_i^+(W_i x_t + U_i h_{t-1} + b_i) \tag{40}$$
$$o_t = \phi_o^+(W_o x_t + U_o h_{t-1} + b_o) \tag{41}$$
$$\hat{c}_t = \phi_c^+(W_c x_t + U_c h_{t-1} + b_c) \tag{42}$$
$$c_t = (c_{t-1} - f_t)^+ + (\hat{c}_t - i_t)^+ \tag{43}$$
$$h_t = (\phi_h^+(c_t) - o_t))^+ \tag{44}$$

where ϕ_f, ϕ_i, and ϕ_o, are non-negative functions, for example, ReLU, which are used to evaluate the gating variables.

Similarly, ϕ_c and ϕ_h are required to be non-negative and are preferably set to some saturating functions, like sigmoid or shifted tanh, $\phi_c(x) = 1 + \tanh(x)$.

Handcrafted weights for the copying problem

Let the hidden state be of size $m + 2$, for which the first m entries are used as a memory bank. Entry m is for output and entry $r = m + 1$ indicates whether we are in sleep mode or record/replay. We propose state by shifting memory upwards towards the output position and placing the input symbol in the 0th position. Therefore, for example for $m = 2$, we have weights

$$W_h = \begin{bmatrix} 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & -1 \end{bmatrix}, U_h = \begin{bmatrix} 1 \\ 0 \\ 0 \\ -1 \end{bmatrix}, b_h = \begin{bmatrix} 0 \\ 0 \\ 0 \\ 1 \end{bmatrix}, \tag{45}$$

where the bottom row, position $r = m + 1$, track whether we are operating in record/replay mode.

Since all the matrix weights are known in clear text, we can simplify the matrix equation and write it as copy and shift operations

$$\hat{h}_{t,i} = h_{t-1,i-1}, \quad \text{for } i = 1, \ldots, m \tag{46}$$

$$\hat{h}_{t,0} = x_t \tag{47}$$

and

$$\hat{h}_{t,r} = (1 - x_t - \hat{h}_{t-1,r})^+ \tag{48}$$

where the $\mathrm{ReLU}(x) = x^+$ activation has been omitted for entries that are always positive (i.e. all positions except $r = m + 1$).

The gating variable u_t is vector valued with size $m + 2$, however the first $m+1$ entries are identical, for example for $m = 2$ we have,

$$W_u = \begin{bmatrix} 0 & 0 & 0 & 18 \\ 0 & 0 & 0 & 18 \\ 0 & 0 & 0 & 18 \\ 0 & 0 & 0 & 0 \end{bmatrix}, U_u = \begin{bmatrix} 0 \\ 0 \\ 0 \\ -1 \end{bmatrix}, b_u = \begin{bmatrix} -9 \\ -9 \\ -9 \\ 0 \end{bmatrix}, \tag{49}$$

which can be simplified since all weight coefficients are known in clear text. We can write

$$u_{t,i} = 18 h_{t-1,r} - 9, \quad \text{for } i = 0 \ldots m \tag{50}$$

and

$$u_{t,r} = -x_t \tag{51}$$

for the operating mode indicator row r.

Finally for the output, e.g. also for $m = 2$,

$$y_t = [0 \quad 0 \quad 1 \quad 0]\, h_t = h_{t,m}$$

Quantization. The input x_t is an integer between 0 and 9, and can, therefore always be stored as type `uint4`, that is, a 4-bit unsigned integer. The update-proposal of equation 11 wraps a ReLU activation function, and is consequently non-negative. All entries for which $i = 0 \ldots m$, that is, equations 46 and 47, are used for memory or output and take values on the same range as the input and of type `uint4`. Entry $r = m + 1$ is `uint1` according to equation 48 since it only can take values 0 or 1. The highest precision, `int5`, is required by the gating variable, u_t for positions $i = 0 \ldots m$ (equation 50), which can be either -9 or +9 depending on whether the previous state at position $r = m + 1$ take the value 0 or 1. It is also `int5` for position $r = m + 1$ (equation 50) as this variable takes values between -9 and 0.

TFHE Circuit Parameters

Here we report the TFHE parameters that were set by the Concrete Python compiler for the different tasks examined in the numerical experiments of Section .

Table 3: Parameters and circuit bit size set by Concrete TFHE compiler for Task 1, the synthetic adding problem.

	lweDim	baseLog	level	polySize	int	uint
Add 4bit	856	15	2	4096	6	6
Mul 1bit	912	22	1	4096	6	6
Mul 2bit	870	22	1	4096	6	6
Mul 3bit	946	15	2	8192	6	7
Mul 4bit	1021	14	2	65536	6	9

Table 4: Parameters and circuit bit size set by Concrete TFHE compiler for Task 2, the copying memory task, with a sequence of length 8 to remember.

	lweDim	baseLog	level	polySize	int	uint
Add 4bit	821	15	2	2048	5	5
Mul 1bit	834	23	1	2048	5	5
Mul 2bit	809	23	1	2048	5	5
Mul 3bit	884	22	1	8192	5	6
Mul 4bit	998	15	2	32768	5	8

Table 5: Parameters and circuit bit size set by Concrete TFHE compiler for the stylized scalar gated RNN with 1-bit quantized weights and n bit precision for input and state variables.

n	gate	lweDim	baseLog	level	polySize	int	uint
2	Add 4b	743	18	1	512	3	3
2	Mul 1b	743	18	1	512	3	3
2	Mul 2b	824	23	1	1024	3	4
2	Mul 3b	802	23	1	2048	3	5
2	Mul 4b	872	22	1	4096	3	6
3	Add 4b	823	23	1	1024	4	4
3	Mul 1b	823	23	1	1024	4	4
3	Mul 2b	802	23	1	2048	4	5
3	Mul 3b	872	22	1	4096	4	6
3	Mul 4b	954	15	2	8192	4	7
4	Add 4b	800	23	1	2048	5	5
4	Mul 1b	800	23	1	2048	5	5
4	Mul 2b	872	22	1	4096	5	6
4	Mul 3b	954	15	2	8192	5	7
4	Mul 4b	999	15	2	16384	5	8
5	Add 4b	868	22	1	4096	6	6
5	Mul 1b	869	22	1	4096	6	6
5	Mul 2b	954	15	2	8192	6	7
5	Mul 3b	999	15	2	16384	6	8
5	Mul 4b	1022	14	2	65536	6	9
6	Add 4b	946	15	2	8192	7	7
6	Mul 1b	946	15	2	8192	7	7
6	Mul 2b	999	15	2	16384	7	8
6	Mul 3b	1022	14	2	65536	7	9
6	Mul 4b	1114	14	2	131072	7	10
7	Add 4b	998	15	2	16384	8	8
7	Mul 1b	998	15	2	16384	8	8
7	Mul 2b	1022	14	2	65536	8	9
7	Mul 3b	1114	14	2	131072	8	10
8	Add 4b	1021	14	2	65536	9	9
8	Mul 1b	1021	14	2	65536	9	9
8	Mul 2b	1114	14	2	131072	9	10