12-11-2023

Privacy-Preserving Deep Learning with Homomorphic Encryption: Addressing Challenges Related to Usability, Memory, and Recurrent Neural Networks

Robert Podschwadt
Georgia State University

Privacy-Preserving Deep Learning with Homomorphic Encryption: Addressing Challenges Related to Usability, Memory, and Recurrent Neural Networks

by

Robert Podschwadt

Under the Direction of Zhipeng Cai, Ph.D. and Daniel Takabi, Ph.D.

A Dissertation Submitted in Partial Fulfillment of the Requirements for the Degree of

Doctor of Philosophy

in the College of Arts and Sciences

Georgia State University

2023

ABSTRACT

Neural networks enable many exciting technologies and products that analyze and process our data. This data is often privacy-sensitive, and we grant companies access to it because we want to use their services. We have little to no control over what happens to our data after that. Privacy-preserving machine learning provides solutions that allow us to use these services while maintaining the privacy of our data. Homomorphic encryption (HE) is one of the techniques that powers privacy-preserving machine learning. It allows us to perform computation on encrypted data without revealing the input, intermediate, or final results. However, HE comes with several limitations and significant resource overhead.

The most important limitations are a reduced set of operations, HE only supports addition and multiplication, and a bound on the depth of the computation. The depth of computation is the number of consecutive operations needed to complete it. This means we cannot compute arbitrary neural networks over encrypted data since they often rely on unsupported operations or are too deep. Recurrent neural networks (RNN) suffer especially from depth limitation. Due to their structure, RNNs produce comparatively deep computation.

In this work, we focus on multiple challenges of neural networks on encrypted data. One of the main challenges common to most architectures is the resource overhead introduced by HE. Since encrypted data is often orders of magnitude larger than plain data memory becomes a significant bottleneck. We analyze the computation of neural network layers and develop a caching and swapping scheme that allows us to dynamically load and unload data from memory while sacrificing as little time as possible. We further study the challenges specific to RNNs, mainly the computational depth. We find that naïve approaches for RNN over encrypted data are too deep. To address this, we design and evaluate multiple architectures that feature recurrent components to capture their strength in sequences but

have a lower depth. To evaluate these models, we design and implement an encrypted neural network runtime based on Tensorflow's XLA (accelerated linear algebra) compiler that allows us to run neural networks over encrypted data with very few extra steps.

INDEX WORDS: privacy, machine learning, neural networks, recurrent neural networks, privacy-preserving machine learning, homomorphic encryption

Privacy-Preserving Deep Learning with Homomorphic Encryption: Addressing Challenges Related to Usability, Memory, and Recurrent Neural Networks

by

Robert Podschwadt

Committee Chair: Zhipeng Cai

Daniel Takabi

Committee: Sergey Plis

Eduardo Blanco

Electronic Version Approved:

Office of Graduate Services

College of Arts and Sciences

Georgia State University

December 2023

DEDICATION

I am dedicating this dissertation to my wonderful wife and partner, Alexis. Without her, I would never have made it this far. She has always supported me, motivated me, and given me the strength to see this through to the end.

Thank you, Alexis. I love you!

ACKNOWLEDGMENTS

I am very grateful for having been given the opportunity to pursue my Ph.D. Although a significant portion of the time I spent at GSU was overshadowed by Covid, the lockdown, and remote work, I will always cherish the memories I made and the people I met. I want to express my sincerest gratitude to my advisor, Dr. Daniel Takabi, who has stuck with me through multiple schools, states, and stages of my research. His advice and guidance have been invaluable in the process. Thank you for all the opportunities you have provided for me! I would also like to thank the other members of my committee, Dr. Zhipeng Cai, Dr. Sergey Plis, and Dr. Eduardo Blanco. They have always provided valuable insight and been a great source of motivation. Thank you so much.

I also want to thank all my current and former lab mates. You have always provided helpful insights, stimulating conversations, or much-needed relief and distraction. This time would not have been the same without you. I want to thank all the people in the Computer Science Department and the college who helped me navigate the Ph.D. process. You are wonderful.

Last but not least, I also want to thank my friends and family, who kept me sane during this long process. My deepest thanks go out to: My wonderful wife, Alexis who always had my back. She provided me with invaluable support, motivation, and the occasional push when needed. My parents, Moni and Armin who always supported me and my endeavors. My brother Max and his wife Carol who have been cheering for me along the way. Jody

and Zach who made the time lockdown so much better. Moxie and Smudge who provide constant, unrelenting entertainment and support. And my friends from all over the world who are always there when I need them.

TABLE OF CONTENTS

ACKNOWLEDGMENTS . . . . . **v**

LIST OF TABLES . . . . . **xii**

LIST OF FIGURES . . . . . **xv**

LIST OF ABBREVIATIONS . . . . . **xviii**

1 INTRODUCTION AND MOTIVATION **1**

1.1 Motivation 2

1.2 Challenges of Deep Learning with Homomorphic Encryption 6

1.2.1 Limited Supported Operations 7

1.2.2 Limited Model Depth 8

1.2.3 Resource Overhead 9

1.3 Contributions 9

1.4 Organization 10

2 BACKGROUND **12**

2.1 Homomorphic Encryption 13

2.1.1 Single Instruction Multiple Data 17

2.1.2 The CKKS Scheme 17

2.2 What is an HE-friendly model? 21

2.2.1 What is required to make a model HE-friendly? 21

2.2.2 Polynomial Approximation: Theoretical Foundation 32

2.3 Compiler-based approaches to running neural network with HE . . 33

2.4 Recurrent Neural Networks 34

2.5 Convolutional Layers 35

2.5.1 Lock-free multi-threaded Convolution 36

3 PRIVATE INFERENCE 39

3.1 Challenges of running a trained model on encrypted data 39

***3.1.1 RNNs on encrypted data* 47**

***3.1.2 Addressing Challenges* 51**

3.2 Related Work 51

3.3 System Description 54

***3.3.1 Compiler* 56**

***3.3.2 XLA* 57**

***3.3.3 Connecting HE and Tensorflow* 60**

***3.3.4 Supported Operations* 65**

***3.3.5 Supported Layouts* 67**

***3.3.6 Optimization* 68**

3.4 Evaluation and Experimental Results 70

***3.4.1 Data* 71**

***3.4.2 Plaintext Performance* 71**

***3.4.3 Results on Encrypted Data* 72**

3.5 Limitations 75

***3.5.1 Activation Functions* 76**

***3.5.2 Optimization* 76**

3.6 Summary and Discussion 77

4 MEMORY EFFICIENT PRIVACY-PRESERVING MACHINE LEARNING BASED ON HOMOMORPHIC ENCRYPTION 78

4.1 Introduction 78

4.2 Related Work 81

4.3 Our Proposed Approach 83

***4.3.1 Batch Packing* 83**

***4.3.2 Modeling the Problem as a Schedule* 84**

***4.3.3 Executing a Schedule* 85**

4.3.4 Cost of a Schedule . . . 87

4.4 Reduced memory schedules . . . 89

4.4.1 Caching by Object Type . . . 91

4.4.2 Full Window Caching . . . 91

4.4.3 Partial Window Caching . . . 92

4.4.4 Column-wise caching . . . 93

4.5 Evaluation and Experimental Results . . . 93

4.5.1 Data . . . 93

4.5.2 Memory Estimate . . . 95

4.5.3 Measurements . . . 97

4.6 Summary and Discussion . . . 108

4.6.1 Application to other layers . . . 108

4.6.2 Optimizing schedules for peak memory usage . . . 109

5 CLASSIFICATION OF ENCRYPTED WORD EMBEDDINGS USING RECURRENT NEURAL NETWORKS . . . 115

5.1 Introduction . . . 115

5.1.1 Threat Model and Problem Statement . . . 118

5.1.2 NLP with Neural Networks . . . 118

5.2 Related Work . . . 119

5.3 The Proposed Privacy-preserving Classification for Recurrent Neural Networks . . . 120

5.3.1 Encrypted word embeddings . . . 121

5.3.2 Noise growth in HE . . . 122

5.3.3 Implementation . . . 122

5.4 Evaluation and Experimental Results . . . 123

5.4.1 Data and Preprocessing . . . 123

5.4.2 Results on Encrypted Data . . . 124

5.5 Summary and Discussion . . . 127

6 NON-INTERACTIVE PRIVACY PRESERVING RECURRENT NEURAL NETWORK PREDICTION WITH HOMOMORPHIC ENCRYPTION . . . 129

6.1 Introduction . . . 130

6.2 Related Work . . . 132

6.3 The Proposed Approach . . . 133

***6.3.1 Threat Model* . . . 133**

***6.3.2 Multiplicative Depth and Noise growth* . . . 134**

***6.3.3 Parallel RNN Blocks* . . . 135**

6.4 Evaluation and Experimental Results . . . 137

***6.4.1 Training with polynomial approximations* . . . 137**

***6.4.2 Data and Preprocessing* . . . 138**

***6.4.3 Plaintext Performance* . . . 138**

***6.4.4 Results on Encrypted Data* . . . 142**

***6.4.5 Comparison with interactive approaches* . . . 144**

6.5 Summary and Discussion . . . 145

***6.5.1 Statistical significance Test* . . . 146**

***6.5.2 Relation to Truncated Backpropagation Through Time* . . . 147**

7 ADDITIONAL RNN-BLOCKS ARCHITECTURES . . . 149

7.1 Averaging block output . . . 150

***7.1.1 Evaluation* . . . 151**

7.2 Hierarchical Blocks . . . 154

***7.2.1 Evaluation* . . . 156**

7.3 Comparing Architectures . . . 159

7.4 Summary and Discussion . . . 164

8 CONCLUSION AND FUTURE WORK . . . 167

8.1 Conclusion . . . 167

8.2 Future Work . . . 168

8.2.1 Future work 1: Addressing the resource overhead 169

8.2.2 Future work 2: Privacy-preserving Neural Networks for Sequence Processing 169

LIST OF TABLES

Table 3.1 Architecture of the Cryptonets for MNIST Model 74

Table 3.2 Architecture of the MNIST small Model 74

Table 3.3 Architecture of the MNIST large Model 74

Table 3.4 Architecture of the CIFAR-10 Model 75

Table 3.5 Architecture of the CIFAR-10 Model with Batch Normalization 75

Table 3.6 Latency and accuracy of all tested models 75

Table 4.1 Architecture of the evaluation model with the layer parameters showing the filter size (FS), stride (S), number of filters (NF), and the activation or pooling function used on plain text (PT) and on encrypted data (HE). . . . 94

Table 4.2 Measurements for **Time** in seconds, Memory (**Mem**) in MB, for all Schedules on Conv 2d (1) with small parameters on the PC wiht 16 and the server with 104 Threads. Additionally, the table shows the memory Estimate in MB and **Score**. 99

Table 4.3 Measurements for **Time** in seconds, Memory (**Mem**) in MB, for all Schedules on Conv 2d (2) with small parameters on the PC wiht 16 and the server with 104 Threads. Additionally, the table shows the memory Estimate in MB and Score. 100

Table 4.4 Measurements for **Time** in seconds, Memory (**Mem**) in MB, for all Schedules on Conv 2d (3) with small parameters on the PC wiht 16 and the server with 104 Threads. Additionally, the table shows the memory Estimate in MB and Score. 100

Table 4.5 Measurements for **Time** in seconds, Memory (**Mem**) in MB, for all Schedules on Conv 2d (1) with medium parameters on the PC wiht 16 and the server with 104 Threads. Additionally, the table shows the memory Estimate in MB and Score. 101

Table 4.6 Measurements for **Time** in seconds, Memory (**Mem**) in MB, for all Schedules on Conv 2d (2) with medium parameters on the PC wiht 16 and the server with 104 Threads. Additionally, the table shows the memory Estimate in MB and Score. 101

Table 4.7 Measurements for **Time** in seconds, Memory (**Mem**) in MB, for all Schedules on Conv 2d (3) with medium parameters on the PC wiht 16 and the server with 104 Threads. Additionally, the table shows the memory Estimate in MB and Score.* values are unavailable because the execution ran out of memory. 102

Table 4.8 **Time** in s, **Memory** in MB requirements, for all Schedules on Conv 2d (1) with large parameters on the PC with 16 and the server with 104 Threads. Additionally, shown are memory **Estimate** in MB and **Score**. 102

Table 4.9 textbfTime in s, **Memory** in MB requirements, for all Schedules on Conv 2d (2) with large parameters on the PC with 16 and the server with 104 Threads. Additionally, shown are memory **Estimate** in MB and **Score**.* indicate out of memory. 103

Table 4.10 **Time** in s, **Memory** in MB requirements, for all Schedules on Conv 2d (3) with large parameters on the PC with 16 and the server with 104 Threads. Additionally, shown are memory **Estimate** in MB and **Score**.* indicate out of memory. 103

Table 4.11 Fastest Schedule for each layer and parameter size (Param.) on the server, 104 Threads (T), and PC, 16 Threads. As well as the increase (Inc.) in time and reduction (Red.) of memory. 106

Table 5.1 Data transferred during encrypted classification 125

Table 5.2 Run times of inference on IMDb test set. 126

Table 6.1 Median F1 score and median balanced accuracy (BA) of the different architectures on different data sets 141

Table 6.2 Resource requirements for encrypted (Ctxt) and plain text (Ptxt) execution 144

Table 6.3 Resource requirements for RNNs using an interactive phase (ia) and our non-interactive approach (nia) 145

Table 7.1 Average F1 score for 5-fold cross-validation of the averaging models with Polynomial Activation and Tanh activation, and a varying number of splits. The length of a subsequence is 64/#Splits. Performed on the Amazon Movie Review Data. 0 splits indicate the baseline RNN model that is not using blocks. 154

Table 7.2 Average F1 score for 5-fold cross-validation of the hierarchical models with Polynomial Activation and Tanh activation, and a varying number of splits. The length of a subsequence is 64/#Splits. Performed on the Amazon Movie Review Data. 0 splits indicate the baseline RNN model that is not using blocks. 158

Table 7.3 Performance comparison of all combination strategies depending on the number of splits. F1 is the average F1 score on the test fold for 5-fold cross-validation. The highest average F1 score of a given number of splits is bold. 162

LIST OF FIGURES

Figure 1.1 Machine learning as a Service with homomorphic encryption for privacy-preservation. 6

Figure 2.1 A simple overview of an RNN layer 2.1a and the "unrolled" RNN layer 2.1b along the time dimension t. The red dotted line shows the initial input value passing x_0 through every unrolled layer 36

Figure 3.1 Internals of a long-short term memory (LSTM) cell with noise-increasing operations highlighted. 48

Figure 3.2 The XLA architecture. 58

Figure 3.3 Python example for simple encrypted execution 61

Figure 3.4 Interaction of the components of TensorFlow and Aluminium Shark . 62

Figure 4.1 Breaking an example base schedule down into multiple sub-schedules. This schedule is executed row-wise. 89

Figure 4.2 Example of how to turn a lock-free execution with three threads into a schedule 90

Figure 4.3 Load instructions that are necessary when moving from the first window to the second using window caching with a 5x5x2 input and a 3x3x2x2 kernel. 92

Figure 4.4 Weights W_i only in Sub-Schedules that correspond to individual filter positions and how they can be reordered to increase caching potential. 92

Figure 4.5 Transforming the base schedule into a column-wise caching schedule . 93

Figure 4.6 Time and Memory requirements schedules, run with large parameters, on the 104 threads servers and the 16 threads PC. The memory graphs also include the PC's memory limit of 18000 MB. 98

Figure 4.7 Comparison of the fastest schedule for each layer with 16 and 104 threads. For each layer, the Figure shows the increase factor in runtime from 104 to 16 threads and the increase factor in memory from 16 to 104 threads for the fastest schedule 99

Figure 4.8 Comparison of the memory requirements at each step for three schedule permutations of the Conv 2D (1) layer. 112

Figure 4.9 Comparison of the memory requirements at each step for three schedule permutations of the Conv 2D (2) layer. 113

Figure 4.10 Comparison of the memory requirements at each step for three schedule permutations of the Conv 2D (1) layer. 114

Figure 5.1 Inference time per batch on the IMDb data 126

Figure 6.1 A standard RNN 135

Figure 6.2 Parallel RNN blocks 136

Figure 6.3 Architecture diagram 136

Figure 6.4 Impact of the number of splits on model performance 142

Figure 6.5 P-values computed by the Mann-Whitney U test on the F1 scores of 5-fold cross-validation. Computed for all model pairs using the concatenation strategy and the two baselines PA None 0 and Tanh None 0. 148

Figure 7.1 The RNN Blocks architecture. The input sequence $x_0, x_1, ...x_l$ is split into i chunks of length n. We process each chunk with an RNN and combine the outputs $y_0, y_1, ..., y_i$ for further processing. 149

Figure 7.2 Averaging the outputs $y_0, \ldots, y_i$ of $i + 1$ RNNs blocks. The input to each block is a subsequence of length n. 150

Figure 7.3 Result of 5-fold cross-validation for RNN blocks with averaging using different numbers of splits and comparing the HE-friendly model (PA) to the non-HE-friendly model (Tanh). 152

Figure 7.4 P-values computed by the Mann-Whitney U test on the F1 scores of 5-fold cross-validation. Computed for all model pairs using the averaging strategy and the two baselines PA None 0 and Tanh None 0. 153

Figure 7.5 Hierarchical RNN Blocks architecture. We stack the output of multiple blocks (2 in this figure) and run them through another block. 155

Figure 7.6 Result of 5-fold cross-validation for hierarchical RNN blocks using different numbers of splits and comparing the HE-friendly model (PA) to the non-HE-friendly model (Tanh). 157

Figure 7.7 P-values computed by the Mann-Whitney U test on the F1 scores of 5-fold cross-validation. Computed for all model pairs using the hierarchical blocks strategy and the two baselines PA None 0 and Tanh None 0. 160

Figure 7.8 Performance comparison of all three RNN block architectures, concatenation (concat), average, and hierarchal. None indicates the baseline vanilla RNN with 0 splits. Tanh and Polynomial Activation (PA) indicate the activation function used. Dotted lines are purely visual dividers for clarity. . . . 161

Figure 7.9 P-values computed by the Mann-Whitney U test on the F1 scores of 5-fold cross-validation. Computed for all model pairs using the concatenation and averaging strategy and the two baselines PA None 0 and Tanh None 0. . 163

Figure 7.10 P-values computed by the Mann-Whitney U test on the F1 scores of 5-fold cross-validation. Computed for all model pairs using the concatenation and hierarchical strategy and the two baselines PA None 0 and Tanh None 0. 165

Figure 7.11 P-values computed by the Mann-Whitney U test on the F1 scores of 5-fold cross-validation. Computed for all model pairs using the averaging and hierarchical strategy and the two baselines PA None 0 and Tanh None 0. . . 166

LIST OF ABBREVIATIONS

CKKS	*Cheon-Kim-Kim-Song*
FHE	*Fully Homomorphic Encryption*
HE	*Homomorphic Encryption*
ML	*Machine Learning*
NN	*Neural Network*
LSTM	*Long Short-Term Memory*
LWE	*Learning With Errors*
PPML	*Privacy-Preserving Machine Learning*
ReLU	*Rectified Linear Unit*
RNN	*Recurrent Neural Network*
RLWE	*Ring Learning With Errors*
XLA	*Accelerated Linear Algebra*

CHAPTER 1
INTRODUCTION AND MOTIVATION

Parts of this dissertation are based on previously published work or work that is currently under review:

- Parts of Chapter 2 and 3 are based on *A survey of deep learning architectures for privacy-preserving machine learning with fully homomorphic encryption*

  Robert Podschwadt, Daniel Takabi, Peizhao Hu, Mohammad H. Rafiei, and Zhipeng Cai in *IEEE Access 10, Pages 117477-117500, 2022* [1]

- Chapter 4 is based on *Memory Efficient Privacy-Preserving Machine Learning Based on Homomorphic Encryption*

  Robert Podschwadt, Parsa Ghazvinian, Mohammad GhasemiGol, and Daniel Takabi; currently under review at *22nd International Conference on Applied Cryptography and Network Security, 5-8 March, 2024*

- Chapter 5 is based on *Classification of Encrypted Word Embeddings using Recurrent Neural Networks*

  Robert Podschwadt and Daniel Takabi in *PrivateNLP 2020: Workshop on Privacy in Natural Language Processing, 2020* [2]

- Chapter 6 is based on *Non-interactive Privacy Preserving Recurrent Neural Network Prediction with Homomorphic Encryption*

Robert Podschwadt and Daniel Takabi in *IEEE 14th International Conference on Cloud Computing (CLOUD), Pages 65-70, 2021* [3]

In reference to IEEE copyrighted material which is used with permission in this thesis, the IEEE does not endorse any of Georgia State University's products or services. Internal or personal use of this material is permitted. If interested in reprinting/republishing IEEE copyrighted material for advertising or promotional purposes or for creating new collective works for resale or redistribution, please go to `http://www.ieee.org/publications_standards/publications/rights/rights_link.html` to learn how to obtain a License from RightsLink. If applicable, University Microfilms and/or ProQuest Library, or the Archives of Canada may supply single copies of the dissertation.

1.1 Motivation

Machine Learning (ML), neural networks (NN) in particular, require large amounts of data and computational resources to train. This makes high-performing models a valuable asset. If a provider (server) who owns the model wants to offer inference to other parties (clients), it is not in its interest to give the model to them to use it. Giving the clients unrestricted access to the model would reveal the server's secret to them. Additionally, the client might not possess the necessary computational resources to run the NN. On the other hand, the client does not want to reveal its data to the server. The client has no control over what the server does with the data. For example, services like Amazon Alexa[1] or Google Assitant[2]

[1] `https://www.amazon.com/alexa-smart-home/`
[2] `https://assistant.google.com/`

send the user voice commands to the server where they are processed and often stored, as highlighted in this Washington Post article [4]. The server can sell the data to third parties without any control of the user.

Or it can mine keywords to use in advertising. Even if the server acts ethically and does not sell the data, it might be forced to reveal the data to law enforcement. For example, this can become a problem for women who use online fitness and healthcare apps to track their periods. With the recent ruling against Roe v. Wade by the U.S. Supreme Court, this data can put them in legal jeopardy [5]. Access to this data might allow law enforcement to determine if a woman terminated a pregnancy, which could have serious legal consequences if she lives in a state that outlawed abortion. But not only period trackers are problematic. Even location data and online search history can be incriminating [6]. Furthermore, it is always possible that the server might not share the data for profit or because it is required by law knowingly. There is always the possibility of a data breach, and highly privacy-sensitive data can be stolen from the server.

These are only a few of many examples of how our data is collected and the consequences of it falling into the wrong hands. With an increasing number of data collection devices and services in our lives, we need to ensure that the data is adequately protected to avoid negative consequences. One way to protect the data is to encrypt it before it leaves our control. However, encryption typically prevents the server from running analytics on the data. To protect the privacy of the data and still benefit from ML models, researchers have investigated numerous techniques to address this problem. One technique that is promising is

homomorphic encryption (HE). HE schemes are encryption schemes that allow computation on the encrypted data without decrypting it. The result of the computation and all intermediate values are also encrypted, making it ideal for outsourced computation. The server never learns anything about the client's data (Fig. 1.1). Researchers investigated other techniques for privacy-preserving ML, such as Differential Privacy [7], Secure Multiparty Computation [8], and Functional Encryption [9, 10].

Differential Privacy adds random noise to the data, obscuring private information in the data. Based on the amount of noise added, a probability bound on the information leakage can be computed, indicating how likely an adversary can extract private information from the data. The main application of Differential Privacy is to protect personal information in the training data. The goal is to prevent an attacker from extracting private information from the trained model by controlling how much each sample can influence the parameters during training [11]. Multiple parties can also use it to train a shared model on distributed data without disclosing private data to other participants [12]. The primary strength of Differential Privacy is that it operates on standard numerical data types, meaning most ML libraries and hardware accelerators are supported, making the implementation and run-time overhead negligible.

On the other hand, using Differential Privacy reduces the quality of the model predictions. Differential Privacy does not offer any cryptographic guarantees and only offers a probability bound on possible information leakage. In contrast, HE relies on provably secure hardness assumptions. Furthermore, Differential Privacy does not protect the data at inference time.

Secure Multiparty Computation (SMC) is a term for numerous algorithms and protocols, like garbled circuits [13], secret sharing [14], and oblivious transfer [15], that allows two or more parties to jointly evaluate a function without revealing their inputs to each other. With these primitives, researchers have developed protocols to evaluate NNs while preserving the privacy of the inputs [16, 17, 18]. However, these protocols often require a lot of communication during the computation, which can make network latency a problem. It is possible to avoid some communication by performing expensive pre-computation in an offline phase [19, 20]. In addition to the high bandwidth requirements, SMC protocols allow the client to infer some of the model architecture. Since the client needs to be involved in most of the computation, with HE, the computation can be performed without any client interaction. However, neither technique protects against model inversion/stealing attacks, like Tramer et al. [21].

Functional Encryption [9, 10] is a form of encryption that allows the evaluation of certain functions over encrypted data. The results of these functions are "leaked" from the ciphertexts, i.e., the result of the function is in plain data. This is beneficial for NN computation since only the first layer needs to be run on encrypted data [22, 23], and the rest can be run on plain data. However, this does leak some client data information to the server; the server learns the model output and the intermediate results of the computation. A malicious server could use this information to reverse the client's inputs.

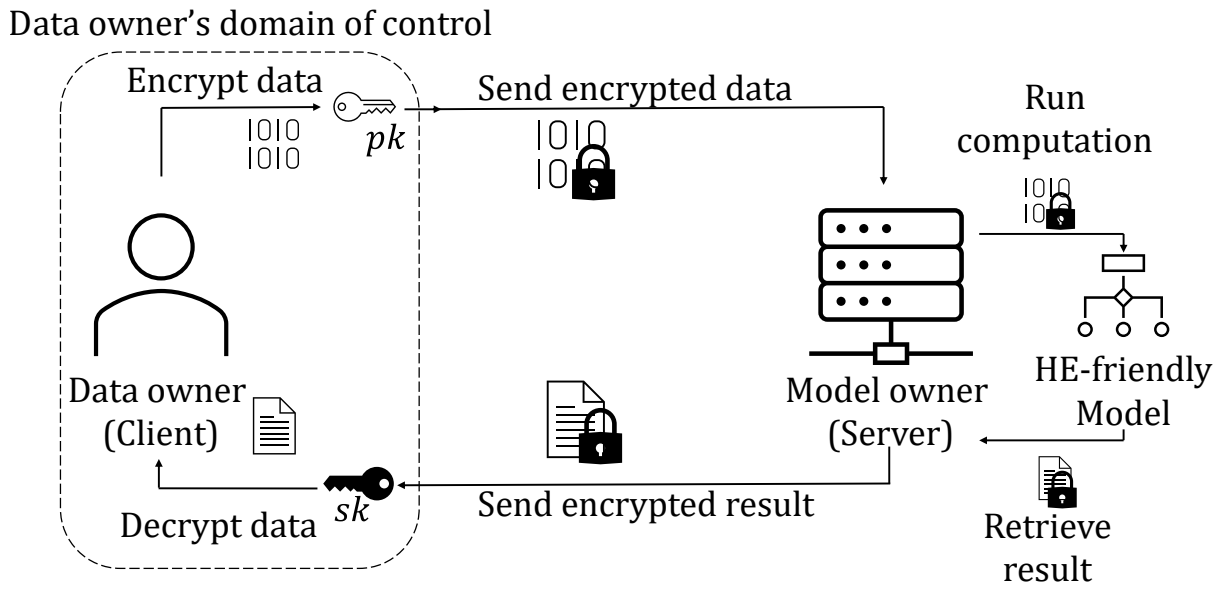


Figure 1.1: Machine learning as a Service with homomorphic encryption for privacy-preservation. (Image credit [1])

1.2 Challenges of Deep Learning with Homomorphic Encryption

While HE offers the best protection for both the client data and the server-side model, it places restrictions on the computation on encrypted data. The most important restrictions are (1) reduced supported operations, (2) often a limited number of layers in the network, and (3) increased resource requirements. Getting neural networks to work within limited operations has been well-studied for simple feed-forward networks. However, other network architectures, like recurrent neural networks (RNNs), have received little attention. Furthermore, running a neural network on encrypted data often requires implementing the ML algorithms from scratch using HE primitives. On plain data, these implementations are readily available and highly optimized, making them easy to use and highly performant.

1.2.1 Limited Supported Operations

We can group current HE schemes into two categories. Schemes that work on numbers and schemes that work on bits. Schemes that work on individual bits like TFHE [24] can only evaluate boolean circuits. This makes them more expressive than schemes that work on numbers since most computations can be broken down into a boolean circuit. It is used, for example, in Google's general-purpose HE transpiler [25]. However, they are slower on numeric operations since these need to be broken down into individual binary gates as well. The majority of computation in machine learning models is numeric. The layers in neural networks consist mainly of matrix multiplication, which can be broken down into addition and multiplication. Addition and multiplication are much faster in the numeric schemes. However, these are the only operations they support, limiting the functions they can evaluate to polynomials. This limitation poses problems for certain parts of the network. In particular, activation functions can not typically be expressed as polynomials. However, they are necessary for the network to learn non-linear functions. Throughout this work, we refer to none polynomial activations that are frequently used on plain data, like ReLU, Tanh, Sigomid, etc., as traditional activation functions. The most common approach is to replace the activation functions with polynomial approximations [26, 27, 28]. There are two main approaches for the use of polynomial activation functions. Note throughout this work, when discussing polynomial activation functions, we exclude the cases of constant (degree zero) and linear (degree one) polynomials unless specifically stated otherwise. (1) Find an approximation that with so little error compared to the original activation function that it

can be used as a drop-in replacement. This way, the network is trained using the traditional activation function, and the polynomial activation is used during private inference. The advantage of this approach is that the model does not need to be retrained or fine-tuned to use the polynomial activation. However, it typically requires a much more precise approximation [28]. Any function can be approximated with arbitrary precision using a polynomial with a large enough degree. However, to keep computational overhead low, we also want to keep the degree of the polynomial low. The other approach (2) is to train the model from scratch or retrain an already trained model using a polynomial approximation. In this case, the approximation does not need to be as precise, typically allowing for a lower-degree polynomial. Training a model with polynomial activation functions poses challenges. The foremost challenge is that the absolute values of both the polynomial and the derivative tend towards infinity. Like traditional activation functions, polynomial activations are also often designed to work best in a small interval around zero. However, unlike traditional activation functions, which typically have bounded derivates, polynomials do not. This can lead to unstable training and easily leads to exploding gradients[29].

Other operations and layers are also affected by the reduced operations that HE supports. For example, since comparison is not supported, Max-Pooling is often replaced with Average-Pooling.

1.2.2 Limited Model Depth

Not only are we limited in the type of layers that we can use, but we are also limited in the number of layers that we use, i.e., the depth of the network. The limiting factor is the

number of multiplications we can perform on ciphertexts. In HE, encryption adds some small noise to the data. When we add two ciphertexts, the noise inside is also added, and when we multiply two ciphertexts, the noise is also multiplied. If the noise reaches a certain threshold, we can no longer decrypt the data correctly. This limits the number of multiplications we can perform in a network. The limited depth is another reason we want to keep the polynomial degree low.

1.2.3 Resource Overhead

In the homomorphic encryption scheme we use, ciphertexts and plaintext are represented as multiple polynomials. These polynomials require several orders of magnitude more memory than the values they represent. Additionally, the addition and multiplication algorithm for these polynomials is more time-consuming than the addition of integers of floating point numbers. So, even if we have a model that is not too deep and consists only of operations supported by the scheme we use, it still might be too computationally expensive or require too much memory for us to compute.

1.3 Contributions

In this dissertation, we investigate and propose solutions for several of the issues outlined above. We focus on the resource requirements of batch-packed neural networks, convolutional neural networks (CNN) specifically. We further investigate recurrent neural networks over encrypted data. Here, the limited model depth poses additional challenges. Throughout this dissertation, we make the following contributions:

- We design and implement a system that allows us to run TensorFlow models over encrypted data. The system is realized as an XLA compiler backend. It allows us to perform privacy-preserving inference directly from Python with little code overhead while providing us with a framework to implement low-level optimizations.

- We investigate the memory requirements of convolutional layers on batch-packed encrypted data and propose a schedule representation for them. The schedule allows us to estimate the memory requirements, giving different caching schemes. We further propose and evaluate strategies for recording the schedule to increase the caching potential of a schedule, which reduces the memory requirements while limiting the time overhead.

- We investigate recurrent neural networks and their applicability to encrypted data. We find that their structure creates very deep computation. To address the depth, we evaluate an interactive approach to reduce the noise during computation.

- We further propose and evaluate a new architecture, RNN blocks, that uses recurrent elements but has a significantly lower depth than a purely recurrent neural network. We evaluate three variants of the RNN blocks architecture with regard to their performance, depth, time, and memory overhead.

1.4 Organization

This dissertation is organized as follows. In Chapter 2 we discuss the required theoretical background and related prior work. In Chapter 3, we introduce our system for private in-

ference and evaluate it against similar work. Chapter 4 presents our scheduling and caching solution for convolutional neural networks. In Chapter 5, we investigate the computational depth of RNNs and propose an interactive solution to reduce the noise during computation. We expand upon this idea in Chapter 6 where we present and evaluate a novel architecture that does not require interactive phases. Chapter 7 expands the analysis of the new architecture by presenting and evaluating two variations of it. We discuss future research directions and conclude in Chapter 8

CHAPTER 2
BACKGROUND

Most work in PPML uses one of three main approaches: Differential Privacy, Secure Multiparty Computation, or Homomorphic Encryption. Here, we briefly describe approaches to outsourced deep learning using Differential Privacy and Secure Multiparty Computation before discussing Homomorphic Encryption in more detail. Differential Privacy [7] adds random noise to the data, obscuring private information while still allowing statistical analysis and machine learning algorithms to work on the data. The biggest strength of Differential Privacy is that it can operate on standard numerical data types, making the integration into existing ML frameworks comparatively easy. Differentially private algorithms are available for TensorFlow in TensorFlow Privacy [30], and for PyTorch in Opacus [31].

Secure Multiparty Computation refers to a collection of algorithms and protocols, like garbled circuits [13], secret sharing [14], and oblivious transfer [15], that enable multiple parties to evaluate a function without revealing their data jointly. Previous work has shown how to evaluate neural networks based on these algorithms and protocols. Works like MiniONN [18] have created privacy-preserving neural networks. Secure Multiparty Computation solutions are communication-bound, making network latency and bandwidth potential bottlenecks. Furthermore, these systems often require parties to be non-colluding to guarantee privacy. Secure Multiparty Computation implementations are available for Tensorflow with systems like CrypTFlow [32] and for PyTorch with CyrpTen [33].

2.1 Homomorphic Encryption

The content of this section, apart from 2.1.2, was previously published in Podschwadt et al. [1] and slightly edited.

Public-key (asymmetric) encryption schemes [34] use separate keys for encryption and decryption. The key used for encryption is called the public key pk, and the key used for decryption is called the private or secret key sk. The public key can be freely shared with anyone and be used for encryption, but only the holder of the secret key can decrypt the message. Public-key schemes are designed in such a way that having access to the public key does not allow an attacker to extract the private key [35]. In addition to being public-key schemes, HE schemes allow computation on encrypted data. The result of the computation is also encrypted; the data does not need to be decrypted during the computation. Having all the stages of the computation encrypted, from the inputs over the intermediate values to the results, makes HE schemes ideal for outsourced computation. After decryption, the result of the encrypted computation is the same as if the computation was performed on plain data [36].

For encryption, we start with a message $m \in \mathcal{M}$. $\mathcal{M}$ is called the message space in this paper. Common message spaces are integers $\mathcal{M} = \mathbb{Z}$, real numbers $\mathcal{M} = \mathbb{R}$, or single bits $\mathcal{M} = \{0, 1\}$. To encrypt m, it needs to be encoded into a plaintext p, which comes from the plaintext space $\mathcal{P}$. The encoding of m into p is done by the encoding function encode and the reverse operation decode. p can then be encrypted into a ciphertext c. The transformation from a plaintext into a ciphertext is done by the encryption function Enc. It uses the public

key to encrypt the plaintext p into the ciphertext c:

$$c = \text{Enc}(\text{pk}, p) \tag{2.1}$$

The decryption operation is performed by the decryption function Dec, which uses the secret key sk to turn a ciphertext c back into a plaintext p:

$$p = \text{Dec}(\text{sk}, c) \tag{2.2}$$

Where HE schemes differ from other public-key schemes is that they also have an evaluation function Eval that can evaluate a circuit C, a sequence of operations. E.g., a circuit can be an ML model. Evaluating C on plain data and on encrypted data gives us the same result after decryption:

$$\text{Dec}(\text{sk}, \text{Eval}(pk, C, c_0, \ldots, c_n)) = C(p_0, \ldots, p_n) \tag{2.3}$$

where $c_0, \ldots, c_n = \text{Enc}(\text{pk}, p_0, \ldots, p_n)$ are the encryptions of the plaintexts $p_0, \cdots, p_n$.

We can categorize HE schemes by the operations that can be used in the circuit C and the depth of C. The depth of a circuit is the number of consecutive operations that need to be performed to evaluate the circuit. Here, we provide brief explanations of the HE schemes categorization established by Armknecht et al. [37]:

Partially Homomorphic Encryption schemes are the "simplest" HE schemes; they only support a limited set of circuits since they can only evaluate either addition or multi-

plication on encrypted data [38, 39].

Somewhat Homomorphic Encryption schemes can support both multiplication and addition on encrypted data. However, the size of the ciphertexts grows with every computation performed. The depth of the supported circuits can be controlled by the encryption parameters [40].

Leveled Homomorphic Encryption schemes are very similar to somewhat homomorphic schemes in definition. However, their schemes require that the size of the ciphertexts does not grow as operations are performed. The depth of the circuits that can be evaluated using Leveled Homomorphic Encryption schemes can be controlled with a parameter L. The size of the ciphertexts needs to be independent of L and only rely on the security level. Leveled Homomorphic Encryption schemes that support both addition and multiplication are sometimes called leveled fully homomorphic [41].

Fully Homomorphic Encryption (FHE) schemes have no restrictions on the circuits they can evaluate. That means they need to be able to evaluate circuits of arbitrary depth and have no restrictions regarding the operations required.

The limited circuit depth stems from the way encryption works. In simple terms, encryption adds noise to the plaintext, thereby obscuring it. The decryption process removes that noise. Every operation that is performed on encrypted data increases the noise inside the ciphertexts. If the noise passes a certain threshold, correct decryption becomes impossible [42]. One solution is to decrypt a ciphertext c and then encrypting it again, leading to a fresh ciphertext c' with a reset noise level. However, this requires access to the secret key sk.

To address this issue, Gentry [43] describes how to build an FHE scheme out of a scheme that can evaluate its decryption function on encrypted data by proposing a bootstrapping method:

$$c' = \text{Eval}(\text{pk}, \text{Dec}, \text{Enc}(\text{sk}), c) \tag{2.4}$$

In Eq. 2.4, the decryption circuit, $C = \text{Dec}$, is evaluated with an encryption of sk as input. This does not completely reduce the noise inside the ciphertext but reduces the noise level; hence, further computation can be performed. The bootstrapping operation can be performed as often as necessary, allowing HE schemes to evaluate circuits of arbitrary depth.

However, bootstrapping relies on circular security, i.e., the secret key needs to be encrypted with its corresponding public key [44]. Circular security can be eliminated by using keyswitching [45, 46]. Keyswitching uses a chain of different secret keys where each key is encrypted with the following key in the chain. It evaluates the bootstrapping function using one of the keys from the chain, resulting in the ciphertext c' being encrypted with the next key in the chain. However, once all keys in the chain have been used, no further computation can be performed. Additionally, bootstrapping is computationally expensive, which is why it is often not used in practice. We refer the reader to Brakerski [45] for a complete discussion of FHE and to Armknecht et al. [37] for more in-depth information about the different HE scheme types.

2.1.1 Single Instruction Multiple Data

Single Instruction Multiple Data (SIMD) [47] is a form of parallel processing. It can be applied to encrypted data in some HE schemes [48]. SIMD operations can reduce the computational overhead introduced by encrypted computation. SIMD allows the user to encode multiple messages into a single plaintext. Operations on the plaintext (or a ciphertext that encrypts it) are performed on all messages encoded within it. The number of messages a ciphertext can hold is called slots. The number of filled slots does affect the complexity of the operations; they have the same complexity if one or all slots are filled. The encryption parameters govern a ciphertext's number of slots.

2.1.2 The CKKS Scheme

In this work, we use the CKKS scheme since its support for real numbers makes it ideal for ML. In this section, we briefly summarize the fundamentals of the CKKS scheme, explained in more detail by Cheon et al. [49]. We have three types of objects in CKKS: the message m, the plaintext p, and the ciphertext c. The relationship between the three is as follows: by using an encoding encode we can turn a message m into a plaintext p. Using the encryption function Enc we can encrypt p into a ciphertext c. In reverse, we decrypt c using the decryption function Dec to turn it back into a plaintext, which in turn we can decode back into a message using the decode function. Let N be an integer power of two and $M = 2N$. Let $\Phi_M(X) = X^N + 1$ be the M-th cyclotomic polynomial and $\xi_M = e^{2i\pi/M}$ the M-th root of unity. With this, we can define σ as the canonical embedding $\mathbb{C}[X]/(X^N + 1) \rightarrow \mathbb{C}^N$, which

is given as the evaluation of the polynomial p at the N roots $\xi, \xi^3, \cdots, \xi^{2N-1}$. This allows us to decode a polynomial into a vector. The reverse operation, encoding, σ^{-1} relies on solving the system

$$\sum_{j=0}^{N-1} \alpha_j (\xi^{2i-1})^j = m_i, i = 1, \ldots, N \tag{2.5}$$

where α_j are the polynomial coefficients and m_i are the elements of the vector. The message space is the space of complex vectors of length $N/2$, $\mathcal{M} = \mathbb{C}^{N/2}$. The plaintext space $\mathcal{P}$ is the integer polynomial ring $\mathcal{R} = \mathbb{Z}/(X^N + 1)$. The encoding function is given as:

$$\text{Encode}(m) = \sigma^{-1}(\lfloor \Delta \pi^{-1}(m) \rceil_{\sigma(\mathcal{R})}) \tag{2.6}$$

where π^{-1} is the function that expands m from $\mathbb{C}^{N/2}$ to $\mathbb{C}^N$, Δ is the scaling factor, and $\lfloor \cdot \rceil$ is coordinate-wise random rounding described by Lyubashevsky et al. [50]. The scaling factor, Δ, prevents the rounding operation from impacting significant parts of the numbers. The decoding function from the polynomial p to the vector m is

$$\text{Decode}(p) = \pi(\sigma(\Delta^{-1} p)) \tag{2.7}$$

Using this decoding and encoding, we can pack a complex vector into a polynomial plaintext p. To encrypt the plaintext, we first need to choose a coefficient modulus q. When then sample sample three polynomials a, sk, and e from the polynomial ring $\mathcal{R}_q = \mathbb{Z}_q[X]/(X^N + 1)$. a is sampled uniformly and, e is typically Gaussian. sk is the secret key

polynomial, and e is a small error polynomial. For details of sampling polynomials, we refer the reader to [49] [51]. We now construct pairs of polynomials $(a, a \cdot sk + e)$. We can informally state the (decision) ring learning with error problem [51] as follows: Given a list of polynomial pairs, is the second part of the pair constructed as $a \cdot \mathrm{sk} + e$ or randomly generated. From this, we construct the public key $\mathrm{pk} = (-a \cdot s + e, a)$. Using pk we can encrypt a plaintext using the following equation:

$$\mathrm{Enc}(p) = (p, 0) + \mathrm{pk} = (p - a \cdot s + e, a) = (c_0, c_1) \tag{2.8}$$

The decryption of a ciphertext c is given as:

$$\mathrm{Dec}(c) = c_0 + c_1 \cdot s = p - a \cdot s + e + a \cdot s = p + e \approx m \tag{2.9}$$

If the error is small enough, the output of the decryption will be close to the true value p. The addition of ciphertexts is straightforward. Given two ciphertexts c and d, which are the encryptions of p_0 and p_1, respectively, that we want to add, we simply compute:

$$\mathrm{Add}(c, d) = (c_0 + d_0, c_1 + d_1) \tag{2.10}$$

By plugging this into Eq. 2.9 we can see that the result is $p_0 + p_1 + 2e$. Again, since we require the error to be small, the result is approximately correct. Multiplication is more complex since, during multiplication, the ciphertext expands from two to three polynomials.

$$\mathrm{Mult}(c, d) = (c_0 d_0, c_0 d_1 + d_0 c_1, c_1 d_1) \tag{2.11}$$

To get it back down to two, we can use relinearization [52]. Relinearization requires a relinearization key (sometimes called evaluation key), $\text{rk} = (-a_0 s + e_0 + s^2, a_0)(\mod vq)$. e_0 is a small random polynomial, and a_0 is polynomial uniformly sampled from $\mathcal{R}_{vq}$ and v a large integer. We perform relinearization of the product of two ciphertexts c and d as:

$$\text{Relin}(\text{Mult}(c, d), \text{rk}) = (c_0 d_0, c_0 d_1 + d_0 c_1) + \lfloor (c_1 d_1 \text{rk})/v \rceil \tag{2.12}$$

During multiplication, the scale Δ inside grows quadratically. To perform multiple operations without it becoming too large and overflowing, we use an operation called rescaling. Let L be the number of multiplications we want to perform, called level. A ciphertext starts at level L and moves down one level with each rescaling operation. Once we are at the lowest level, we can not perform any more multiplications. Let $p_1, \ldots, p_L, q_0$ be prime numbers where each $p_l \approx \Delta$ and $q_0 > \Delta$, and $q_L = \prod_{l=1}^{L} p_l q_0$. The rescaling operation of a ciphertext c from level l to $l-1$ is then defined as:

$$\text{Res}(c, l, l-1) = \lfloor c/p_l \rceil \mod q_{l-1} \tag{2.13}$$

The selection of N and q has security implications. Simply speaking, the security relies on the ratio N/q. We want q to be large enough to handle the required number of multiplications. However, to remain secure as q grows, so must N. The exact ratio relies on many factors, such as the key and error distribution. Selecting appropriate values to instantiate the scheme can be done by referring to the Homomorphic Encryption Security Standard

[53], using tools like LWE Estimator [54], or it is often built into crypto libraries like SEAL [55] or OpenFHE [56]. CKKS additionally supports rotations that can move the slots in a ciphertext with roll semantics and bootstrapping as proposed by Cheon et al. [57]. However, since we use neither operation in this dissertation, we omit them here.

2.2 What is an HE-friendly model?

In this work, we perform privacy-preserving computation of machine learning models using HE. Since HE places restrictions on the operations we can feasibly perform, we use the term HE-friendly to refer to the operations we can perform. Since our work uses the CKKS scheme, we use its capabilities to define the term HE-friendly. We say an operation is HE-friendly if it consists of only addition, multiplication, or division with a plaintext divisor. We can extend this definition to an ML model. We call a model HE-friendly if it only consists of HE-friendly operations. In addition to the supported operations, we also include a complexity component in the term HE-friendly. Even if a network consists of only HE-friendly operations, it might be too complex, e.g., consisting of too many layers, for us to consider it HE-friendly. Models with too many layers require bootstrapping, further increasing the computational overhead.

2.2.1 What is required to make a model HE-friendly?

The fundamental building blocks of many models, neural networks especially, are already HE-friendly. Most of these consist of matrix and vector multiplication and addition. These are operations that can easily performed over encrypted data. In this dissertation, we work with neural networks and limit the following discussion to the components of HE-friendly

neural networks. There are two ways to create an HE-friendly network. The first way is to design and train the model from scratch. However, this can be hard since HE-friendly models often use polynomials as activation functions, and training with a model with polynomials can quickly lead to exploding gradients and put the training in an unrecoverable state. The second approach relies on training and a model that is not HE-friendly and transforming it into an HE-friendly one.

The transformation gradually replaces unsupported operations and finetunes the model after each replacement. We will now describe proposed approaches and techniques for HE-friendly neural network components. The remainder of Section 2.2.1 was previously published in Podschwadt et al. [1] and has been edited only slightly.

2.2.1.1 Fully Connected and Convolutional Layers

Fully Connected layers are the simplest forms of NN layers. The fundamental building block is a neuron or unit. Given inputs x_i, weights w_i, a bias b and an activation function f, an output, $\hat{y} = f(b + \sum_{i=1}^{n} w_i x_i)$, is computed. Instead of computing the output of every neuron individually, for the neurons of a layer, the output values can be computed using simple matrix multiplication.

Unlike fully connected networks, CNNs often include various NN layers, such as convolution, pooling, and batch normalization. Fundamentally, fully connected and convolutional layers are similar. In a convolutional layer, a window is moved across the input data. Each cell in the window has a weight associated with a weight. The output of the window is the weighted sum of all elements in the window. For multiple channels, the window is moved

across all channels, and the output of each cell is summed up along the channel dimension . Like fully connected layers, convolution layers can be expressed as matrix multiplications and dot products. Matrix multiplication and dot products are HE-compatible since they only consist of additions and multiplications. No work needs to be done to adapt them to HE. However, a naive implementation can be costly; every element of a matrix/vector is a single ciphertext. The naive implementation of d-dimensional matrix multiplication requires d^3 multiplications, and the dot product of two d-dimensional vectors requires d multiplications.

One strategy for speeding up computation is to reduce the number of operations needed. SHE [58] replace matrix multiplication and dot product with shift operation to reduce the number of multiplications. When a number is represented in binary format, as in SHE, multiplying it by two can be achieved by shifting the decimal point right by one. In fact, multiplying by any power of two is simply shifting the decimal point. Shift operation requires quantizing the NN's weights to be a power of two and representing the NN's inputs as fixed-point numbers in their binary representation. Faster CryptoNets [59] uses a very similar approach. The authors also quantize the NN's weights to be a power of two, giving them a sparse polynomial (mononomial) representation. Multiplication between ciphertexts and these sparse plaintexts is much faster. Research on plaintext [60] suggests that weight quantization does not sacrifice accuracy.

Another option is using the SIMD operations offered by many schemes. The goals are to use fewer ciphertexts, decrease latency, and organize the data to speed up computation. This organization is called ciphertext packing. It is essential to have efficiently packed ci-

phertexts since it can drastically reduce the number of operations needed to evaluate specific algorithms. For example, E2DM [61] includes an encoding map that allows packing one or more matrices in a single ciphertext for efficient matrix multiplications. PrivFT [62] uses a packing structure that encodes the input in a relatively small number of ciphertexts. For efficient dot products, the embedding matrix is packed vertically. Brutzkus et al. [63] propose packing schemes for efficient convolutions and matrix multiplications. The packing schemes are memory efficient since they use fewer ciphertexts. The authors propose algorithms for switching between packing schemes depending on the operation, i.e., convolutions or matrix multiplications. Lee et al. [28] propose a packing scheme for strided convolutions. In contrast to E2DM and PrivFT, the CHET compiler [64] can automatically select packing schemes depending on the input model, HE scheme, and data domain by introducing a cost model. Mihara et al. [65] develop a diagonal ciphertext packing scheme for efficient matrix transpose operation; it boosts the training backpropagation speed.

The primary challenge of ciphertext packing schemes is that they need to work efficiently with data for operations such as convolutions or matrix multiplications; there is no unique packing scheme for all data/operations. Besides, except for CHET [64], other techniques such as E2DM and PrivFT require trial and error to identify efficient ciphertext packing schemes. Furthermore, packing often requires rotations of the slots. Switching packing schemes, which often requires rotations, impacts the overall NN runtime. Additionally, the transformation from one representation to another needs to be considered when calculating the noise budget.

2.2.1.2 Pooling Layers

The output of the window is the pooling function, which is applied to all values in the window. The most common pooling layer is max-pooling, where the pooling function is the maximum of all values in the window. However, it is inefficient with HE schemes except SHE [58]; it benefits from the TFHE scheme. Some studies replace the max-pooling operation with either average [66, 64, 67] or scaled (by a constant factor) average pooling [26, 68, 27, 59, 69]. The most common scaling factor is the number of input elements; it turns average pooling into input summations without any multiplication (i.e., plaintext division) operation. The advantage of using average pooling is the tighter control over the magnitude of the values. The parameters of the HE scheme need to be chosen so that all values that occur during computation fit into the plaintext space. With average pooling, the output of the pooling operations is in the same range as the inputs. With scaled average pooling, the output values can grow larger. The cost of using average pooling is an additional multiplication. Al-Badawi et al. [67] show that pooling is not necessary on simple data sets such as MNIST, but on more complex data sets such as CIFAR-10, it does improve performance. Usually, studies that utilize MNIST do not implement pooling layers in their NNs.

2.2.1.3 Batch Normalization Layers

Batch normalization stabilizes the training process on plain data by reducing the internal covariate shift. It forces the inputs to a layer to follow a zero-mean normal distribution. This normalizing operation makes inputs with large absolute values less likely. Chabanne

et al. [66] used batch normalization to stabilize NN training with polynomial activation functions. Polynomial activations usually work well in a small interval around zero. Outside of that interval, they are unbounded and have rapidly growing derivatives during training (exploding gradient problem). By placing a batch normalization before a polynomial activation layer, the probability of encountering values outside the optimal polynomial activation range decreases. In addition to stabilizing training, it helps prevent values from overflowing the limits of the plaintext space. This approach is used in many studies [26, 68, 59, 58, 69].

2.2.1.4 Recurrent Layers

There is little work on PPML using recurrent layers. Simple recurrent layers (Fig. 2.1a) at their core are similar to fully connected layers; they only consist of matrix multiplication. The depth of recurrent layers depends on the number of elements in the input sequence (Fig. 2.1b). Recurrent layers often have higher multiplicative depth (i.e., more noise), requiring numerically larger crypto parameters than convolutional or fully connected layers. Noise buildup prevents the network from completely processing the inputs. Large crypto parameters require more computation and memory resources. To alleviate these challenges, Podschwadt & Takabi [2] (discussed in greater detail in Chapter 5) and Bakshi & Last [70] (CryptoRNN) propose to use the client to remove the built-up noise in the ciphertext. CryptoRNN uses simple RNNs with client communication, too. They refresh noise at three predefined points in the networks: 1) after every multiplication, 2) before every non-linear activation function, and 3) after processing each sequence element. It should be noted that at point (2), the client also computes the activation functions. Similarly, Podschwadt & Tak-

abi [2] use naive matrix multiplication and a degree three polynomial Tanh approximation in simple RNNs. The authors dynamically decide when to use the client for interactive noise removal depending on the remaining noise budget (which prevents unnecessary communication) instead of at predefined points.

In a later study, Podschwadt & Takabi [3] (discussed in greater detail in Chapter 6) propose RNN Blocks, an RNN architecture that does not need client interaction. It reduces the multiplicative depth by splitting each instance into multiple shorter chunks across the time dimension. Any recurrent operation on these chunks is parallelized, decreasing the multiplication depth significantly due to shorter chunks. The RNN produces an output for each chunk; they are then concatenated and fed to a fully connected layer.

Later recurrent layers such as long short-term memory [71] (LSTM) (Fig. 3.1), or gated recurrent units [72] (GRU), impose even higher multiplicative depth due to their complex structure. SHE [58] modified the LSTM structure to use the ReLU function, which is comparatively easier to compute using TFHE than Tanh. Typically, RNNs use the Tanh function, which requires at least a polynomial of degree three for an adequate approximation. The authors also replace all the multiplications with shift operations, which they can perform at no cost to the noise budget.

Jang et al. [73] implement GRUs on encrypted data using their proposed encryption scheme MatHEAAN. The internal structure of a GRU is similar to that of an LSTM and introduces high multiplicative depth. To address this issue, the authors rely on bootstrapping during the computation of the network.

RNN architectures are not well suited for execution over homomorphically encrypted data [3]; they require either fundamental changes as seen in SHE [58], and RNN Blocks [3], bootstrapping [73], or establishing client communication as seen in [70] and [2]. However, interactive communication takes away some HE key advantages over Secure Multiparty Computation, such as the lower communication overhead and the ability to perform the computation independently from the client.

The primary strength of RNNs is that they have some "memory" of previous states. However, it is their downfall on encrypted data since this "memory" produces networks with a large multiplicative depth.

2.2.1.5 Embedding Layers

Word embeddings are real-valued vector representations of words' meaning in natural language processing; they are useful tools since they turn words into real values (instead of integers) to be later fed to NNs. There are different word embedding methods. PrivFT [62] relies on the fasttext [74] architecture, where the words need to be encoded into "one-hot" vector representations. In a one-hot vector representation, all elements are zero except one. In PrivFT, the client does the one-hot encoding. On plaintext, the translation from a word to one-hot encoding would happen on the server. However, this operation is prohibitively costly over encrypted data. The client sums up all the one-hot encodings, encrypts the results, and sends the ciphertext to the server alongside the number of encoded words. The client transmits this number of words in plain. The server multiplies the vector with the embedding matrix and scales the results with the number of encoded words. Podschwadt &

Takabi [2, 3] also work with textual data. In their work, they outsource the embedding to the client. The client performs the embedding operation (a simple table lookup) and sends the encrypted result to the server for computation.

In all approaches above, the server needs to share some information about the model and the data with the client. They all need to provide the client with an enumerated vocabulary of all the words known to the model. This only leaks very little information. In Podschwadt & Takabi [2, 3] studies, the client also needs to access the learned embeddings in plaintext. This plaintext access is not a problem in their later work RNN Blocks[3], where they use publicly available, pretrained embeddings. However, using publicly available embeddings prevents the server from using fine-tuned or self-learned embeddings and keeping them secret from the client. Besides, embeddings further increase the size of the data. Embeddings often turn a single word into a vector of 100+ real values, increasing size during encryption.

2.2.1.6 Activation Functions

PPML approaches are often different from standard ML in their choice of activation functions. The choice of activation often depends on the scheme. With some HE schemes, one can use popular NN activation functions. For example, SHE [58] uses the TFHE scheme, enabling the user to implement ReLU on encrypted data using binary gates. Bourse et al. [75] use a step function that outputs either 1 or -1 based on the sign of the input value. This sign function can be evaluated using a custom bootstrapping operation in TFHE. Every activation function evaluation is also a bootstrapping operation, allowing unlimited deep networks. However, training a network with the step function is difficult since it does

not provide gradient information. Another option for using standard activation functions is client-side computation. CryptoRNN [70] outsources the computation of the activation function to the client, allowing the authors to use arbitrary activation functions.

In other schemes, low-degree polynomials are popular choices for activation functions since most schemes can easily evaluate polynomials. The simplest non-linear polynomial, the square function $f(x) = x^2$, is often used as a replacement for ReLU [27, 68, 70, 67, 61, 65, 69, 63, 26]. The square function is relatively fast and adds little overhead to the computation. However, it does not perform well in all problem domains. For example, using it in RNNs is infeasible due to its rapidly growing derivative [2]. The square function can not be used to replace ReLU in an already trained network since their outputs are too different.

More complex polynomials can reduce the approximation error. The Stone-Weierstrass theorem [76] states that any continuous function can be approximated with arbitrary precision over a closed interval using polynomials. However, accurate approximation requires high-degree polynomials, which introduce large computational overhead and noise. One needs to find a trade-off between approximation accuracy and polynomial degree for HE solutions.

Chabanne et al. [66] study polynomials with even degrees two, four, and six as approximations of the Relu function. They find the approximations by applying the "polyfit" method of the numpy[1] package to the output of ReLU on the standard normal distribution. The authors use this distribution since they put a batch normalization before every ReLU activation function. Combining polynomial activation functions with batch normalization

[1] `https://numpy.org`

can also help during training; polynomial activation functions' unbounded derivatives can lead to exploding gradients, which are counteracted by batch normalization. Hesamifard et al. [26] propose a method for finding approximations of common activation functions. The authors find a polynomial approximation for the function's derivative using Chebyshev polynomials [77]. By integrating that function, they can find an approximation for the activation function. The intuition behind their approach is that the derivative of the activation function plays a large role in training the NN, and approximating it more closely leads to better results. Their approximation approach is used by Podschwadt & Takabi [2], and RNN Blocks [3] to approximate the Tanh function, with degree three polynomials, for the use in RNNs.

Lee et al. [28] approximate the ReLU functions as $\text{ReLU}(x) = \frac{1}{2}x(1+\text{sign}(x))$. They use a composition of polynomials [78] to approximate the sign function. In their experiments, they use three polynomials of degrees 7, 15, and 27. This allows for a very accurate approximation of ReLU that can replace the activation in a trained NN without the need for further fine-tuning. However, the high degree of the polynomials requires bootstrapping to control the noise. The authors further propose a method for approximating the softmax function:

$$\text{softmax}(x) = \frac{e^{x_i}}{\sum_{j=1}^{n} e^{x_j}}; \text{for each } i = 0, 1, \ldots, n$$

Using the least-squares method, the authors use a degree 12 polynomial to approximate the exponential function. To compute the inverse function, the authors use Goldschmidt division [79], allowing them to approximate the division as a series of multiplications. Al

Badawi et al. [62], use minimax approximation [80] to find a polynomial approximation for the softmax function. Minimax approximation is to minimize the maximum error. Using this approach, the authors find $1/8x^2 + 1/2x + 1/4$ as an approximation of the softmax function. Since the softmax function is usually the last operation in an NN, most studies leave its evaluation to the client. However, this poses a risk since directly exposing the logits of the NN to the client makes it more vulnerable to adversarial and model inversion attacks.

CHET [64], nGraph-HE [69], and Jang et al. [73] use a different approach to finding polynomial approximations. Rather than numerically finding an approximation, in polynomials of the form $f(x) = a_n x^n + a_{n-1}x^{n-1} + \cdots + a_2 x^2 + a_1 x$, the coefficients a_i are learned during the NN's training. CHET and nGraph-HE learn polynomials of degree two while Jang et al. [73] train polynomials with degree seven, which requires bootstrapping during inference.

Nandakumar et al. [81] do not use polynomial approximations for the sigmoid activation function; they use a lookup table instead [82]. The client needs to precompute the table before running the network.

2.2.2 Polynomial Approximation: Theoretical Foundation

One of the major limitations of homomorphic encryption is the limited set of operations that can be performed. CKKS supports addition and multiplication. Division is supported only for plaintext divisors. Basically, this allows us to evaluate only polynomials. Tanh, a popular activation function in RNNs, can not be expressed as a polynomial. This means we can not evaluate it over encrypted data. A way to circumvent this is to find a polynomial approximation.

Hesamifard et al. [83] use an approach that is based on Chebyshev polynomials. Given the family of all continuous real-valued functions X on a non-empty compact space $\mathrm{C}(X)$ and let μ be a finite measure on X. The authors define $f, g \in \mathrm{C}(X)$ as $\langle f, g \rangle = \int_X fg d\mu$. To generate Chebyshev polynomials they use $d\mu = \frac{dx}{\sqrt{1-x^2}}$ as the measure on $[-1, 1]$. For better computational performance we want to stick to low-degree polynomials.

2.3 Compiler-based approaches to running neural network with HE

Running an HE-friendly neural network poses a new challenge. All operations in the network need to be implemented using HE primitives, in our case, addition, multiplication, and division by plaintext. While this is certainly feasible, it makes a fast turnaround from model training to execution on encrypted data difficult for individual experiments. To address this issue, prior work proposes privacy-preserving neural network compilers. Their goal is to transform neural networks, provided by popular frameworks like TensorFlow or PyTorch, into HE computation. This transformation often includes optimizations to make the execution more efficient. Podschwadt et al. [1] identify two such compilers: "Boehmer et al. [69] develop nGraph-HE, a backend for Intel's nGraph graph compiler. nGraph-HE takes existing HE-friendly NNs from popular ML frameworks like TensorFlow or PyTorch and translates them into HE operations. The goal is to require as little knowledge about HE as possible. nGraph-HE can perform several optimizations that speed up the NN inference. CHET is an optimizing compiler for tensor circuits over HE by Datathri et al. [64]. CHET's optimizations are encryption parameter selection, data layout selection, rotation key selection, and fixed-

point scaling factor selection. The compiler uses a cost model to find the most efficient operations and crypto parameters.“ Since then, a new framework, HElayers by Aharoni et al. [84], was released. Similar to CHET, HELayers also performs layout optimization. It analyzes the neural network and splits the computation into so-called tiles. Tiles represent ciphertexts and plaintexts. The layout optimizer aims to find the best tiling given the model and the input data. It does so by simulating the computation first and using heuristics to find the configuration with the lowest latency. For transforming models from multiple frameworks, the authors rely on Open Neural Network (ONNX) [2], an interchange format for neural networks. However, some of these compilers are closed source, like HELayers or CHET, which prevents us from making modifications during our experiments. For a more in-depth discussion of HE compilers, which is not limited to machine learning workloads, we refer the reader to Viand et al. [85].

2.4 Recurrent Neural Networks

Recurrent neural networks excel at processing sequential information. They do this by processing one sequence element after another and feeding prior internal states back into themselves (Fig. 2.1a). This recurrent behavior updates the internal with each new element. However, on encrypted data, this design causes problems. Conceptually, we can “unroll” any RNN layer along the temporal dimension. (Note: It is common in RNNs to consider one of the data dimensions as time. This is the dimension along which the sequence elements are ordered. E.g., if we consider the input, “This is a test input” the time dimension would go

[2] `https://onnx.ai/`

from “This” to “input”. It establishes a relationship from the first to the last element in the sequence.) The unrolling creates a network that does not have any recurrent connections but a new input from the input sequence at every layer (Fig. 2.1b). Computationally, the unrolled RNN is identical to the normal representation, it is just a different visualization. In a Simple RNN [86], the computation of the hidden state h_t at time t is computed as (for simplicity, we ignore the biases here):

$$h_t = f(x_t W + h_{t-1} V) \tag{2.14}$$

Where f is the activation function, x_t is the sequence element at t, W the input weights, and V the recurrent weights. We can expand h_{t-1} by substituting the equation at $t-1$:

$$h_t = f(x_t W + f(x_{t-1} W + h_{t-2} V) V) \tag{2.15}$$

We can repeat this expansion until we reach x_0:

$$h_t = f(x_t W + f(x_{t-1} W + f(\dots f(x_0 W + h_{-1} V) V) \dots V) \tag{2.16}$$

2.5 Convolutional Layers

Here, we consider two-dimensional convolutions commonly used in neural networks; however, other dimensionalities work fundamentally the same. Goodfellow et al. [87] define the operation as follows: given the inputs X,W, and the output Y, which are all tensors, we can

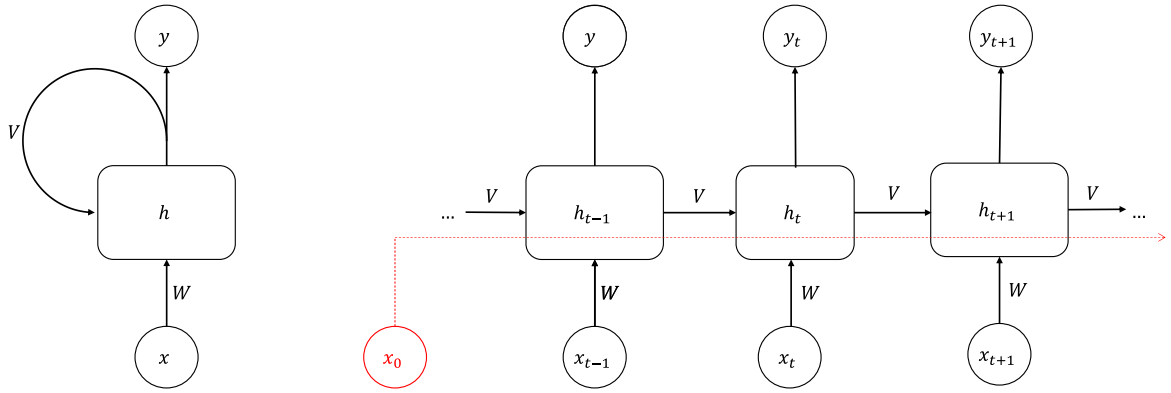


Figure 2.1: A simple overview of an RNN layer 2.1a and the "unrolled" RNN layer 2.1b along the time dimension t. The red dotted line shows the initial input value passing x_0 through every unrolled layer. (Image credit [3])

define the two-dimensional convolution as:

$$Y_{b,m,n,c_{\text{out}}} = \sum_{j} \sum_{k} \sum_{c_{\text{in}}} X_{b,m-j,n-k,c_{\text{in}}} W_{j,k,c_{\text{in}},c_{\text{out}}} \tag{2.17}$$

We use the subscript to indicate a single element in the tensor, where b is the batch index, c_{in} the input channel index, and c_{out}, the output channel index. Eq 2.17 needs to be computed for all values in Y.

2.5.1 Lock-free multi-threaded Convolution

The most straightforward way to compute a convolution with multiple threads is to have each $\mathbf{y}$ computed by a thread; Eq. 2.17 is computed by a separated thread for each unique $(b, m, n, c_{\text{out}})$. For $s = |Y|$, we can use at most s parallel threads without requiring some synchronization between the threads since all threads read from the shared resources X and W but do not modify them; every $Y_{b,m,n,c_{\text{out}}}$ is only modified by one thread, ruling out any

race conditions that could lead to lost updates. With fewer than s threads, threads can compute multiple $Y_{b,m,n,c_{\text{out}}}$. With more than s threads, we either need synchronization or can not use these threads. Generally, given n_t threads where each thread is assigned a unique integer $i \in [1, n_t]$, we can use Algorithm 1.

Algorithm 1 Lock-free multi-threaded Convolution

Inputs: input tensor X, weight tensor W, output tensor Y, number of threads n_t
Outpus: output tensor Y containing the result of the convolution

```
1: while t ≤ n_t and t ≤ |Y| do
2:     Start Thread t and execute:
3:       q := t
4:       while q ≤ |Y| do
5:           convert q to multi-dimensional index b, m, n, c_out
6:           Y_{b,m,n,c_o} := Σ_j Σ_k Σ_{c_i} X_{b,m−j,n−k,c_i} W_{j,k,c_i,c_o}
7:           q := q + n_t
8:       end while
9:     End Thread
10: end while
```

We assume that inputs are stored on a disk (the term disk refers to any persistent storage, i.e., a hard disk drive or solid-state drive) and must be loaded into memory. With the Algorithm 1, we have two options to keep it lock-free. 1.) Load X and Y before we start the computation, or 2.) Each thread loads the $\mathbf{x}$ and $\mathbf{y}$ as needed. 1.) has the upside in that we only need to load each value once and can reuse them at no additional cost. However, the downside is that we need to keep them in memory for the entirety of the computation. 2.) On the other hand, it needs to keep much fewer objects in memory. Each thread has only three objects in memory: one $\mathbf{x}$, one weight $\mathbf{w}$, and the output $\mathbf{y}$. However, each thread must perform two loads for each iteration of the nested sums in line 6. Furthermore, multiple threads may load the same $\mathbf{x}$ and $\mathbf{w}$, causing redundant loads. A further issue with

this algorithm arises when $|Y|$ is not divisible by n_t. In this case, $|Y| - |Y| \mod n_t$ threads finish one iteration, line 4, early and are idle for the rest of the computation, leading to unused computational resources. However, this impact is small if $|Y|$ is large compared to n_t. Performing the second option on plain data will lead to slower results since arithmetic operations are much faster than data loading. Additionally, single $\mathbf{x}$ and $\mathbf{w}$ are so small that we can not save significant memory by loading them on demand.

Running Algorithm 1 on encrypted data is straightforward using batch-packing described earlier. To do this, we replace X with X', W with W', and Y with Y'. This replacement sets the batch dimension to one, allowing us to remove it from consideration. For the algorithm, it does not matter if W' is encoded HE plaintexts or ciphertexts if the model is encrypted. In the case of the plaintext model, we assume that unencoded weights W are loaded into memory before the computation starts and are encoded when needed; therefore, we don't need to load them strictly speaking. However, we call this operation loading for simplicity in this context.

CHAPTER 3
PRIVATE INFERENCE

This chapter will identify the challenges of running a trained model on encrypted data. We assume the model is trained on plain data and has already been transformed into an HE-friendly model. We identify numerous challenges, such as resource overhead, the depth of a model, issues stemming from unsupported operations, and usability issues. We then present and evaluate Aluminium Shark, a tool to run neural networks over encrypted data with little code overhead.

3.1 Challenges of running a trained model on encrypted data

Based on our initial implementation, CryptoDL[88], we identify design considerations for a tool to run neural networks over encrypted data. CryptoDL taught us valuable lessons regarding design and usability. It supports fully connected, convolutional, and vanilla RNN layers. Other than the limited number of layers, CryptoDL has other limitations. Many of the limitations are around usability. Running a model requires the user to implement it in C++. A rudimentary model importer can convert and import Keras models saved in the h5 file format. CryptoDL extracts the model configuration from this format and exports the weights into a file format it can read. It is very strongly coupled to the layer format, as Kears specifies it. Despite this strong coupling, it only supports a small subset of the specification. For example, to use CNNs and image data, the model and the data need to be formatted to have the channel information as the first dimension after the batch dimension. Other formats are not supported. Any new additions or changes to the format need to be

implemented in C++. This strong coupling makes us very inflexible. Keras is not the only strongly coupled dependencies. CryptoDL uses HELib[89] as its computational backend and lacks an abstraction layer to support other libraries.

One of the major problems of running neural networks on encrypted data is the time overhead. Plaintext models can typically run in a few seconds on today's systems; models over encrypted data can take hours. Even models are considered simple, for plain data can take a long time. For example, LeNet5 [90] for MNIST digit classification can take 1,272 seconds. These values are with batch-packing, so the amortized per instance cost is much lower at 0.08s per instance. While this might be sufficient for big batch jobs, classifying a few instances carries a large overhead. Ciphertext packing strategies like Brutzkus [63] can alleviate that problem only to some extent. In a recently published paper, Lee et al. [28], run the ResNet-20 model on a single encrypted instance in a little under three hours. Because of the depth of their model, they need to employ bootstrapping, which further increases the time overhead.

But time is not the only resource that has a significant overhead. Running neural networks on encrypted data comes with large memory requirements. This becomes apparent when we compare the sizes of plain data and ciphertexts. Let us assume that on plain data, a single number requires 64 bits of memory. A vector of 16,384 numbers then requires 128 KB of memory. The size of a ciphertext depends on the crypto parameters; however, if we assume parameters that can encode 16,384 values into a single ciphertext, then a single encryption of this vector can take up to 10 MB of memory. The size of a ciphertext is not

only dependent on the security parameters but also on the number of levels. As the number of levels increases, so does the size of the ciphertext. To go back to our LeNet5 example from earlier, running this model takes over 300 GB of memory without optimizations. The input data on MNIST that is $28 = 784$ ciphertexts, is 7840 MB alone. Or take Lee et al. [28], who require 172 GB to run inference on a single instance. Requiring machines that have hundreds of GB of memory is not uncommon for private inference using neural networks [1].

We can see that the depth of a network influences its resource requirements on encrypted differently than on plain data. On plain data, adding an additional layer only adds the time and memory requirements for that layer. It does not change the computation time and memory requirements of the other layers. This is not the same on encrypted data; here, adding a layer has an impact on all the other layers. The ciphertext size needs to increase to accommodate the multiplications added by the new layer. If the new level is not secure enough with the current polynomial degree, we need to increase that as well, and operations with a larger polynomial degree are also slower. However, we cannot add levels indefinitely; eventually, the computation becomes so slow due to the polynomial degree that bootstrapping is faster. To keep execution time and memory requirements low, we want to keep the model as shallow as possible.

There are more challenges and open issues to running a neural network over encrypted data. The following analysis, until Section 3.1.1, was previously published in Podschwadt et al. [1] and has been slightly edited.

A handful of libraries are available that support one or more encryption schemes. How-

ever, even if libraries support the same scheme, they are not interoperable. For instance, ciphertexts or keys do not have a standardized format. The same is true for the encryption parameters. The interpretation of what a parameter means can differ between libraries. In the past few years, there has been an effort to standardize HE by the community [91]. This standardization effort [53] includes an overview of the security and secure parameter recommendations, potential applications, and design considerations. It is a good start, but at the moment, developing solutions locks one into using a particular library. HELayers [84] addresses these issues by incorporating parameter selection.

Choosing a library is usually difficult; developing a well-performing secure PPML solution without in-depth knowledge about HE is hard, if not impossible. PPML developers need to have expertise in both ML and security. PPML, using HE, has not been widely adopted by the ML community, likely due to the high barrier of entry that is HE and the lack of user-friendly tools. On the other hand, security researchers often lack ML knowledge. The ML community has developed user-friendly tools allowing fast plaintext ML development [92, 93, 94]. These tools hide much of the complexity from the user; such complexity is necessary to understand when implementing solutions based on HE.

There should be an effort to develop HE-based PPML libraries by a shared community of ML specialists and crypto experts. Some work has been done on easing the entry from both sides by using compilers [69, 64, 84]. Compilers not only make it easier to get started, but they can also significantly improve performance by applying optimizations. Compilation can help speed up the execution of encrypted data. However, the models are designed with

plaintext execution assumptions.

Fast and efficient techniques on plaintext are not always the best solution for encrypted data [27, 26]. Researchers should take a more holistic approach that considers the entire ML pipeline. Much of the work focuses on running existing model architectures on encrypted data [27, 64, 28]. This work is often done by making small changes to the model architecture to make it HE-compliant and training the model from scratch [59, 58]. Ideally, the compiler suite would be able to perform all these transformations automatically while optimizing the process end-to-end. Most ML techniques in the field are minor adaptions of strategies that work on the plaintext [62, 27, 58]. However, developing solutions specifically for encrypted execution might require new types of networks, activation functions, or optimizers.

Activation functions are a bottleneck in NNs using HE. These non-linear functions are necessary for the network to perform well. However, on encrypted data they significantly impact the noise budget [28, 59]. Multiple alternatives have been proposed, but most of them make training the model harder, often due to their unbounded and non-monotonic derivatives [66, 3]. Other solutions that do not impact training, such as replacement after training or lookup tables, suffer from a loss in model performance [28]. Polynomials are an adequate replacement for traditional activation functions [28, 27, 26]. The question of whether there are activation functions that perform well, both on encrypted and plain data, is still open and worth investigating.

The need for HE-specific algorithms and adaptions is especially apparent in training. Work investigating training on encrypted data is minimal [81, 62, 65]. One of the big

reasons is the noise build-up and the resulting need for bootstrapping or interactive noise removal. Solutions that use neither are limited to using training hyperparameters, such as batch size and the number of epochs, that are sub-optimal [62]. Approaches that do use bootstrapping are very slow and computationally expensive [95, 73, 81]. On the one hand, faster bootstrapping would remove some of the computational overhead of training on encrypted data, and we could use standard training parameters. On the other hand, having a training algorithm that works with the parameters required by HE would eliminate the need for bootstrapping or interactive noise removal and make training much more feasible.

Performance is not only an issue during training but also during testing. Researchers can evaluate models on the CIFAR-10 dataset within a reasonable time [67, 64]. However, larger models and larger inputs still take hours to compute [58, 28]. Hence, scaling up to larger, real-world datasets is costly. Improvements in latency and memory consumption have been due to ciphertext packing [63, 61, 18]. Without it, scaling up to real-world data is impractical. Packing can reduce the memory requirement by an order of magnitude. Models on the CIFAR-10 dataset that do not use packing require hundreds of GB of memory [64, 59, 58], making scaling up to datasets like ImageNet near impossible.

Another potential source of increased performance is hardware accelerators like GPUs or FPGAs (Field Programmable Gate Arrays). Al Badawi et al. [67] ran models using HE on GPUs, but memory constraints were severe. High-end GPUs do not have more than 50GB of memory. The memory limitation of GPUs calls for new encoding and packing schemes for efficient execution. Research on GPU-aided HE is not limited to ML applications. For

example, Geraldo et al. [96] propose an implementation of BFV accelerated by GPUs. Their results show an up to 5 times speedup in homomorphic operations. Increasing the efficiency of NNs can also help reduce the memory and runtime requirements of PPML. Much work has been done on running plaintext ML applications on resource-constrained devices like phones [97, 98]. Techniques used in that area, such as pruning and quantization, can be implemented in PPML [99]. Chou et al. [59] investigated reducing the number of computations in the network through pruning. Helbitz & Avidan [100] tackled the computational overhead of PPML by reducing the ReLU count in the network. They show that activations can be grouped, and one activation output can be used for all in the group. The accuracy impact on this grouping is low, while it can provide about 30% speedup. They test their system using two and three-party computation. If a similar speedup can be observed on HE systems or if it is even applicable to other activations, then ReLU still needs to be investigated.

Most PPML solutions focus on client data privacy; it is guaranteed if the underlying schemes remain secure. However, if these systems are to be adopted in practice, model providers likely need more assurance that their models are secure; server-side model information can be leaked using HE.

Information leakage can occur in various directions. For example, either information about the client's data can leak to the server, or the server's model can leak to the client. All approaches in this survey focus on protecting the privacy of the client's data; however, the model's privacy can also be a concern. While the client does not have direct access to the model's parameters, Tramer et al. [21] show that an attacker can reverse engineer a

model with access only to the model's predictions. Interestingly, there is more information leakage about the model to the client on encrypted data. One reason is that it is common practice to leave the decoding of the results to the client (usually softmax or beam search [68]). These raw values make it easier for the client to learn the network's weights. Another reason is that the client can infer some of the network architecture through the crypto parameters. Especially when using leveled HE, the client gains information about the depth of the computation because the multiplicative depth is directly correlated to the number of layers in NNs. The server can obfuscate this information by choosing larger parameters than the network requires, increasing running time and memory consumption. Alternatively, bootstrapping can deny this information to the client. However, this also increases resource requirements. Additionally, the encoding methods used with ciphertext packing can reveal the model's information.

The information leakage becomes even more severe when the client needs to participate in the computation. CryptoRNN[70] and Podschwadt & Takabi [2] face the problem of revealing internal states (e.g., layer's outputs) to the client during noise removal. Knowing the internal state could allow the client to learn the model's parameters since the client learns the exact number of units used in the hidden layers. Furthermore, with access to the intermediate states, the client gains more information than assumed by most model stealing attacks [101]. It can potentially use this additional information to improve the attack. The server can prevent the client from learning the actual internal state by adding randomness to the data before sending it to the client for noise removal. The server can remove the

randomness once the client returns the encrypted ciphertext to the server. This obfuscation becomes more complicated when the client computes nonlinear functions, as in CryptoRNN [70].

Multiple approaches need to share additional information with the client for preprocessing. Podschwadt & Takabi [2], RNN Blocks [3], and PrivFT [62] need to share all the words known to the model. Unless the model operates in a very specific domain with a very specific vocabulary, the client does not gain valuable information. Podschwadt & Takabi [2] and RNN Blocks [3] additionally need to share the embedding matrix. However, this does not leak any information to the client since there are plenty of pretrained embeddings [102] that are publicly available. So, the client does not learn anything new about the model.

3.1.1 RNNs on encrypted data

On encrypted data, RNNs typically exhibit much higher multiplicative depth than feed-forward networks. One of their greatest strengths, internal memory, is responsible for the increase in multiplicative depth. Fig. 2.1b helps highlighting the problems of RNNs on encrypted data. It shows the first sequence element x_0 passing through every layer in the network. The problem on encrypted data is the multiplicative depth. In an RNN, the multiplicative depth is tied to the number of elements in the input sequence. Simply put, each element adds a layer to the unrolled network. The formula shows this more clearly (Eq. 2.16). Here we can see that the multiplicative depth of a network with t sequence elements is the multiplicative depth of f times t. If we consider polynomial activations, a popular class of activation functions for encrypted data, we can establish an estimate of the multiplicative

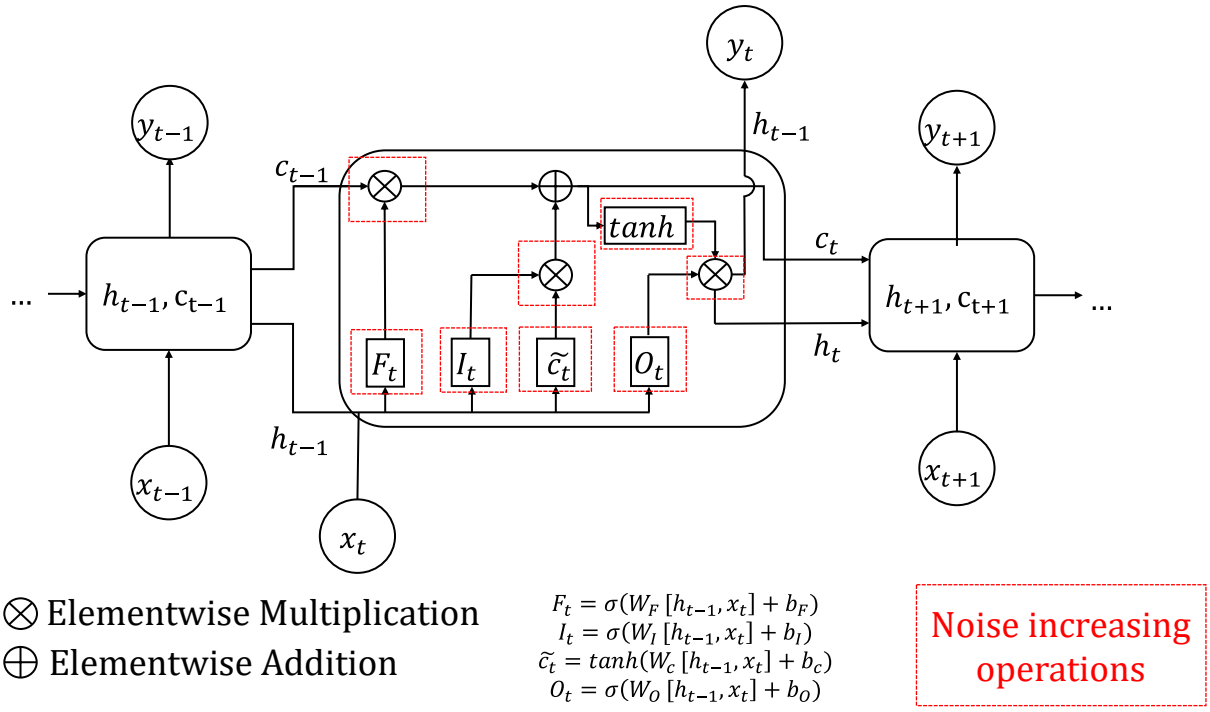


Figure 3.1: Internals of a long-short term memory (LSTM) cell with noise-increasing operations highlighted. (Image credit [1])

depth. For a polynomial with coefficients a_i and degree n:

$$f(x) = a_n x^n + a_{n-1}x^{n-1} + a_{n-2}x^{n-2} + \cdots + a_2 x^2 + a_1 x + a_0; a \in \mathbb{R}, n \in \mathbb{Z}^+ \tag{3.1}$$

we can estimate the multiplicative depth as n. For a monomial, the multiplicative depth is $n-1$. For higher-degree polynomials and monomials, this gives us an upper bound since we can compute the exponentiation in each term with a multiplicative depth of less than $n-1$ by successive squaring of the intermediate results. This gives us a best multiplicative depth of $log_2(n)$ and an estimated worst-case multiplicative depth of $\lfloor log_2(n+1) \rfloor$. Therefore we

can estimate the multiplicative depth of the entire RNN to between $t*log_2(n)$ and $t*n$; with n the degree of the activation function and t the length of the input sequence. For example, assuming the simplest non-linear polynomial, x^2, and an input length of 20, we can estimate the multiplicative depth to be 20. These noise estimates are only for simple recurrent neural networks; other recurrent layers, such as long short-term memory [71] (LSTM), or gated recurrent units [72] (GRU), impose even higher multiplicative depth due to their complex structure. We can see the difference in depth if we compare the computation in a vanilla RNN cell to that in an LSTM. The import part is h_{t-1} since that contains the ciphertext that has already undergone computation and, therefore, is at a lower level, as shown in Fig. 2.1b. From Equation 2.15 we can see that the vanilla RNN cell requires one multiplication of h_{t-1} with V and an application of the non-linear function f. In an LSTM cell we have (we omit the input from the current step x_t and the biases for better readability):

$$h_t = \tanh(\sigma(W_F h_{t-1}) \times c_{t-1} + A) \times \sigma(W_0 h_{t-1}) \tag{3.2}$$

With $\times$ being element-wise multiplication and $A = \sigma(W_I h_{t-1}) \times \tanh(W_I h_{t-1})$. However, A has no impact on the multiplicative depth since it is "parallel" to $\sigma(W_F h_{t-1}) \times c_{t-1}$. Here, we have one multiplication with the weights W_F, two non-linear actuation functions, one multiplication with c_{t-1}, and one multiplication with $\sigma(W_O h_{t-1})$. Assuming that the model is in plain, the total is one ciphertext x plaintext multiplication, two ciphertext x ciphertext multiplications, and two activation functions. The vanilla RNN has one ciphertext x plaintext and one activation function. This means the LSTM is two levels plus the depth of one

activation function deeper than a vanilla RNN.

Most work on neural networks with HE (e.g., [27, 26, 67]), work with networks that have at most half the multiplicative depth. Furthermore, it is unclear if the square function is suitable for use in RNNs. Typically, RNNs use the Tanh function. Tanh produces outputs in $[-1, 1]$. If the negative values are needed in the recurrent state or if we can make do with positive values, it only needs to be investigated.

3.1.1.1 Previous work

There is little work on PPML using recurrent layers. Bakshi & Last [70] propose CryptoRNN for RNN privacy-preserving classification. However, this approach requires client-server interactions for noise removal, although it only uses small datasets. CryptoRNN uses simple RNNs with client communication too. They refresh noise at three predefined points in the networks: 1) after every multiplication, 2) before every non-linear activation function, and 3) after processing each sequence element. It should be noted that at point (2), the client also computes the activation functions.

Lou et al. [58] propose a solution called SHE for Shift-accumulation-based leveled-HE-enabled deep NN, based on TFHE, which allows the implementation of the ReLU function and max-pooling layers. This solution increases the performance since there is no need for function approximation. However, TFHE performs matrix operations slower than the other HE schemes. SHE [58] modified the LSTM structure to use the ReLU function, which is comparatively easier to compute using TFHE than Tanh. Typically, RNNs use the Tanh function, which requires at least a polynomial of degree three for an adequate approximation.

The authors also replace all the multiplications with shift operations, which they can perform at no cost to the noise budget.

Jang et al. [73] propose a variation of the CKKS scheme, MatHEAAN (Matrix HEEAN), that specializes in matrix operations. Based on this scheme, the authors Gated Recurrent Units (GRU) [72], to handle sequential data. Since recurrent NNs (RNNs) typically have high multiplicative depth, the authors use their custom bootstrapping algorithm to refresh the noise during computation.

3.1.2 Addressing Challenges

Throughout the remaining parts of this work, we will take a more in-depth look at several of the challenges above. We will first describe and evaluate Aluminium Shark in the remainder of this chapter. Afterward, we present caching techniques to reduce the memory requirements of batch-packed convolutions in Chapter 4. We then go on to investigate RNNs over encrypted data and present an interactive approach in Chapter 5 and a new architecture in Chapter 6, which expand upon in Chapter 7.

3.2 Related Work

In recent years a number of HE compilers have been proposed. Most of them are general-purpose compilers that do not specifically target machine learning. ALCHEMY is a general-purpose compiler by Crocket et al. [103]. It uses its own domain-specific input language, which is implemented in Haskell. ALCHEMY uses the BGV crypto scheme. Cingulata, by Carpov et al. [104] use BFV or TFHE internally. It compiles C++ code into boolean circuits.

It also offers some optimizations that reduce the depth of the circuit. Encrypt-Everything-Everywhere by Chielle et al. [105] also uses C++ as its input language. It supports the BGV, BFV, and THFE schemes and supports multiple crypto libraries. Internally it uses a propriety hardware design software to reduce the depth of the compiled circuits. Marble by Viand et al. [106] is another compiler that uses C++. However, it does not compile C++ code directly but allows users to write their own C++ code with easy-to-use HE primitives. It uses the BFV and BGV crypto schemes. By Archer et al. [107], Ramparts is built on the PALISADE crypto library and uses the BFV scheme. It uses a symbolic simulator to simplify the circuit. We refer the reader to Viand et al. [85] for a more in-depth comparison of the compilers.

In theory, some of these compilers could directly translate the code used by machine learning frameworks. However, these frameworks include thousands of lines of code that are not strictly necessary for pure neural network inference. Additionally, by being aware of the compiler's domain, we can rule out irrelevant scenarios and make optimizations based on the narrower range of target uses.

Multiple HE compilers geared towards neural network inference have been proposed to harness the strength of domain-specific compilers. CHET [64] takes a so-called tensor circuit that describes the computation. However, these tensor circuits, similar to HLO graphs, need to be written by hand. There does not seem to be a way to generate them from TensorFlow or PyTorch automatically. A strength of CHET is its support of different ciphertext layouts and the ability to select the best layout for a given tensor circuit. According to the authors, this

drastically reduces the inference time and even beats hand-crafted solutions done by experts. nGraph-HE [69], similarly to Aluminium Shark, directly extracts the computational graph from TensorFlow. In nGraph-HE, the graph is processed by the external nGraph graph compiler. This compiler is an optimizing compiler for multiple different hardware backends. nGraph-HE implements homomorphic encryption as a virtual hardware backend for which the graph can be compiled. In their second release [108], the authors add secure multiparty computation to evaluate activation functions incompatible with the underlying HE scheme interactively. Their overall goal is similar to ours: making encrypted machine learning more accessible by providing integration into tools that machine learning practitioners are already familiar with. However, we offer a more extendable framework that allows us to integrate new cryptosystems and libraries easily. SEALion [109] builds on top of TensorFlow. It uses TensorFlow for its plaintext operations. The trained model can be executed over encrypted data. However, due to their design, all layers and activation functions used in the model need to be specifically implemented for encrypted execution. Our approach has the advantage that we only need to implement a small set of basic operations used by TensorFlow to build its models. This means that as long as future models are built on these basic operations, Aluminium Shark can support them.

Other approaches to integrating privacy preservation into machine learning frameworks are either not based on HE or do not use a compiler. CrypTFlow [32] and CyrpTen [33] both integrate PPML into popular machine learning frameworks. CryptTFlow into TensorFlow and CrypTen in PyTorch. However, they do not use homomorphic encryption for privacy

preservation but rather guarantee the privacy of the data with secure multiparty computation. TenSEAL [110] is a tensor library for computation on homomorphically encrypted tensors. It provides basic tensor operations and can convert tensors from popular frameworks into encrypted tensors. However, a downside is that network structures, like layers and activation functions, need to be built using the basic tensor operations. There is no translation from high-level objects, as, for example, defined Keras [1], into the supported low-level tensor operations. HElayers by Aharoni et al. [84], was released. Similar to CHET, HELayers also performs layout optimization. It analyzes the neural network and splits the computation into so-called tiles. Tiles represent ciphertexts and plaintexts. The layout optimizer aims to find the best tiling given the model and the input data. It does so by simulating the computation first and using heuristics to find the configuration with the lowest latency. For transforming models from multiple frameworks, the authors rely on Open Neural Network (ONNX) [2], an interchange format for neural networks. However, some of these compilers are closed source, like HELayers or CHET, which prevents us from making modifications during our experiments.

3.3 System Description

We now present Aluminium Shark, a privacy-preserving backend for TensorFlow's XLA (Accelerated Linear Algebra) compiler. This system provides a foundation for running deep learning models that make predictions on data encrypted using homomorphic encryption

[1] `https://www.tensorflow.org/api_docs/python/tf/keras`
[2] `https://onnx.ai/`

(HE), integrated into one of the most widely-used deep learning libraries. Users of our system can run existing neural network models over data without ever exposing the data in its non-encrypted form.

Running neural networks on plain data has become increasingly simple, thanks to frameworks like TensorFlow [93] and PyTorch [94]. These frameworks allow the user to run neural networks with little implementation work. The picture is quite different for PPML. While there are some frameworks available, like CrypTFlow [32] and CyrpTen [33], which are based on secure multiparty-computation and therefore require server-client interaction during evaluation. TenSEAL [110] and nGraph-HE [69] are based on homomorphic encryption. However, these libraries are locked into a crypto library and scheme. Further, they require much manual work to run neural networks on encrypted data. The latest version of nGraph-HE [108] does not only support HE but also uses interactive phases based on secure multiparty-computation to compute the non-linear activation functions.

Aluminium Shark is an an integrated backend that compiles or interprets the XLA [111] intermediate representation (IR) for encrypted execution. XLA, which stands for accelerated linear algebra, is an optimizing domain-specific compiler for machine learning models. XLA is integrated into TensorFlow, one of the most popular deep-learning frameworks currently in use. In September 2023 alone, TensorFlow was installed over 17 million times using the PyPI package manager, according to PyPistats.org[3]. It is precisely this wide adoption and the already integrated compiler infrastructure that led us to choose TensorFlow as the basis of Aluminium Shark. Furthermore, PyTorch, another very popular deep learning framework,

[3] `https://pypistats.org/packages/tensorflow`

can also be used as an XLA frontend, making Aluminium Shark useful for both TensorFlow and PyTorch users.

Our goal with this platform is to make HE-based PPML more accessible to the machine learning community, making the integration as simple as possible for the user. Aluminium Shark supports the most common operations required for neural network inference. It performs various optimizations to speed up computation and supports multiple data layouts. Aluminium Shark is not locked into a single HE library or scheme. The design allows it to add a new HE library easily. By default, Aluminium Shark uses Microsoft SEAL [112]; however, OpenFHE [56] is also supported.

3.3.1 Compiler

A compiler takes code written in one language and translates it into another language. Often this is the translation from a high-level programming language into machine code, creating an executable. For HE compilers, this definition is often relaxed to include interpreters. An interpreter does not produce an executable but executes the input code directly. Interpreters and compilers can be combined. For example, the input code can first be compiled into a different representation which an interpreter then executes. HE compilers often act as interpreters. Their input can be coded in a common programming language like C++, their own language, or some domain-specific language. The compiler takes these inputs and produces a circuit that can be run over encrypted data. Often the compiler also estimates the crypto parameters and handles the key management so that the user does not need to interact with the low-level crypto objects. Some compilers can also select the best data

packing scheme that reduces the execution time and memory requirements.

Deep learning frameworks also use compilation and interpretation to execute neural networks. In particular, TensorFlow used to only work in compiled mode. Here the entire computation graph needed to be defined before it could be executed. Recently, the so-called eager mode has gained popularity. In eager mode, the graph does not need to be defined before execution. Each node is executed as soon as it is called. Eager mode is popular because it is intuitive and easy to debug. However, the compiled mode offers more opportunities for optimizations. Knowing the entire computational graph before execution allows the compiler to determine which intermediate results are needed for future computation. With this knowledge, the compiler can reorder or parallelize computation, reduce memory usage by freeing unused intermediate results, and fuse operations together. Fusing operations is especially helpful on hardware accelerators. When multiple operations are fused, the intermediate results are not written back into main memory but can remain in the accelerator's memory. It is then used directly as the input for the following operation. Aluminium Shark uses the output of the TensorFlow internal XLA optimizer and interprets the HLOs on encrypted data.

3.3.2 XLA

XLA compiles the TensorFlow graph into an intermediate representation, called the high-level operations intermediate representation (HLO). XLA analyzes the computational HLO graph and performs some analyses and optimizations, resulting in an optimized HLO graph. These operations are target-independent. The optimized HLO is then passed to a target-

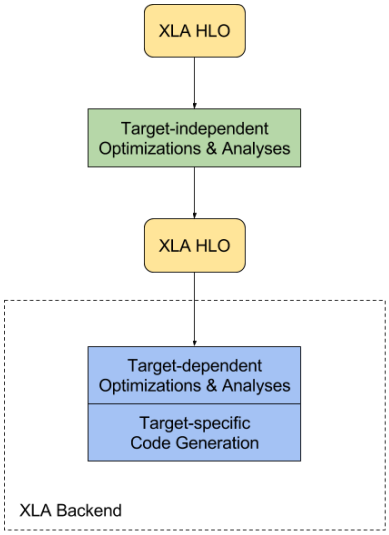


Figure 3.2: The XLA architecture. Image credit `https://www.tensorflow.org/xla/architecture` licensed under Creative Commons Attribution 4.0 License `https://creativecommons.org/licenses/by/4.0/`

specific XLA backend. The backend can perform more optimizations that are specific to the target platform. After that, the backend creates target-specific code to execute the graph on the given platform. Tensorflow provides backends for X64, ARM64, and NVIDIA GPUs. These backends are based on LLVM[4] [113], a compiler toolchain. An overview of the architecture is shown in Figure

HLO defines over 100 operations[5]. However, only a small subset of these is required to perform inference on the most common model architectures. We impose some restrictions on the supported architectures and HLO operations based on the operations supported by the HE schemes.

The goal of Aluminium Shark is to make running machine learning models on encrypted data as easy as possible. The user should need as little knowledge about HE as possible.

[4] `https://llvm.org/`
[5] `https://www.tensorflow.org/xla/operation_semantics`

Aluminium Shark is an XLA backend that allows users to run TensorFlow on encrypted data easily. Execution on encrypted data only requires a few lines of extra code and is entirely done in Python, with no C++ necessary. We use SIMD to speed up the computation and have designed a serializable representation for the encoding scheme that can easily be shared between data and model owner.

We build our backend on top of XLA. The backend takes the target-independent optimized HLO graph from TensorFlow as input and executes it on encrypted data. To make the execution as simple as possible for the end user, Aluminium Shark is directly integrated into TensorFlow and can be used directly from Python. One of the design goals of Aluminium Shark is that the user should be able to use it with as little effort as possible.

As an example, Figure 3.3 shows a simple piece of Aluminium Shark code that loads and runs a model on encrypted MNIST data. The user only has to take a few extra steps. First, the user loads the HE backend and creates a context. By default, SEAL is loaded as the HE backend. The context instantiates the scheme. The parameters that are passed rely on the type of scheme used. The scheme in the example code is CKKS, and the passed values are a coefficient modulus degree of 8192, a list of bit-lengths that will go into creating the coefficient modulus, and an initial scaling factor of 40 bits. Second, just as when running on plaintext, the user needs to create a batch of samples to feed to the model. To run on encrypted data, the batch needs to be encrypted first. Here the user selects as many instances as the HE scheme provides slots. By using the context object's encrypt function and specifying the batch layout, the context will lay out the data correctly and encrypt

it. Third, to execute the model on encrypted data, the model needs to be wrapped in an EncryptedExecution object. This object requires a function that creates (or loads) the model and the context. Fourth, passing the encrypted data to the EncryptedExecution instance triggers just-in-time compilation and execution of the model on the data. Finally, the result of the execution is encrypted as well and can be decrypted using the context, provided the context has access to the private key.

3.3.3 Connecting HE and Tensorflow

TensorFlow allows developers to extend it with custom operations[6]. However, there is no official API that provides the registration of custom datatypes, and while the development of custom XLA backends is supported, it appears that their dynamic loading at runtime is not supported at the moment. For these reasons, we integrate Aluminium Shark directly into TensorFlow. This integration does come with a few downsides as well. We need to rely on non-public APIs, which makes maintenance harder. Furthermore, the integration into TensorFlow requires a custom-built TensorFlow rather than just a plugin.

Aluminium Shark consists of four main components—the Python layer, HE Backend, XLA compiler and interpreter, and C API. The Python layer provides easy-to-use functions to allow the user to encrypt data and run models on encrypted data. The compiler analyzes and transforms the HLO graph so that it can be run on encrypted data. The interpreter executes the operations in this transformed graph on encrypted data. The HE backend provides an API that abstracts different HE libraries and provides a unified way of interacting

[6] `https://www.tensorflow.org/guide/create_op`

```python
import tensorflow as tf
import aluminum_shark as shark
from tensorflow.keras.datasets import mnist

# load model
def create_model():
  model = tf.keras.models.load_model('mnist.h5')
  return model

# load the MNIST data
(x_train, y_train), (x_test, y_test) = mnist.load_data()

# First: load the HE backend and create the context
backend = shark.HEBackend()
context = backend.createContextCKKS(8192, [40, 20, 20, 40], 20)
context.create_keys()

# Second: create a batch and encrypt it
n_slots = context.n_slots
x_in = x_test[:n_slots]
ctxt = context.encrypt(x_in, layout='batch')

# Third: wrap the model for encrypted execution
enc_model = shark.EncryptedExecution(create_model, context=context)
# Fourth: compile graph and run computation
result_ctxt = enc_model(ctxt)

# Finally: decrypt the result
decrypted = context.decrypt_double(result_ctxt)
```

Figure 3.3: Python example for simple encrypted execution

with them. To use a new HE library, the backend API needs to be implemented for that library.

A big problem that needs solving is that we need to transport encrypted data from Python into the TensorFlow runtime. The TensorFlow runtime is written in C++ and can be controlled from Python. On plain data, data is transported between Python and

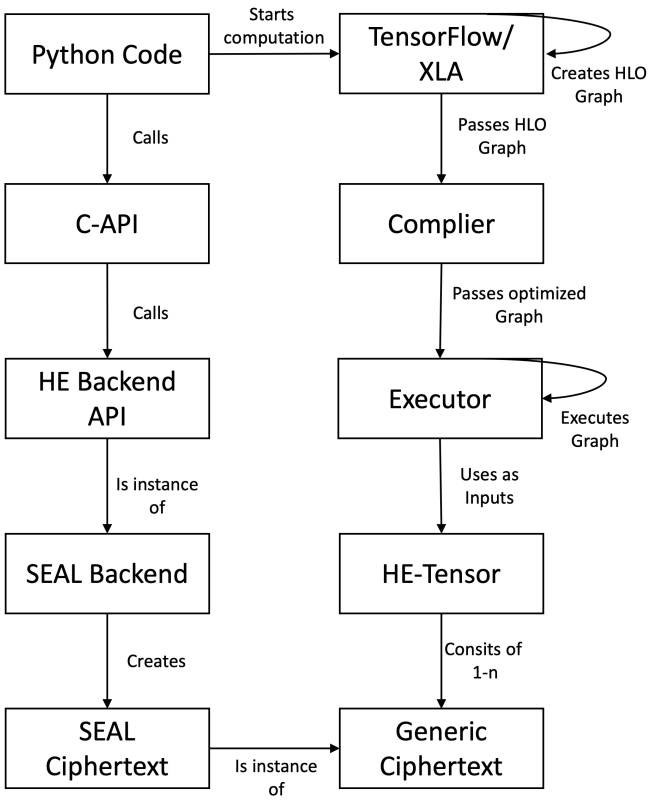


Figure 3.4: Interaction of the components of TensorFlow and Aluminium Shark

TensorFlow using Google's protocol buffers. Since most ML practitioners use TensorFlow from Python, we want to make Aluminium Shark usable from Python too. However, most crypto libraries are written in C++. It would be a significant overhead to transfer ciphertext from C++ to Python and from Python into the TensorFlow runtime.

We solve this problem by never explicitly passing the ciphertext to the Python layer. We only ever pass a handle to a ciphertext from the C-API to Python. These handles can be used to decrypt ciphertexts or to use them in computation.

We define a (virtual) platform and device to integrate with TensorFlow. A platform and device are TensorFlow internal abstractions for execution environments such as CPU,

GPU, and TPU. Our backend does not provide access to hardware like the other platforms but rather to a different type of data. We register this virtual device with the TensorFlow runtime, which provides us with an HLO graph when a function is placed on our virtual device. Here we pass the graph to our compiler. The compiler can analyze and optimize the HLO graph, after which it instantiates an executor. The executor visits all the nodes in the HLO graph and performs the represented operations. Operations that have only plaintext inputs are executed on plaintext. For operations on ciphertexts and plaintext, the plaintext values are laid out and encoded right before the evaluation of the node.

Operations on ciphertexts are performed on HE-Tensors. HE-Tensors are objects that, similar to plaintext tensors, have a shape that defines how the values are organized along the dimensions. They consist of multiple ciphertexts or encoded plaintext objects holding actual values. They additionally have a so-called layout. The layout describes how the slots of one or more ciphertexts or plaintexts (all of this is true for both plain and ciphertext but for ease of use, we only talk about the ciphertexts from here on out) map to a message tensor. The difference between messages and ciphertexts is that ciphertexts are encoded into polynomials, and a message is not. An HE-Tensor consists of multiple ciphtertexts, which in turn consists of multiple values. We can create an HE-Tensor by encrypting a message tensor. Before the values in the message tensor can be encrypted the need to be encoded. Since multiple values can go into a single ciphertext we need a mapping between values and ciphertexts. We also need the reverse mapping during decryption to map the multiple values in a ciphertext to a message tensor. To understand how the mapping works it is important to

understand how shapes are handled internally. Consider that we have a tensor T with shape S and n dimensions. S_i denotes the i-th element of the shape S. The number of elements or the size of T can be calculated as $\prod_{i=0}^{n} S_i$. Every value in T can be addressed by using a multidimensional index I where I contains as many elements as S and $\forall i(S_i > I_i); 0 < i \leq n$. The values of T can be saved in a continuous memory block, also called a flat block, since it only has one dimension. Given a multidimensional index I, the value l indicates which element in the block of memory is referenced by I. I can be converted into l using Eq. 3.3.

$$l = \sum_{i=0}^{n} I_i \times \prod_{j=n-i}^{n} S_j \tag{3.3}$$

This allows access to tensors like NumPy arrays, e.g., x[1,2,3], where the list (1,2,3) is translated into a single index into the storage memory block. Mapping a message onto one or multiple ciphertexts can change the order of elements as they appear in the ciphertext. For example, with batch encoding, the message x with the shape (10,10) would be encoded into ten ciphertexts where the i-th ciphertext contains the values $x_{i,j}$ with $0 \leq j < 10$. The layout tells us how to create the mapping from a message to a ciphertext and vice versa. The mapping from message to ciphertext tells us how we need to arrange the values prior to encoding. It maps from $\mathcal{N} \to \mathcal{N}^2$. The value at index i in the flat storage block of the message goes into the kth slot of the jth ciphertext according to the mapping. This mapping is serializable as a list of triples, wherein for each triple, the first element is i, followed by k and j. This way, the model owner can easily transmit encoding instructions to a client in a machine and human-readable format, which is also easy to implement.

An HE-Tensor holds multiple HE Backend objects that represent the actual crypto objects. Since these objects can be large, creating copies would quickly fill up memory. An HE Tensor does not exclusively own the crypto object; it only holds references, making copies cheap. Since we have access to the entire computational graph, we can determine when an actual copy of the underlying crypto objects is needed.

The computation on encrypted data has two inputs: an HE-Tensor that holds the input data and the computational graph consisting of HLOs. The executor visits every HLO in the graph and performs the computation the HLO represents. It traverses the graph in such a way that it only visits nodes where all inputs are known, meaning if node B relies on the output of node A, the evaluator will visit A first.

3.3.4 Supported Operations

Out of the over 100 operations defined by the HLO specification, we only support a subset on encrypted data. Aluminium Shark supports all the HLOs on plaintexts. On ciphertext, we currently only support the operations required for neural network inference. These operations are: reshape, copy, convolution, broadcast, convert, multiply, add, transpose, and dot. We can group them into categories based on what they do and the restrictions they share. The first group is the binary element-wise operations. These HLOs take exactly two inputs and need to have the same shape; this way, the operation can be performed between each element of both operands. For neural network inference, we need the add operation to add the bias to the result of a layer. On encrypted data, we often use polynomials as activation functions, and to evaluate these, we also need multiplication. Another group of operations

concerns shape manipulation. These three operations are: reshape, transpose, and broadcast. Reshape changes the shape vector. It requires that the number of elements in the tensor stays the same. Reshaping is a very low-cost operation in most cases since it only requires setting a new shape vector and does not require any actual data movement. This is not the case when using batch layout, and the batch dimension of the shape changes. This requires more expensive computation and is not supported at the moment. Transposition is similar to reshaping, but it requires actually moving data. A transposition is a permutation of the shape vector, and the underlying data needs to be moved to reflect the shape vector. However, like reshaping, transposition is not supported when using the batch layout and moving the batch axis of the data.

The last operation of this group is broadcasting. Broadcasting adds another dimension to the shape vector and fills it by repeating data from other dimensions. For example, a scalar can be broadcast to an n-dimensional vector by repeating it n times, and a vector can be broadcast to a matrix by repeating it numerous times and stacking it horizontally. For example, the broadcasting operation is used to expand the bias vector of a layer so that it matches the shape of the resulting matrix, and an element-wise operation can be used. In most cases, the input to broadcasting is a plaintext, and when using the batch layout and when broadcasting is performed along the batch axis, the expansion of the plaintext prior to the encoding can be omitted.

The two most time and memory-consuming operations are convolution and dot. The dot HLO combines all contracting products between matrices and vectors, such as vector

dot products, matrix dot products, and the more general tensor product. Similarly, the convolution HLO is a general convolution and includes 1D, 2D, and higher dimensional convolutions. For both HLOs, we adapt the TensorFlow implementation so that they can handle HE-Tensors and different data layouts.

Supporting these HLOs allows us to run a variety of layers on encrypted data. We support fully connected layers, convolutional layers, and simple recursive layers. Many neural network architectures rely on only these layers. While Aluminium Shark supports recurrent layers, it is worth noting that they are very impractical due to their quickly increasing multiplicative depth, as highlighted by Podschwadt and Takabi [2, 3].

3.3.5 Supported Layouts

Aluminium Shark can use SIMD operations to make processing more efficient. The At the moment, we support two ciphertext data layouts.

Simple Layout is the easiest and most straightforward layout. In the simple layout, every ciphertext contains only one value. The number of ciphertexts is equal to the size of the tensor. Reshaping is trivial. However, for schemes that support SIMD, encoding only one value per ciphertext is highly inefficient. Mathematical operations in this layout are almost identical to plaintext in terms of algorithms. However, schemes supporting SIMD operations waste a significant amount of processing time. However, in these schemes, the simple layout can be helpful for debugging.

Batch Layout speeds up computation by encrypting multiple values into a single ciphertext. The batch layout packs the same feature from every instance into the same ciphertext.

Reshaping is trivial as long as the batch dimension is not part of the reshaping. The number of ciphertexts is equal to the number of features. If there are more instances in a batch then there are slots in a ciphertext, we need to split the data along the batch axis. Mathematical operations in this layout are almost identical to plaintext in terms of algorithms, as long as they do not perform computation along the batch dimension.

3.3.6 Optimization

We make several optimizations to improve the execution time and memory requirements of models run with Aluminium Shark. In XLA, all values are immutable. This has the benefit that we can compute operations in any other, as long as all the inputs are available and we adhere to the correct order of operations. We never need to worry that a required value might have changed in the meantime. However, this is a significant downside on encrypted data. Here, keeping intermediate results alive and treating them as immutable requires large amounts of memory. Therefore, we introduce two techniques to combat this problem. 1.) T o free up memory, we detect no longer-needed values that can be unloaded. Every HLO maintains a list of HLOs that require it as input as well as a list of HLOs that are inputs for it. After evaluating an HLO a we check all HLOs b that are inputs to a if they are used as input to any HLOs that have not yet been evaluated. If all HLOs that require b have been evaluated, we can unload b from memory; see Algorithm 2. 2.) Some operations we can perform in-place instead of creating and returning an intermediate result. These operations are element-wise tensor operations, like addition, multiplication, or negation. In the case of operations with two operands, we then need to decide which operator we want to contract

the operation into. If only one operand is a ciphertext, we choose that; otherwise, we choose the first eligible one. An operator is eligible for in-place operation if all the HLOs, besides the current one, that take it as input have been evaluated. In other words, we could unload it after evaluating the current HLO. Both of these techniques reduce the memory requirements of running a model on encrypted data.

Algorithm 2 Decide if an HLO can be unloaded from memory

Inputs: HLO a that was just evaluated **Outpus:** List d of HLOs that can be unloaded

```
1: a.inputs := all HLO that are an input to a
2: d := [ ]
3: for each b ∈ a.inputs do
4:     b.users := all HLOs that b is an input to
5:     for each c ∈ b.users do
6:         if c has not been evaluated then
7:             continue loop in line 3
8:         end if
9:     end for
10:    append b to d
11: end for
12: return d
```

We implement an analyzer that detects polynomials in the computational graph to properly support polynomial activation functions. The subgraph representing the polynomial can then be fused into a single operation and evaluated efficiently. Algorithm 3 shows an algorithm that can be used to determine if an HLO is the input to a polynomial. The algorithm explicitly does not detect monomials of degree two or lower. Once we identify a polynomial subgraph, we can execute it more efficiently. Normally, a polynomial is represented as a sequence of exponentiation, addition, and multiplication. We compute exponents by repeated multiplication, e.g., $x^5 = xxxxx$, which requires five multiplications. We can

reduce the depth to two multiplications by computing $x^5 = x^2x^2x$. We only need to compute x^2 once and can reuse it. Furthermore, we can reuse intermediate results in other terms of the polynomial, which would not be possible if we treated the terms as individual exponentiations.

Algorithm 3 Determine if a subgraph is a polynomial

Inputs: x HLO that we want to determine if the represents the variable of a polynomial
Outpus: If x is the variable of a polynomial subgraph

```
1: x.users := HLOs that x is an input to
2: x.nusers := number of users of x
3: if x.nusers ≤ 1 then
4:     return False
5: end if
6: ispoly := False
7: for each h ∈ x.users do
8:     if h.op = multiplication then
9:         if h.operand[0] = h.operand[1] or h.op = exponent then
10:            ispoly := True
11:        end if
12:    else if h.op ≠ addition or h.op ≠ subtraction then
13:        return False
14:    end if
15: end for
16: return ispoly
```

3.4 Evaluation and Experimental Results

To evaluate Aluminium Shark, we compare it to nGraph-HE. nGraph-HE is very similar to Aluminium Shark as it is another system that integrates into TensorFlow and allows the user run the model on encrypted data. We only use the homomorphic encryption component of nGraph-HE and do not enable the multiparty computation activation functions.

3.4.1 Data

As evaluation data we use the MNIST handwritten digit dataset [114] and the CIFAR-10 image dataset [115] and train multiple models on each dataset. On the MNIST dataset, we train a model with the Cyrptonets architecture [27] consisting of a convolutional layer and a fully connected layer before the output layer. It uses the square function as activation. The exact model architecture can be seen in Table 3.1. We further train two models with two convolutional layers on the MNIST data. The model we call 'small' uses 4 and 8 filters, and our 'large' model uses 40 and 80 filters. Both models use a square activation layer after each convolutional layer. For the full configuration see Table 3.2 for the small model and Table 3.3 for the large model. Finally, we train two models on the CIFAR-10 dataset. We also use the square activation in these models since Aluminium Shark does not support other polynomials at the moment. The first model, which we call CIFAR-10 model, consists of convolution layers, each followed by an activation layer (see Table 3.4). To evaluate our system on deeper models, we take the architecture of the CIFAR-10 model and add another fully connected layer followed by an activation layer. We also add a Batch Normalization layer before each activation. The detailed configuration is shown in Table 3.5.

3.4.2 Plaintext Performance

We train all models using TensorFlow 2.7 using the adam optimizer with standard parameters. The MNIST models we train for 32 epochs. The Cyrptonets model performs slightly worse on the plain test data (with 97.41% accuracy) than the small and large models (with

98.19% and 98.39% accuracy). On CIFAR-10, we take a random 10% (6000 instances) of the training data as validation data and train the models for 512 epochs. If the loss on the validation data does not improve for 10 epochs, we reduce the learning rate by a factor of 0.1; if the loss does not improve for 25 epochs, we end training. Both models achieve similar accuracy on the plain test data. The Model with Batch normalization achieves 72.67% accuracy compared to the model without, which achieves 71.26% accuracy.

3.4.3 Results on Encrypted Data

We run the experiments for both systems on a machine with two Intel(R) Xeon(R) Gold 6230R processors and 755GB of RAM. For nGraph-HE we use the latest version available on GitHub . We use the same crypto parameters for each system and ensure the security level is at least 128-bit. For all models except the CIFAR-10 with Batch Normalization, we use a polynomial modulus degree of 8192 and a coefficient modulus with 180 bits. For The CIFAR-10 model with Batch Normalization, we need larger parameters and choose a polynomial modulus degree of 16384 and coefficient modulus with 396 bits. The accuracy of all models on plaintext and encrypted data is nearly identical. There are slight variations in accuracy due to the approximate nature of CKKS, but these are less than 0.1% percentage points.

More interesting is the inference delay. We run all models five times on each system and average the execution time. Both systems can evaluate the small Cryptonets model in under 10 seconds, 4.95 seconds on a nGraph-HE, and 6.6s on Aluminium Shark. A batch size of

[6] `https://github.com/IntelAI/he-transformer` commit: 3ba6e51a59ee552685180c745b6d204d4bf2b64f

4096 leads to an amortized time per instance of 0.0012s and 0.0016s, respectively. Aluminium Shark is 25% slower than nGraph-HE. This slowdown remains constant for the more complex MNIST small model with 20.16s and 26.83s. However, as models grow more complex with the CIFAR-10 and MNIST large model, nGraph-HE becomes 31-32.5% faster. nGraph-HE can evaluate the CIFAR-10 model in 168.9 seconds and the large MNIST model in 372.8 seconds, whereas Aluminium Shark needs 244.7 seconds and 552.7 seconds. With the larger crypto parameters, on the CIFAR-10 with Batch Normalization model, this gap widens even further to 58%, with 479.8 seconds for nGraph-HE and 1153.7 seconds for Aluminium Shark. A full side-by-side comparison is shown in Table 3.6

This clearly shows that nGraph-HE is better optimized than Aluminium Shark, and we have more work to do in that area. However, the main focus of our system's design is usability. Moreover, in this area, we have an advantage over nGraph-HE. One of the main drawbacks of nGraph-HE is that it requires TensorFlow 1.15, which is at this point almost three years old and lacks several features that have been added to TensorFlow in the meantime. The most obvious is eager mode and support for cloud TPUs. Additionally, models trained in newer versions cannot always be loaded in older TensorFlow versions, requiring tedious manual conversion. Aluminium Shark is built on TensorFlow 2.7 with an easy path to update to newer versions. Part of the problem for nGraph-HE is that concepts in TensorFlow fundamentally changed between versions 1 and 2. The significant change was the redesign of graph mode and the switch to eager mode as the default execution. However, nGraph-HE relies heavily on graph mode, whereas Aluminium Shark relies on XLA, which is

Table 3.1: Architecture of the Cryptonets for MNIST Model

Layer	Configuration
2D Convolutional Layer 1	filters: 5, kernel size: 5 x 5, stride: 2 x 2, padding: valid
Activation	x^2
Fully Connected Layer	units: 100
Activation	x^2
Fully Connected Layer	units: 10

Table 3.2: Architecture of the MNIST small Model

Layer	Configuration
2D Convolutional Layer 1	filters: 4, kernel size: 5 x 5, stride: 1 x 1, padding: valid
Activation	x^2
2D Convolutional Layer 2	filters: 8, kernel size: 5 x 5, stride: 1 x 1, padding: valid
Activation	x^2
Fully Connected Layer	units: 10

not only integrated into TensorFlow but can also be used with other systems like PyTorch or JAX.[7] Further, to work with nGraph-HE, the computational graph must be frozen manually, and the output and input tensors need to be extracted manually. Aluminium Shark, on the other hand, can work with any TensorFlow or Keras model. There is no need to extract or freeze a graph manually. All the user needs to do is put the code that should be run inside a python function.

[7] `https://github.com/google/jax`

Table 3.3: Architecture of the MNIST large Model

Layer	Configuration
2D Convolutional Layer 1	filters: 32, kernel size: 3 x 3, stride: 1 x 1, padding: valid
Activation	x^2
2D Convolutional Layer 2	filters: 64, kernel size: 3 x 3, stride: 1 x 1, padding: valid
Activation	x^2
Fully Connected Layer	units: 10

Table 3.4: Architecture of the CIFAR-10 Model

Layer	Configuration
2D Convolutional Layer 1	filters: 40, kernel size: 5 x 5, stride: 2 x 2, padding: valid
Activation	x^2
2D Convolutional Layer 2	filters: 80, kernel size: 3 x 3, stride: 2 x 2, padding: valid
Activation	x^2
Fully Connected Layer	units: 10

Table 3.5: Architecture of the CIFAR-10 Model with Batch Normalization

Layer	Configuration
2D Convolutional Layer 1	filters: 40, kernel size: 5 x 5, stride: 2 x 2, padding: valid
Batch Normalization	-
Activation	x^2
2D Convolutional Layer 2	filters: 80, kernel size: 3 x 3, stride: 2 x 2, padding: valid
Batch Normalization	-
Activation	x^2
Fully Connected Layer	units: 100
Batch Normalization	-
Activation	x^2
Fully Connected Layer	units: 10

3.5 Limitations

Aluminium Shark comes with a few limitations. Some restrictions are rooted in the encryption schemes used, while others come from not yet having implemented desirable functionalities. The most significant restriction is that we do not support arbitrary neural networks. This includes networks that follow state-of-the-art practices on the plaintext. The following

Table 3.6: Latency and accuracy of all tested models

Model	System	Latency	Latency Increase	Accuracy	Accuracy plain
Cryptonet MNIST	Aluminium Shark nGraph-HE	6.60s 4.95s	24.93%	97.41% 97.41%	97.41%
MNIST Small	Aluminium Shark nGraph-HE	26.83s 20.16s	24.84%	98.19% 98.19%	98.19%
MNIST Large	Aluminium Shark nGraph-HE	552.70s 372.81s	32.55%	98.39% 98.34%	98.34%
CIFAR-10	Aluminium Shark nGraph-HE	244.77s 168.90s	31.00%	71.19% 71.29%	71.26%
CIFAR-10 with Batch normalization	Aluminium Shark nGraph-HE	1153.68s 479.86s	58.41%	72.67% 72.50%	72.67%

subsections describe some of these limitations in more detail.

3.5.1 Activation Functions

We cannot run networks that use comparison operations, like $<, >$, or $==$. This disqualifies networks that use ReLU $max(0, x)$ activations. We further cannot compute encrypted exponents, which eliminates popular activation functions like, $tanh(x) = \frac{e^x+e^{-x}}{e^x-e^{-x}}$, $sigmoid(x) = \frac{1}{1+e^{-x}}$, or $softmax(x)_i = \frac{e^{x_i}}{\sum_{j=1}^{n} e^{x_j}}$ for $x \in \mathcal{R}^n$ and $i = 1, \ldots, n$. These, however, are known issues in the PPML community and numerous solutions have been proposed. Softmax, when it is the last operation in the network, is often omitted and left to the client. Other activation functions are often replaced with polynomial approximations. At the moment the support for polynomial activations is limited to monomials of the form $ax^n + b$. The optimizer cannot detect polynomials yet and too aggressively replaces them with in-place operations, breaking the output.

To address the issue of the ReLU function, we will implement TFHE as a backend. With TFHE, it is possible to implement efficient comparison, and therefore, the ReLU function, as shown by Lou et al. [58].

3.5.2 Optimization

As we can see from the experiments Aluminium Shark, there is still room for further optimizing secure inference. One of the issues is the naive application of ciphertext maintenance operations. At the moment, Aluminium Shark uses relinearization and rescaling rather naively after every ciphertext multiplication. nGraph-HE uses these operations more

sparsely, thereby reducing execution time. Right now, we also do not use any depth-reducing optimizations. Improvements like constant folding can further bring down the runtime.

One of the most considerable improvements in inference latency has been the introduction of sophisticated ciphertext packing strategies like the ones presented by Brutzkus et al. [63] and Jiang et al. [61]. These packing strategies reduce the number of ciphertexts, and thereby ciphertext operations, drastically speeding up the inference process. One of the next milestones in our development is the support of these packing strategies.

3.6 Summary and Discussion

In this chapter, we presented Aluminium Shark, an XLA backend that allows users to run machine learning models on encrypted data easily with TensorFlow. Execution on encrypted data only requires a few lines of extra code and is entirely done in Python, with no C++ necessary. We use SIMD to speed up the computation and have designed a serial representation for the encoding scheme that can easily be shared between data and model owner. Currently, Aluminium Shark supports batch encoding, but we aim to implement more efficient encodings in the future. With more encodings, we also plan to introduce an optimizer that finds the scheme and compilation time. We further aim to reduce the HE knowledge required by providing support for automatically analyzing the computational graph and generating suitable crypto parameters, which is a task that currently needs to be performed manually.

CHAPTER 4

MEMORY EFFICIENT PRIVACY-PRESERVING MACHINE LEARNING BASED ON HOMOMORPHIC ENCRYPTION

In Section 3.3.6, we presented a strategy for unloading data from graph nodes we no longer need. This memory management happens after the computation of one layer is complete and before the computation of the next layer starts. We now extend the memory management to take place during the computation of a layer, reducing the resource requirements of the layer itself. This is not possible with the approach in Section 3.3.6. It only "cleans up" once a layer has been entirely computed.

This chapter is currently under review at *22nd International Conference on Applied Cryptography and Network Security, 5-8 March, 2024*. Further discussion that is not part of the submission starts in Section 4.6.

4.1 Introduction

Machine learning (ML) and neural networks specifically are widely deployed in many different scenarios, from voice assistants like Siri[116], Alexa[117], and Google Assistant[118] over writing assistants like Grammarly[119], and chatbots like Bard[120] and ChatGPT[121], to medical diagnostic systems [122, 123]. Many of these systems deal with privacy-sensitive data, some of which enjoy special legal protections, e.g., medical data. These systems send the data to a server, which runs it through its model and returns the result to the client. Since the server needs access to the unencrypted client data to perform the computation, the client's privacy is at risk. The server might use the data to train further ML models, which

could expose the data to privacy attacks, or the server itself could be breached and the data stolen. Researchers have recently proposed solutions to protect user data privacy in ML applications using different methods. Differential Privacy [7] solutions preserve the privacy of the training data in the trained model [11, 12]. To protect the data during inference, solutions commonly use Secure Multiparty Computation (SMC)[16, 17, 18], Homomorphic Encryption [124, 95, 28] or a mixture between the two [108]. SMC allows multiple parties to jointly evaluate a function without revealing their private inputs; however, it requires all parties to stay online during the computation. HE, on the other hand, can be used entirely offline. HE is a type of encryption that allows computation on encrypted data without exposing any inputs, intermediate, or final results. Neural networks are a popular choice for privacy-preserving ML models since most operations, like fully connected layers or convolutions, can be performed easily using HE. Additionally, neural networks perform very well on a wide range of tasks. However, HE introduces significant time and memory overhead. Some HE schemes support single instruction multiple data (SIMD) processing, which can offset some time and memory overhead. HE ciphertexts can be thought of as fixed-sized encrypted vectors containing thousands of elements, called slots. Two approaches for filling the slots have been used for ML. 1.) Pack all the features of an instance into as few ciphertexts as possible and perform convolutions and dot products with the help of rotations [63, 28, 84], called inter-axis packing. This has the advantage that the number of ciphertexts and total operations is relatively small, making it fast for a small number of instances. However, this approach often requires large rotation keys, and the rotations require additional time. 2.)

Pack multiple instances' features into a single ciphertext[27, 125, 3], called batch-packing. This produces as many ciphertexts as the data has features. Batch-packing allows us to simultaneously compute results for many instances, leading to low amortized per-instance cost and high throughput. However, it suffers from high latency and memory requirements. Batch-packing is beneficial when many instances need to be processed, and low latency is not essential. For example, in a medical image diagnostic system, where images are collected throughout the day, and an ML system analyzes them overnight. This work focuses on convolutional neural networks (CNN), specifically. We address the memory requirements for convolutional layers by trading disc space for main memory. Disc space is typically orders of magnitude cheaper. However, it is also slower. We dynamically load ciphertexts and plaintexts and clear them from memory when no longer needed. We present and compare different strategies and their impact on memory and runtime. Prior work focuses primarily on latency reduction; reduction in memory is often a side effect of inter-axis packing. To the best of our knowledge, this is the first study that performs an in-depth analysis of caching strategies and memory reduction for batch-packed inference. Brutzkus et al. [63] or Lee et al. [28] propose input packing techniques, which reduce the number of ciphertexts and thereby memory requirements. However, these approaches require additional operations like masking and rotation, which lower the overall throughput. Boemer et al. [108] present a complex encoding, allowing them to fit more values into a ciphertext. This can reduce the number of ciphertexts and plaintext when using inter-axis packing. However, for batch-packing, it only affects the batch size. Approaches that use client interaction, such as Boemer et al. [108],

Podschwadt et al. [2], and Cai et al.[126] can often use smaller crypto parameters, since the client interaction resets the noise level, allowing for further computation. However, these approaches require the client to be online during the computation. We make the following main contributions:

- We propose a schedule representation for convolutions that allows us to reorder its fundamental operations to achieve increased caching performance.
- We propose a memory estimation algorithm for schedules.
- We propose an algorithm for executing a schedule using multiple threads.
- We propose multiple strategies for creating schedules, which we analyze and experimentally evaluate with regard to their time and memory requirements.

4.2 Related Work

Akavia et al.[127] focus on reducing the storage footprint of HE ciphertext rather than the in-memory size during computation. They design a protocol that allows multiple data producers to upload and store data in the cloud with no overhead compared to storing AES (Advanced Encryption Standard) encrypted data. Storing AES encrypted on an untrusted server and using secret sharing, a computing server can use the data for HE computation with the help of an auxiliary server. In contrast, our proposed solution reduces the memory footprint at computation rather than the encrypted storage size.

Jiang et al. [61], Brutzukus et al. [63], Lee et al. [28], Dathathri et al. [64], and Lee et al. [95] are conceptually similar works, who all reduce the number of ciphertexts required

by using inter-axis packing. While all these approaches reduce the inference latency, they require expensive rotations, lowering the throughput compared to batch-packed solutions. Additionally, they often rely on designing the packing strategy for the specific network architecture.

Other studies rely on interactive solutions for privacy preservation. Hao et al. [128] and Huang et al. [129] both propose efficient matrix multiplications in a two-party setting. Both studies propose rotation-free matrix multiplication over polynomial encoded ciphertexts. However, both require interactive phases where one party must extract specific polynomial coefficients and mask the result. Zheng et al. [130] propose a method for fast private inference using transformers and SMC. The authors use a similar protocol to the one proposed by Juvekar et al. [131], where the server performs much of the expensive matrix multiplication computation in an offline phase. Zheng et al. [130] reduce the number of ciphertext rotations required by packing the same feature of different tokens into the same ciphertext, similar to the batch-packing we use in our approach. However, we compute the intermediate terms in a less memory-consuming way.

Prior work on batch-packed PPML using HE [27, 66, 26, 69] does not explicitly state how they perform matrix multiplication or convolutions. They focus on other improvements like better polynomial approximation [26, 66], or parameter fusion and special value bypass [69]. We believe most of these solutions could decrease memory requirements using our proposed algorithm. Another work that addresses memory limitations is Badawi et al. [67], which implements a CNN over HE data using GPU acceleration for the basic ciphertext operation.

To fit the input to the convolution into GPU memory, they split it into multiple blocks of the same size as the filter. The filter and as many of these blocks as possible are loaded into GPU memory, where the convolution is performed. Compared to our proposed approach, this process only reduces the memory requirement on the GPU. The input and weights still need to be present in the main memory. Shivdikar et al. [132] also present techniques aimed at GPUs. They aim to reduce the repeated memory reads inside the GPU when performing polynomial multiplication for HE primitives. While this speeds up the low-level operations underpinning most HE schemes, unlike our work, it does not address the issue of requiring a large number of plaintexts or ciphertexts in memory.

4.3 Our Proposed Approach

To address the issues of memory consumption and unused resources, we model the convolutional layer as a schedule, which determines the order of operations. We present and compare multiple schedule construction strategies based on the computation and available resources. We further present an algorithm to execute a schedule. From now on, we assume that all tensors are flattened.

4.3.1 Batch Packing

We consider n_s to be the number of slots in a ciphertext. For simplicity, we assume that the batch size is equal to n_s. Otherwise, we would need to split the data into multiple batches or pad it. We partially flatten all dimensions in X except the batch dimension to encrypt the inputs. We take each column from the resulting two-dimensional matrix and encrypt that

into a ciphertext, leaving us with a vector of ciphertexts. We need to encode the weights as well. Each weight value in W is encoded into its own plaintext. Before encoding, we turn each value into a vector by repeating it n_s times. This produces $|X|/n_s$ ciphertexts and $|W|$ plaintexts. If the model needs to be encrypted as well, we can encrypt the encoded plaintext weights. Similarly, the encoding of W contains $|W|/n_s$ ciphertexts. We can think of the encoding as setting the batch axis to one. The issue that arises is that HE ciphertexts and plaintexts require a substantial amount of memory. A single ciphertext can be between a few hundred kilobytes to multiple megabytes, depending on the crypto parameters; a plaintext is half the size of a ciphertext. We refer to the encoded and/or encrypted inputs, weights, and outputs as X', W', and Y', respectively, and values taken from them as $\mathbf{x}'$, $\mathbf{w}'$ and $\mathbf{y}'$.

4.3.2 Modeling the Problem as a Schedule

We can write each element $\mathbf{y}$ as a sum of products of $\mathbf{x}$ and $\mathbf{w}$. We denote a product of two elements of $\mathbf{x}$ and $\mathbf{w}$ as the triple $t = (\mathbf{x}, \mathbf{w}, \mathbf{y})$, where $\mathbf{y}$ is the result that holds the sum that the product $\mathbf{xw}$ is a part of. To refer to an element in a triple t, we use the following notation $t^i; i \in \{x, w, y\}$.

Definition: Schedule Let f be a convolutional layer; we say $t_i \in f$ iff the sum to compute t_i^y contains the product $t_i^x t_i^w$. A schedule is an ordered list of triples t_i that contains all $t_i \in f$ exactly once.

In other words, we represent f as a sequence of all its element-wise products. To compute the function f, we need to compute all products given in the schedule. Additionally, we must sum all products with the same value for $\mathbf{y}$. We call the number of triples in a schedule the

Algorithm 4 Generating a schedule from a 2D Convolution

Inputs: Output shape $h_{\text{out}}, w_{\text{out}}, c_{\text{out}}$, Input shape $h_{\text{in}}, w_{\text{in}}c_{\text{in}}$, Filter size w_h, w_w
Output: The schedule S

1: $S := [\,]$
2: **for each** $i \in [1, \ldots, h_{\text{out}}w_{\text{out}}c_{\text{out}}]$ **do**
3: convert i to multi-dimensional index (m, n, t^y)
4: **for each** $j \in [1, \ldots, w_h w_w c_{\text{in}}]$ **do**
5: convert j to multi-dimensional index (p, q, r)
6: $t^x := m - p + ((n - q)w_{\text{in}} + (rh_{\text{in}}w_{\text{in}})$
7: $t^w := p + (qw_h) + (rw_w w_h) + (rw_w w_h c_{\text{in}}))$
8: append (t^x, t^w, t^y) to S
9: **end for**
10: **end for**

length or steps of a schedule, denoted by $|S|$. Algorithm 4 shows how to generate a schedule for two-dimensional convolutions. Higher dimensional convolutions work analogously by expanding the iteration bounds in lines 2 and 4, the decomposition of i and j in lines 3 and 5, and the formula for t^x and t^w by the extra dimensions. In addition to the computation steps, we also insert load instructions into the schedule. Load instructions specify which elements to load into memory, discard from memory, or write back to disk in case they were updated.

4.3.3 Executing a Schedule

To execute a schedule, we evaluate all triples in order. To evaluate a triple t we multiply the input X_{t^x} with the weight W_{t^w} and add the result to Y_{t^y}; $Y_{t^t} = Y_{t^y} + X_{t^x}W_{t^w}$. We assume that all $\mathbf{y}$ are 0 at the beginning. We parallelize the execution of the schedule across multiple threads. Each repeatedly evaluates the first unevaluated triple. This requires synchronization at two points. 1.) We must ensure that every triple is evaluated exactly

once. 2.) Unlike in Algorithm 1, we cannot guarantee that multiple threads do not write to the same output; therefore, we need locking to prevent race conditions. We use the following algorithm to ensure all values are correctly summed into the output value. We show our proposed algorithm in Algorithm 5. The parts that must be protected from concurrent access are marked as Critical Section.

Algorithm 5 Executing a schedule

Inputs: inputs X, weights W, output Y, number of threads n_t, schedule S
Outputs: Y containing the result of the convolution

1: ensure $Y_i = \varnothing; \forall i \in [1, |Y|]$
2: $i_s := 1$
3: **while** $i \leq n_t$ **and** $i \leq |S|$ **do**
4: **Start Thread** i and execute:
5: **while** $i_s \leq |S|$ **do** } Critical
6: $j := i_s$ } Section
7: $i_s := i_s + 1$
8: perform load instructions
9: $t := S_j$
10: $r := X_{t^c} \cdot W_{t^p}$ ▷ HE multiplication
11: **if** $Y_{t^o} = \varnothing$ **then**
12: $Y_{t^o} := r$ } Critical
13: **else** } Section
14: $v := Y_{t^o}$
15: $Y_{t^o} := \varnothing$
16: $r := v + r$ ▷ HE addition
17: **goto** line 11
18: **end if**
19: **end while**
20: **End Thread**
21: **end while**

We indicate where in the algorithm process load instructions in line 8. A load instruction has three attributes: 1.) the step that is executed on, 2.) the type of instruction, load or unload, and 3.) the object to load. Every iteration, each thread checks if there is

an unprocessed load instruction with a step equal to or lower than the step the thread is executing. If there is, the thread marks it complete and executes it. Again, we must ensure that only one thread updates the load instructions at any time. Each thread tries to execute any outstanding load instructions before moving on. Objects loaded through load instructions stay cached until explicitly unloaded through another load instruction or until the computation is complete. If a thread requires values not loaded by any load instructions, it loads them on demand and does not cache them.

4.3.4 Cost of a Schedule

We can use the schedule to estimate the maximum memory required on encrypted data. Maximum memory is important since we cannot execute the schedule if it requires more than the available memory. Most of the memory required during execution stems from the ciphertexts and plaintexts; therefore, we ignore additional objects like keys, the schedule, and other data in our estimation. To estimate the cost, we look at the load instructions, the number of threads, and the objects loaded on demand. We first examine the simpler case with only one thread and extend it to multiple threads later. Let s_x be the size of a single $\mathbf{x}'$, s_w the size of a single $\mathbf{w}'$, and s_y the size of a single $\mathbf{y}'$. To estimate the memory requirement of a schedule, we need to perform the following steps:

1. Split the schedule into parts at the load instructions so that each part begins with load instructions and contains no other load instructions except those at the beginning. A part must not only contain load instructions.

2. For each part, count how many $\mathbf{x}'$, $\mathbf{w}'$, and $\mathbf{y}'$ are loaded and unloaded.

3. Weight the count of $\mathbf{x}'$, $\mathbf{w}'$, and $\mathbf{y}'$ by s_x, s_w, and s_p, respectively.

4. For every step, weigh the on-demand loaded objects and add them.

5. For each part, add the weighted counts from step 2 and the maximum from step 4. The maximum of all the parts is our estimate for the schedule.

We now extend the estimation to multiple threads. The estimate for multiple threads is less precise than that for a single thread since we can only make assumptions about how multiple threads will interact. We make the following simplifying assumptions: 1.) threads execute schedule steps at the same speed, and 2.) a continuous block of load instructions is executed simultaneously, no matter how many instructions are in that block. We start as we did in the single thread case above. Next, we look at the number of steps in each part. If the part has fewer steps than the number of threads n_t, we combine it with the next part to form a new part by adding the cost of the load instructions. We repeat this until the new combined part has more steps than threads. We repeat this for all parts of the schedule. To estimate the cost of the on-demand loaded objects, we assume that n_t schedule steps are executed at the same time. In the final step, we handle the cost of the schedule steps. We compute the on-demand cost for all steps in the schedule parts created in the previous step. The computation happens the same way as described in the single-threaded case above. However, now we not only add the step with the highest cost; we add the n_t

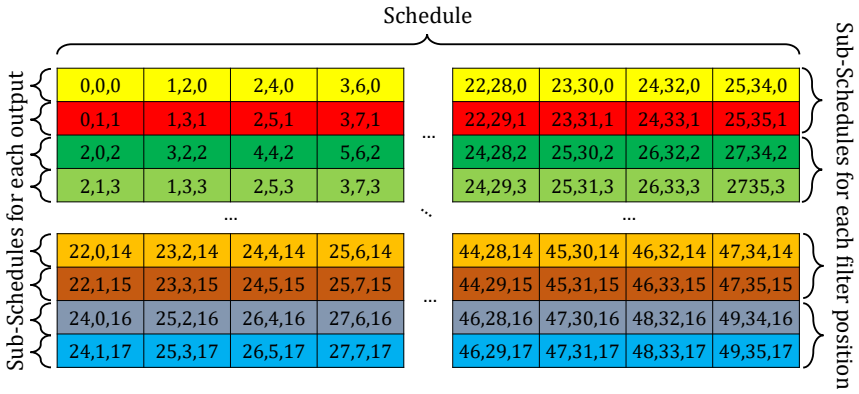


Figure 4.1: Breaking an example base schedule down into multiple sub-schedules. This schedule is executed row-wise.

steps with the highest costs. This method provides a reasonable estimate for the memory cost of a schedule with multi-threaded execution.

4.4 Reduced memory schedules

In this section, we propose different ways to construct schedules. These schedules provide trade-offs between runtime and memory. The fastest we can execute a schedule is by loading all data at the beginning of the computation and then using the lock-free algorithm 1. However, this requires a large amount of memory. We can reduce the memory footprint by loading everything on demand. However, this increases runtime significantly.

We can transform the computation performed by Algorithm 1 into a schedule. Again, consider n_t to be the number of threads, n_o the number of outputs of the computation, and n_f the number of products that sum up into a single output. In algorithm 1, every thread executes a subschedule where all $t^y \mod i = 0; t \in S$, where i is the thread id. We can obtain the combined schedule by taking all subschedules and interleaving them elementwise.

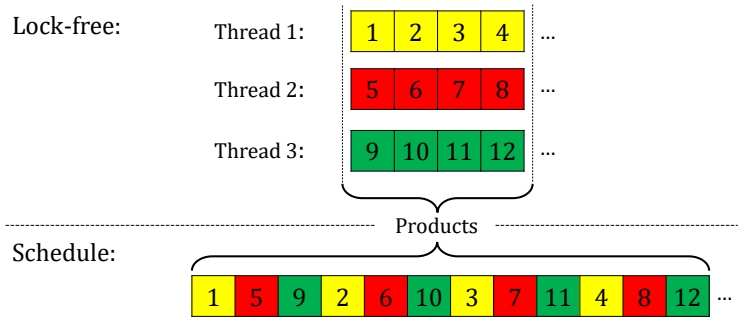


Figure 4.2: Example of how to turn a lock-free execution with three threads into a schedule

See Fig. 4.2 for an example with three threads.

The lock-free algorithm computes and needs to keep in memory n_t outputs simultaneously. The base schedule, on the other hand, fully computes a single output before moving on to the next one. This allows us to keep fewer outputs in memory. This is the lowest amount of memory we can achieve. However, we need to load objects from disk frequently and are not using any caching. Caching aims to reduce the number of loading operations as much as possible. We can exploit the regular structure of convolutions to find the best values for caching. We can split a schedule into a regular, repeating pattern defined by the size and number of filters and input channels. In two-dimensional convolutions, as used in neural networks, we have a four-dimensional filter volume, W, where the dimensions are in order: i, j the position in the filter, c_{in} the input channel, and c_{out} the output channel. We move W across the entire input, creating c_{out} outputs at every position. Note how far W we move the filter is given by the stride, which we assume to be one here. However, our method remains applicable to other stride values. Each output, at a given position of W, uses the same values from X. We call each unique position of the filter on the inputs the filter position or window.

We need to keep three kinds of objects in memory during the computation. Inputs $\mathbf{x}$, weights $\mathbf{w}$ and, outputs $\mathbf{y}$. We design multiple caching strategies based on the memory available. We will not go over the trivial case that we can fit all values of $\mathbf{x}$ and $\mathbf{w}$ into memory.

4.4.1 Caching by Object Type

The simplest caching strategy, is to load either all values from X' or W' at the beginning of the computation and load the other values on demand as needed. This strategy creates very simple schedules; however, it underutilizes caching. If we preload all $\mathbf{x}'$, we load too many values much earlier than needed in the computation, and if we preload all $\mathbf{y}'$ we need to load $\mathbf{x}'$ values frequently.

4.4.2 Full Window Caching

We can improve the caching by object type strategy by utilizing the underlying structure of the convolution operation. Each position of W requires only $|W'|/c_{\text{out}}$ $\mathbf{x}'$. If we can fit these objects and all $\mathbf{y}'$ into memory, we only load W' once. Since W' usually moves over the inputs with some overlap, i.e., the stride is smaller than the width and height of the filter, we can reuse many $\mathbf{x}'$ and only need to unload and reload a small amount. We start at the top left and move W from left to right. Once we reach the end on the right, we move down and start over on the right, repeating until we reach the bottom right. If the filter size or stride is not symmetrical, it is beneficial to change the behavior to first move in the direction that has the most overlap, reducing the number of values that need to be loaded

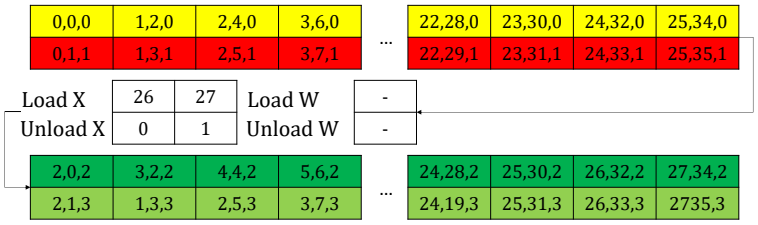


Figure 4.3: Load instructions that are necessary when moving from the first window to the second using window caching with a 5x5x2 input and a 3x3x2x2 kernel.

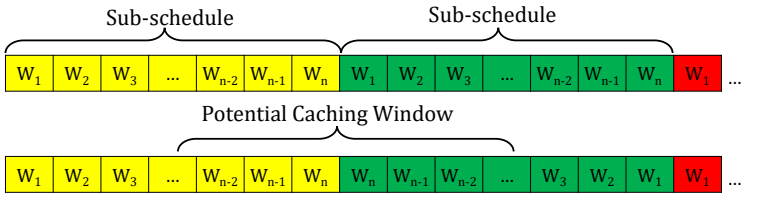


Figure 4.4: Weights W_i only in Sub-Schedules that correspond to individual filter positions and how they can be reordered to increase caching potential.

and increasing the number of values that can be reused.

4.4.3 Partial Window Caching

If we can fit $|W'|/c_{\text{out}}$ $\mathbf{x}'$ but not all of W' into memory, we can modify the full window caching strategy to reduce the number of loads. Let n be the number of $\mathbf{w}'$ that we can fit into memory in addition to all the $\mathbf{x}'$ in the window. We then split the schedule into sub-schedules for every position of W. To reorder the sub-schedules to increase caching potential we reverse every second sub-schedule; see Fig. 4.4. This reordering makes it so that i values to the left and right of the sub-schedule boundary are the same for all $i \in [1, |W|]$. This allows us to cache n values before the sub-schedule boundary and reuse them in the next one.

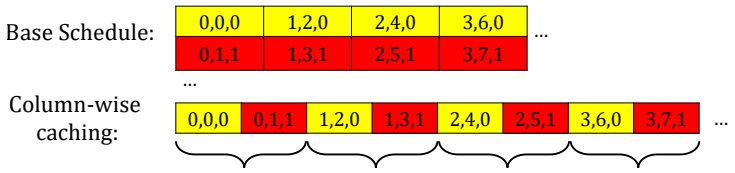


Figure 4.5: Transforming the base schedule into a column-wise caching schedule

4.4.4 Column-wise caching

If we cannot fit $|W|/c_{\text{out}}$ $\mathbf{x}'$ or W' into memory, we cannot use any of the caching methods described above. However, we can construct a different schedule that allows us to cache $\mathbf{x}'$ values. For this schedule, we need to be able to fit c_{out} $\mathbf{y}'$ into memory. By taking each window sub schedule and reordering it column-major instead of row-major, see Fig. 4.5, we can reuse the same $\mathbf{x}'$ multiple times before we unload it. This ordering requires us to keep c_{out} $\mathbf{y}'$ in memory. This ordering is most beneficial when the number of input channels is much larger than the output channels or the filter is relatively large. Both scenarios lead to a large number of $\mathbf{x}'$ in a window. Depending on how much memory is available, we can cache multiple columns. Additionally, we can combine this with the idea from partial window caching of reordering the computation to generate adjacent window subschedules that end and start with the same X values.

4.5 Evaluation and Experimental Results

4.5.1 Data

We evaluate our proposed solution on the layers of a convolutional neural network (CNN) trained on the CIFAR-10 [115] dataset. Table 4.1 shows the model's architecture. We have

Table 4.1: Architecture of the evaluation model with the layer parameters showing the filter size (FS), stride (S), number of filters (NF), and the activation or pooling function used on plain text (PT) and on encrypted data (HE).

Layer	Input Shape	Output Shape	Parameters
Conv 2D (1)	32x32x3	30x30x32	FS: 3x3, S: 1x1, NF: 32, PT: ReLU, HE: x^2
Pooling	30x30x32	15x15x32	FS: 2x2, S: 2x2, PT: Max, HE: Average
Conv 2D (2)	15x15x32	13x13x64	FS: 3x3, S: 1x1, NF: 64, PT: ReLU, HE: x^2
Pooling	13x13x64	6x6x64	FS: 2x2, S: 2x2, PT: Max, HE: Average
Conv 2D (3)	6x6x64	4x4x64	FS: 3x3, S: 1x1, NF: 64, PT: ReLU, HE: x^2
Flatten	4x4x64	1024	-
Dense	1024	64	Units: 64, PT: ReLU, HE: x^2
Dense	64	10	Units: 10

two different models. One for plain data and one adapted to be HE-friendly, meaning it only contains operations that are easy to compute on encrypted data. Both models achieve very similar accuracies on the test data, 70.9% for the original model and 69.7% for the HE-friendly model. The main interest of this chapter is not to propose new models or techniques that increase the accuracy of models on encrypted data but to analyze and reduce the memory consumption of these models.

We define three sets of crypto parameters: small, medium, and large. The small parameters have a ring dimension of 2^{14} and a multiplicative depth of 2. The medium parameters have a ring dimension of 2^{14} and a multiplicative depth of 8. And the large parameters have a ring dimension of 2^{15} and a multiplicative depth of 19. This results in a ciphertext size of 0.75 MB, 2.225 MB, and 10 MB for the small, medium, and large parameters respectively. A plaintext is always half the size of a ciphertext. We have two machines. One with 16 cores, 20 GB of memory, and 32 GB of operating system (OS) swap space, and another with 104 cores and 768 GB of memory. Both machines have two TB solid-state drives. We define different schedules, then estimate the required memory using the technique described

in section 4.3.4, and finally execute the schedules to obtain real measurements.

We define several schedules that we estimate and measure the memory requirements for. The names of the schedules are given italicized. We use the Lock-free algorithm (Algorithm 1) as our baselines once we load all values on demand (*Lock-free on demand*) and once we preload all values before execution *Lock-free Preload*. We compare these baselines to their direct equivalent using our proposed algorithm (Algorithm 5), where we preload all values (*Preload everything*). Next, we investigate the behavior when we either preload all of X', *Preload* X', or all W' values *Preload* W', x' *on demand*. Finally, we look closer at the window, partial window, and column-wise caching. For (partial) window caching, we always load all of $\mathbf{x}'$ in the window and investigate the following strategies for loading $\mathbf{w}'$s:

- load all of W', *Load* X' *window* W'
- load $\mathbf{w}'$s on demand, *Load* X' *window,* w' *on demand*
- load half of W' values, *Load* X' *window,* $W'/2$
- load a quarter of W', *Load* X' *window,* $W'/4$

We only cache one $\mathbf{x}'X$, *Column Major* for column-wise caching. For all schedules, we cache the y's from their first appearance in the schedule to their last.

4.5.2 Memory Estimate

To demonstrate that our proposed solution is scalable from large servers to consumer hardware, we run the selected schedules on two different machines. A desktop PC with a 16-core

AMD Ryzen CPU, 20 GB of RAM, 32 GB of swap space, and a large server with two Intel 54-core CPUs and 756 GB of RAM. Both machines have a 2 TB solid-state drive and run Ubuntu Linux 20.04 LTS. In the tables and figures throughout this chapter, we refer to the server and PC by their number of threads: 104 and 16, respectively.

We use the algorithm described in Section 4.3.4 to estimate the cost of all convolutional layers for small, medium, and large parameters and 16 and 104 threads. We need to estimate the memory requirements based on the number of threads that are used during execution since that can influence the number of objects in memory. The estimate column in Table 4.8, 4.9, and 4.10 shows the estimates for each layer and schedule for large parameters (for the small parameters see the appendix). We can see that, especially for the large parameters, the estimate frequently goes beyond the 20 GB of the PC. The estimate also often exceeds the 52 GB of memory and swap space combined. The estimate never exceeds the 756 GB of the server. For the estimate and following experiments, we assume the input X' is encrypted while the model W' is in plain.

Unsurprisingly, the schedules that preload all objects, *Preload everything* and *Lock-free Preload*, have the highest memory estimate. On the other hand, schedules that load most objects on demand and cache very little, *Lock-Free on demand* and *Column Major* have the lowest memory estimate. For the Conv 2d (1) layer, the estimates range from 380 MB to about 35 GB. Schedules that do not load all of X' are significantly below that value, estimated at most 6193 MB. For the second layer, Conv 2D (2), both the number of $\mathbf{x}'$ and $\mathbf{w}'$ is significantly larger. This, however, does not significantly change the estimate for

the *Lock-Free on demand* schedule. This observation also holds for the next layer, Conv 2D (3). The estimation aligns with the insights of a theoretical analysis of the execution. As discussed earlier, during runtime, this schedule has at most n of each $\mathbf{x}'$, $\mathbf{w}'$, and $\mathbf{y}'$ in memory, where n is the number of threads. Therefore, the memory consumption of the schedule is only influenced by the number of threads and independent of the layer. For the Conv 2D (2) layer, we also encounter values outside the PC's available memory, ranging from 400 MB to 164 GB. We see a similar picture for the last convolutional layer, Conv 2d (3). Large estimates of up to 208 GB, especially for layers that load and cache W' values.

4.5.3 Measurements

After obtaining the estimates, we execute the schedules on both the server and the PC. We measure the time it takes to execute the schedules, and the memory the process requires. For the memory measurement, it is important to note that it does include swap memory and only measures actual main memory usage. The PC has 20 GB of memory, about 1.5 GB of which the OS uses, leaving about 18.5 GB for the execution of the schedule. Therefore, measurements in the range of 18.5 GB on the PC will likely have used the OS's swapping mechanism, especially if the estimated value is much larger. As mentioned in the previous section, for some schedules, the memory available is insufficient, even with swapping. In these cases, the execution is terminated by the OS, yielding no result. We deliberately leave the OS swapping mechanism on to test if our implementation is faster than simply relying on the in-built OS methods. We further assign each schedule a score combining time and memory requirements. To calculate the score, we compute the geometric mean of the time

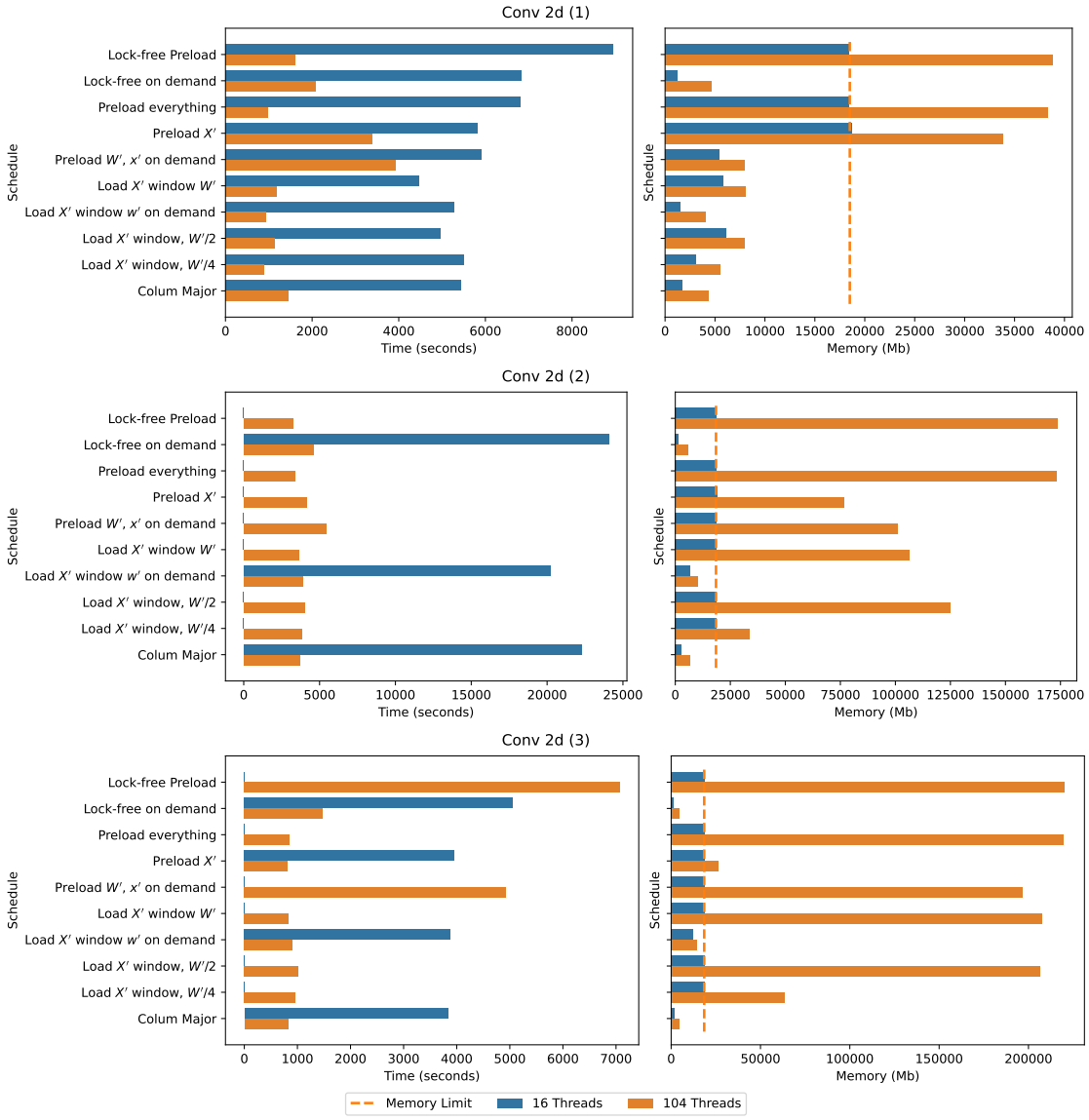


Figure 4.6: Time and Memory requirements schedules, run with large parameters, on the 104 threads servers and the 16 threads PC. The memory graphs also include the PC's memory limit of 18000 MB.

t and m as $\sqrt{tm}$. The lower the score, the better. However, the schedule with the lowest score is automatically the best schedule on a given machine. The best schedule is typically the schedule that executes the fastest on the machine. It is possible for a slower schedule to achieve a lower score due to it requiring less memory. This, however, indicates that we

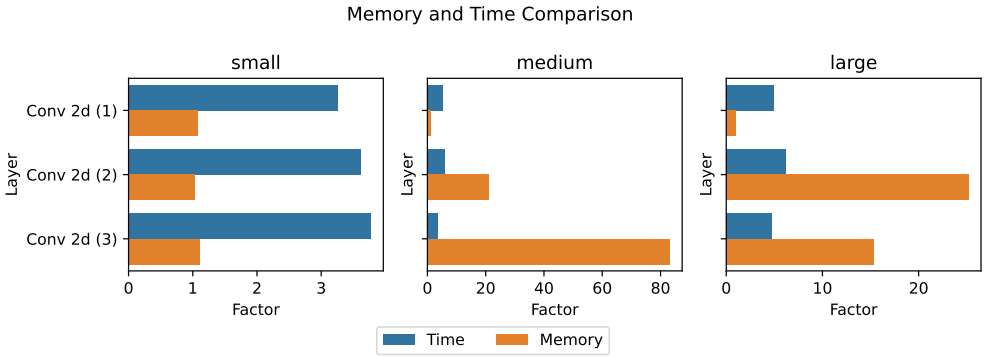


Figure 4.7: Comparison of the fastest schedule for each layer with 16 and 104 threads. For each layer, the Figure shows the increase factor in runtime from 104 to 16 threads and the increase factor in memory from 16 to 104 threads for the fastest schedule

Table 4.2: Measurements for **Time** in seconds, Memory (**Mem**) in MB, for all Schedules on Conv 2d (1) with small parameters on the PC wiht 16 and the server with 104 Threads. Additionally, the table shows the memory Estimate in MB and **Score**.

Schedule	Threads	Time	Memory (MB)	Estimate	Score
Lock-free Preload	16	136	3084	2645	648
	104	42	3319	2711	372
Lock-free on demand	16	433	290	30	354
	104	131	652	195	292
Preload everything	16	142	3024	2634	656
	104	45	3316	2695	387
Preload X'	16	184	2593	2316	691
	104	52	2872	2409	385
Preload W', x' on demand	16	333	724	338	491
	104	108	1027	465	333
Load X' window W'	16	143	771	347	332
	104	46	1186	408	234
Load X' window w' on demand	16	182	325	29	243
	104	56	717	122	199
Load X' window, $W'/2$	16	201	818	185	405
	104	57	1197	246	261
Load X' window, $W'/4$	16	190	450	109	293
	104	74	972	203	268
Colum Major	16	233	599	31	373
	104	131	891	190	342

could perform the computation on a machine with less memory.

Tables 4.8, 4.9, and 4.10 list the time and memory requirements and the score, using the large crypto parameters (for medium and small, see the Appendix). The first important observation is the accuracy of the estimation algorithm. We expect the memory measurements

Table 4.3: Measurements for **Time** in seconds, Memory (**Mem**) in MB, for all Schedules on Conv 2d (2) with small parameters on the PC wiht 16 and the server with 104 Threads. Additionally, the table shows the memory Estimate in MB and Score.

Schedule	Threads	Time	Memory (MB)	Estimate	Score
Lock-free Preload	16	583	15327	12346	2990
	104	161	15570	12412	1583
Lock-free on demand	16	1708	529	30	951
	104	524	892	195	683
Preload everything	16	581	15165	12335	2968
	104	187	15446	12335	1700
Preload X'	16	727	5930	5417	2076
	104	205	6187	5450	1127
Preload W', x' on demand	16	1341	9729	6938	3613
	104	449	10054	7004	2126
Load X' window W'	16	609	10180	7143	2490
	104	194	10427	7143	1423
Load X' window w' on demand	16	765	934	225	845
	104	210	1176	258	497
Load X' window, $W'/2$	16	816	10281	3681	2896
	104	239	11778	3681	1679
Load X' window, $W'/4$	16	796	3257	1955	1610
	104	223	3770	1988	917
Colum Major	16	853	1099	55	968
	104	492	1434	172	840

Table 4.4: Measurements for **Time** in seconds, Memory (**Mem**) in MB, for all Schedules on Conv 2d (3) with small parameters on the PC wiht 16 and the server with 104 Threads. Additionally, the table shows the memory Estimate in MB and Score.

Schedule	Threads	Time	Memory (MB)	Estimate	Score
Lock-free Preload	16	117	18700	15591	1477
	104	31	20728	15657	800
Lock-free on demand	16	329	254	30	289
	104	101	614	195	249
Preload everything	16	132	19362	15581	1601
	104	45	20710	15581	970
Preload X'	16	141	1980	1739	528
	104	39	2221	1772	292
Preload W', x' on demand	16	278	18688	13862	2281
	104	95	18975	13928	1346
Load X' window W'	16	134	19276	14283	1605
	104	48	19769	14283	972
Load X' window w' on demand	16	146	1085	441	399
	104	43	1330	474	239
Load X' window, $W'/2$	16	167	19254	7359	1792
	104	56	20839	7359	1081
Load X' window, $W'/4$	16	152	5781	3902	938
	104	45	6374	3935	539
Colum Major	16	160	416	55	258
	104	94	787	172	272

Table 4.5: Measurements for **Time** in seconds, Memory (**Mem**) in MB, for all Schedules on Conv 2d (1) with medium parameters on the PC wiht 16 and the server with 104 Threads. Additionally, the table shows the memory Estimate in MB and Score.

Schedule	Threads	Time	Memory (MB)	Estimate	Score
Lock-free Preload	16	859	8448	7928	2695
	104	160	9142	8126	1208
Lock-free on demand	16	1120	409	90	676
	104	233	1428	586	576
Preload everything	16	873	8422	7897	2712
	104	171	9286	8079	1258
Preload X'	16	1014	7304	6942	2721
	104	215	8182	7223	1327
Preload W', x' on demand	16	973	1472	1014	1197
	104	188	2375	1394	669
Load X' window W'	16	891	1600	1041	1194
	104	174	2634	1223	677
Load X' window w' on demand	16	1023	519	86	729
	104	188	1600	367	548
Load X' window, $W'/2$	16	988	1651	554	1277
	104	194	2439	737	688
Load X' window, $W'/4$	16	993	797	327	889
	104	185	1732	608	567
Colum Major	16	1177	751	92	940
	104	242	1726	570	646

Table 4.6: Measurements for **Time** in seconds, Memory (**Mem**) in MB, for all Schedules on Conv 2d (2) with medium parameters on the PC wiht 16 and the server with 104 Threads. Additionally, the table shows the memory Estimate in MB and Score.

Schedule	Threads	Time	Memory (MB)	Estimate	Score
Lock-free Preload	16	17154	18531	37011	17829
	104	654	40776	37209	5162
Lock-free on demand	16	4636	631	90	1710
	104	944	1668	586	1255
Preload everything	16	16589	18764	36979	17643
	104	769	40970	36979	5612
Preload X'	16	3946	16852	16239	8155
	104	881	17736	16339	3952
Preload W', x' on demand	16	12054	19356	20798	15275
	104	955	24631	20996	4850
Load X' window W'	16	10675	19036	21413	14255
	104	803	25897	21413	4561
Load X' window w' on demand	16	3885	1939	673	2745
	104	958	2864	773	1656
Load X' window, $W'/2$	16	11570	18950	11034	14807
	104	1069	29330	11034	5598
Load X' window, $W'/4$	16	3989	8043	5861	5664
	104	1438	10983	5960	3974
Colum Major	16	3933	1395	164	2342
	104	1759	2758	516	2203

Table 4.7: Measurements for **Time** in seconds, Memory (**Mem**) in MB, for all Schedules on Conv 2d (3) with medium parameters on the PC wiht 16 and the server with 104 Threads. Additionally, the table shows the memory Estimate in MB and Score.* values are unavailable because the execution ran out of memory.

Schedule	Threads	Time	Memory (MB)	Estimate	Score
Lock-free Preload	16	*	18640	46741	
	104	212	52416	46939	3331
Lock-free on demand	16	817	387	90	562
	104	329	1396	586	677
Preload everything	16	*	18971	46709	
	104	325	52493	46709	4131
Preload X'	16	748	5552	5212	2038
	104	283	6296	5311	1336
Preload W', x' on demand	16	5237	18853	41556	9936
	104	547	47242	41754	5083
Load X' window W'	16	*	18821	42819	
	104	347	49830	42819	4159
Load X' window w' on demand	16	753	2950	1322	1491
	104	263	3475	1421	956
Load X' window, $W'/2$	16	*	19015	22062	
	104	441	52804	22062	4826
Load X' window, $W'/4$	16	752	15296	11699	3392
	104	354	17400	11798	2482
Colum Major	16	744	630	164	684
	104	323	1715	516	745

Table 4.8: **Time** in s, **Memory** in MB requirements, for all Schedules on Conv 2d (1) with large parameters on the PC with 16 and the server with 104 Threads. Additionally, shown are memory **Estimate** in MB and **Score**.

Schedule	Threads	Time	Memory	Estimate	Score
Lock-free Preload	16	8952	18432	35215	12845
	104	1600	38839	36095	7883
Lock-free on demand	16	6830	1179	400	2838
	104	2091	4662	2601	3123
Preload everything	16	6805	18408	35075	11192
	104	973	38365	35885	6109
Preload X'	16	5812	18680	30833	10420
	104	3379	33826	32084	10691
Preload W', x' on demand	16	5911	5459	4502	5681
	104	3925	7965	6193	5591
Load X' window W'	16	4458	5857	4622	5110
	104	1183	8036	5432	3083
Load X' window w' on demand	16	5278	1486	380	2801
	104	933	4074	1631	1950
Load X' window, $W'/2$	16	4955	6104	2461	5499
	104	1144	7972	3271	3021
Load X' window, $W'/4$	16	5498	3061	1451	4102
	104	899	5504	2701	2225
Colum Major	16	5442	1736	410	3074
	104	1452	4363	2531	2517

Table 4.9: textbfTime in s, **Memory** in MB requirements, for all Schedules on Conv 2d (2) with large parameters on the PC with 16 and the server with 104 Threads. Additionally, shown are memory **Estimate** in MB and **Score**.* indicate out of memory.

Schedule	Threads	Time	Memory (MB)	Estimate	Score
Lock-free Preload	16	*	18875	164390	
	104	3257	173712	165270	23787
Lock-free on demand	16	24060	1531	400	6069
	104	4599	6016	2601	5260
Preload everything	16	*	18819	164250	
	104	3364	173326	164250	24146
Preload X'	16	*	18949	72131	
	104	4138	76506	72571	17792
Preload W', x' on demand	16	*	18816	92379	
	104	5416	101272	93260	23419
Load X' window W'	16	*	18754	95110	
	104	3662	106345	95110	19735
Load X' window w' on demand	16	20246	6888	2991	11809
	104	3914	10203	3431	6319
Load X' window, $W'/2$	16	*	18839	49011	
	104	4003	125113	49011	22379
Load X' window, $W'/4$	16	*	18871	26031	
	104	3843	33731	26471	11385
Colum Major	16	22251	2809	730	7906
	104	3724	6516	2291	4926

Table 4.10: **Time** in s, **Memory** in MB requirements, for all Schedules on Conv 2d (3) with large parameters on the PC with 16 and the server with 104 Threads. Additionally, shown are memory **Estimate** in MB and **Score**.* indicate out of memory.

Schedule	Threads	Time	Memory (MB)	Estimate	Score
Lock-free Preload	16	*	18788	207608	
	104	7074	220310	208489	39479
Lock-free on demand	16	5054	1133	400	2393
	104	1469	4823	2601	2662
Preload everything	16	*	18838	207468	
	104	855	219812	207468	13709
Preload X'	16	3955	18910	23150	8648
	104	815	26310	23590	4631
Preload W', x' on demand	16	*	18859	184578	
	104	4935	196842	185459	31168
Load X' window W'	16	*	18775	190191	
	104	831	207599	190191	13134
Load X' window w' on demand	16	3876	12218	5872	6881
	104	913	14437	6313	3631
Load X' window, $W'/2$	16	*	18860	97992	
	104	1024	206385	97992	14539
Load X' window, $W'/4$	16	*	18871	51962	
	104	955	63491	52402	7788
Colum Major	16	3834	1714	730	2564
	104	825	4609	2291	1950

to be larger than the estimate since there is runtime overhead, like the schedule itself, key material, and other data structures that the estimation does not take into account. However, in some cases, the estimate is off by a factor of 4-5. This is especially true for smaller values. An explanation for the discrepancy in estimate and measurements most likely lies in how we process cache instructions that drop data from memory. To ensure that we do not delete data that other threads still need, we only execute the delete instructions once all threads have passed the point for which the instructions are scheduled. During execution, we have little control over how fast threads advance. It is certainly possible for some threads to fall far behind, waiting for locks or input/output operations, thereby preventing the deletion of objects from memory. We have no way of predicting how the threads will interact at runtime and, therefore, need to make simplifying assumptions that can cause the differences in estimated and measured values. Overall, the estimate can still provide us with a useful tool to understand the schedule's memory requirements without running it.

The most important metric is time. The schedule that executes the fastest is typically the schedule that uses the available resources the most efficiently. Fig 4.6 shows the time and memory requirement for large parameters and all schedules. Note that the OS terminated schedules that do not display a time for 16 Threads (the PC) for running out of memory. For Conv 2d (2) and Conv 2d (3) we can see multiple schedules that reach the critical limit of 18000 GB memory, after which the OS's swapping system kicks in. On the medium parameters and Conv 2d (2) (complete Figures and Table in the Appendix), we observe that *Load* X' *window* W' schedule reaches the swapping limit and takes 10675 seconds. The *Load*

X' window w' on demand schedule does not reach that limit needing $\tilde{2}$GB. However, despite needing to encode data more often, it is faster at 3885s. This strongly suggests that our algorithm is more efficient than relying on the OS's swapping mechanism.

Table 4.11 and Fig. 4.7 compare the fastest schedule for each layer and set of parameters. We are most interested in the increase in runtime and the reduction in memory when running on the 16-thread PC as compared to running on the 104-thread server. For the small parameters, the fastest schedule is either the *Lock-free Preload* or *Preload everything* schedule. Since these schedules have very similar memory requirements, there is no significant reduction in memory. The time, however, increases by a factor of 3.3-3.8. We start to see a much bigger difference when moving to the medium parameters. For the Conv 2d (1) layer, the time increases by a factor of 5.4 while the memory usage stays almost the same between PC and server. For this layer, both systems can still use the *Lock-free Preload* schedule, which explains the negligible reduction in memory. The time increase for the next two layers is 5.9 and 3.5, respectively; however, the memory reduction is significant and a factor of 21 and 83.3. While the server still uses the *Lock-free Preload* schedule the PC is forced to use window caching and column-wise window caching to fit the objects into memory. The picture repeats for the large parameters. Except that now the server uses a more memory-efficient schedule for Conv 2d (3), which leads to only 15.3 times memory reduction and an increase in runtime by 4.7. An interesting observation: on the small parameters, the PC seems to have a higher per-thread performance as the time increase is only around 3.5 for all layers despite the number of threads on the server being 6.5 more. As the parameters get larger,

Table 4.11: Fastest Schedule for each layer and parameter size (Param.) on the server, 104 Threads (T), and PC, 16 Threads. As well as the increase (Inc.) in time and reduction (Red.) of memory.

Param.	Layer	T	Time	Inc.	Memory	Red.	Schedule
small	Conv 2d (1)	16	135	3.3	3084	1.1	Lock-free Preload
		104	41		3319		Lock-free Preload
	Conv 2d (2)	16	580	3.6	15164	1.0	Preload everything
		104	160		15570		Lock-free Preload
	Conv 2d (3)	16	116	3.8	18699	1.1	Lock-free Preload
		104	30		20727		Lock-free Preload
medium	Conv 2d (1)	16	859	5.4	8447	1.1	Lock-free Preload
		104	159		9141		Lock-free Preload
	Conv 2d (2)	16	3885	5.9	1939	21.0	Load X' window w' on demand
		104	653		40775		Lock-free Preload
	Conv 2d (3)	16	743	3.5	629	83.3	Colum Major
		104	211		52415		Lock-free Preload
large	Conv 2d (1)	16	4457	5.0	5857	0.9	Load X' window W'
		104	899		5504		Load X' window, $W'/4$
	Conv 2d (2)	16	20246	6.2	6887	25.2	Load X' window w' on demand
		104	3257		173712		Lock-free Preload
	Conv 2d (3)	16	3834	4.7	1714	15.3	Colum Major
		104	815		26309		Preload X'

the time increase seems to approach 6.5 as expected.

Additionally, we compare the time and memory of the different schedules run on the large crypto parameters executed on the server. For the Conv 2d (1) layer the fastest schedule is *Load X' window, $W'/4$*. It is 74s, 8%, faster than the *Preload everything schedule*. The *Preload everything schedule*, in turn, is much faster, 627s (64%), than the *Lock-free Preload* schedule. However, both preload schedules require 38 GB of memory, compared to the 5.4 GB of the *Load X' window, $W'/4$* schedule. For the second layer, Conv 2d (2), the *Lock-free Preload* schedule is the fastest at 3257s. The *Preload every* is marginally slower at 3364s. Both schedules require 170 GB of memory. Schedules that require significantly less memory *Load X' window, $W'/4$* (33 GB) and *Column Major* (6.3 GB) are only slightly slower at 3662s and 3843s. For the Conv 2d (3) layer the *Lock-free Preload* schedule is the slowest and consumes the most memory at 7074s and 215 GB. The comparable *Preload*

everything schedule requires approximately the same amount of memory, but only 12.5% of the time, 855s. Interestingly, schedules that cache very little *Preload* X' and *Column Major* are faster than the *Preload everything* at 815s and 825s. Both of these schedules also require significantly less memory, at 25.7 GB and 4.5 GB. This is a reduction factor of 47.8 between *Lock-free Preload* and *Column Major*.

Interestingly, schedules with minimal caching of w's are often faster than schedules that substantially cache these values. A potential explanation could be the cache locality inside the CPU. Values that are not cached by our method and are loaded on demand could be accessed faster because they are placed inside the CPU cache. Alternatively, locking that is required for processing the load instructions could introduce additional slowdowns that are not present when values are loaded on demand. Another interesting observation is the poor performance of the *Lock-free Preload* schedule in the Conv 2d (3) layer. It is eight times slower than *Preload everything* schedule. Both schedules load all the required data at the start of the computation and do not need to load any values during. Where they differ is the the points at which they write the results to disk. If we assume that all threads advance in lockstep, in the *Lock-free Preload* schedule, all threads want to write to disk at once. In *Preload everything* schedule, the write operations are more spaced out. It could be that the large number of simultaneous writes slows the schedule down significantly.

4.6 Summary and Discussion

In this chapter, we present ways of reordering the computation to tailor the memory requirements to the hardware available while executing as fast as possible. We further present a technique to estimate the required memory of convolutions over batch-packed, encrypted data. We show that our proposed caching mechanism is faster than relying on the OS's swapping mechanism. The method proposed in this chapter is especially suited for ML workloads with thousands of instances that can run longer, i.e., overnight or over the weekend, and don't need a fast turnaround. Since our method can reduce the memory requirements for inference, it opens up the potential to save on hardware costs.

The memory optimization presented above only works for convolutional layers and relies on loading data from disk, which can introduce additional slowdown. We will now discuss the challenges of applying this approach to other layers and present and possible extension of our coaching approach that reduces memory uses while eliminating reading data from disk.

4.6.1 Application to other layers

Our proposed method heavily relies on the repeating structure of convolutional layers. In fully connected layers, we have no such structures. When we use batch-packing the input matrix becomes a vector. On plain data, the input matrix is $n_b \times n_f$ where n_b is the batch size and n_f the number of features. Encrypting the data turns into a vector X of length n_f ciphertexts. We need to multiply the vector with the weight matrix W, which is $n_f \times n_n$, with n_n as the number of neurons. The result is a vector Y of length n_n. Every value of x

is used in every iteration, and every value from W is used exactly once, as we can see from the following equation:

$$Y_i = \sum_{j=1}^{n_f} X_j W_{j,i} \tag{4.1}$$

This does not allow us to reuse any values from W. However, we design a schedule that could be useful in certain configurations. The schedule S is ordered so that all steps containing a specific X_i are all grouped together. This means we compute all products that contain one input feature before moving to the next one. This order can be useful when we have significantly more inputs than we have neurons in the layer because the downside of this schedule is that we need to keep all the outputs Y_i in memory for the entire computation and can not write them disk early. Since vanilla RNNs share the same computational structure as fully connected layers, this schedule could be helpful here as well.

4.6.2 Optimizing schedules for peak memory usage

The technique presented above relies on swapping data in and out of memory, which can significantly reduce maximum memory requirements. However, this requires we read data from disk and or encode data frequently. Essentially, we sacrifice some latency for a smaller memory footprint. Ideally, we want to reduce the memory requirement without sacrificing latency. We now outline an approach that could help us achieve this. We now assume that the data is in memory as long as we need it and only unloaded once we are completely done with it. We again use the schedule introduced in Chapter 4. We assume that at the

beginning of the computation, all X' are in memory, and all W' and Y' are not in memory. Additionally, we impose the following constraints:

1. any x', w' or y' can be unloaded once no further step in the schedule relies on it
2. any w' and y' is loaded at the first step that relies on it

Based on these constraints and $s_{x'}$, $s_{w'}$ and $s_{y'}$ as the size of x', w', and y' we can estimate the memory of the schedule. We show the algorithm in Algorithm 6. It is an extension of the maximum overlap algorithm to multiple inputs. We set the first occurrence of every x' to 0, the start, and the last occurrence of every y' to the last step of the schedule. This models or constraints from earlier that the computation starts with every x' in memory and that every y' is in memory at the end of the computation.

Algorithm 6 Estimating the maximum memory requirements of an completely in-memory schedule

Inputs: Schedule S; x_f, x_l, w_f, w_l, y_f, y_l as first and last occurrence of every x', w', and y'; x_f is all 0 and y_l is all $|S|$; f a function that returns the size of an object
Outputs: Cost of a Schedule

```
1: c := [0]
2: start := combine and sort x_f, w_f, y_f
3: end := combine and sort x_l, w_l, y_l
4: while True do
5:     if end[i] ≤ start[i] then
6:         if end[i] ≥ |S| then
7:             return max(c)
8:         end if
9:         c[end[i]] := c[end[i]] − f(end[i])
10:    else
11:        c[start[i]] := c[start[i]] + f(end[i])
12:    end if
13:    i := i + 1
14: end while
```

With this estimation function C, define our optimization problem. We want to find a permutation S' of the schedule S so that $C(S')$ is minimal. A permutation of a schedule is the reordering of its steps. We call the default, or base schedule, output priority because it prioritizes computing individual output values. The ordering rules for this schedule are: For all tuples/steps in the t_i, t_j in the schedule the following conditions must hold, (1) $t_i^y \leq t_j^y$, (2) $t_i^x \leq t_j^x | t_i^y = t_j^y$ and, (3) $t_i^w < t_j^w | t_i^y = t_j^y \wedge t_i^x = t_j^x$. In other words, we order it by y, with x as a first tiebreaker and w as a second tiebreaker. Further, we also define two additional schedules, the left-hand side priority schedule, and the right-hand side priority schedule, The ordering rules for the left-hand side priority schedule are (1) $t_i^x \leq t_j^x$, (2) $t_i^w < t_j^w | t_i^x = t_j^x$ and, (3) $t_i^y \leq t_j^y | t_i^x = t_j^x \wedge t_i^w = t_j^w$; and for the right-hand side priority schedule: (1) $t_i^w \leq t_j^w$, (2) $t_i^x < t_j^x | t_i^w = t_j^w$ and, (3) $t_i^y \leq t_j^y | t_i^x = t_j^x \wedge t_i^w = t_j^w$.

We apply the estimation algorithm to the same three convolutional layers, Conv 2D (1), Conv 2D (2), and Conv 2D (3), which we used earlier in this section, with the large crypto parameters parameters. Fig. 4.8 shows the estimated memory usage at each schedule step. We can see that the output priority schedule is in fact the one with the lowest memory requirement at 286 GB. The left-hand side priority schedule only requires a little more memory with 287 GB, and the right-hand side priority uses 311 GB. The output priority schedule requires about 7.5x more memory than the most memory-hungry schedule discussed in Section 4.5. This is because the execution in 4.5 writes each output to disk and clears it from memory once it is done. However, this requires loading data from disk when computing the next layer. This loading and unloading is what we want to avoid here.

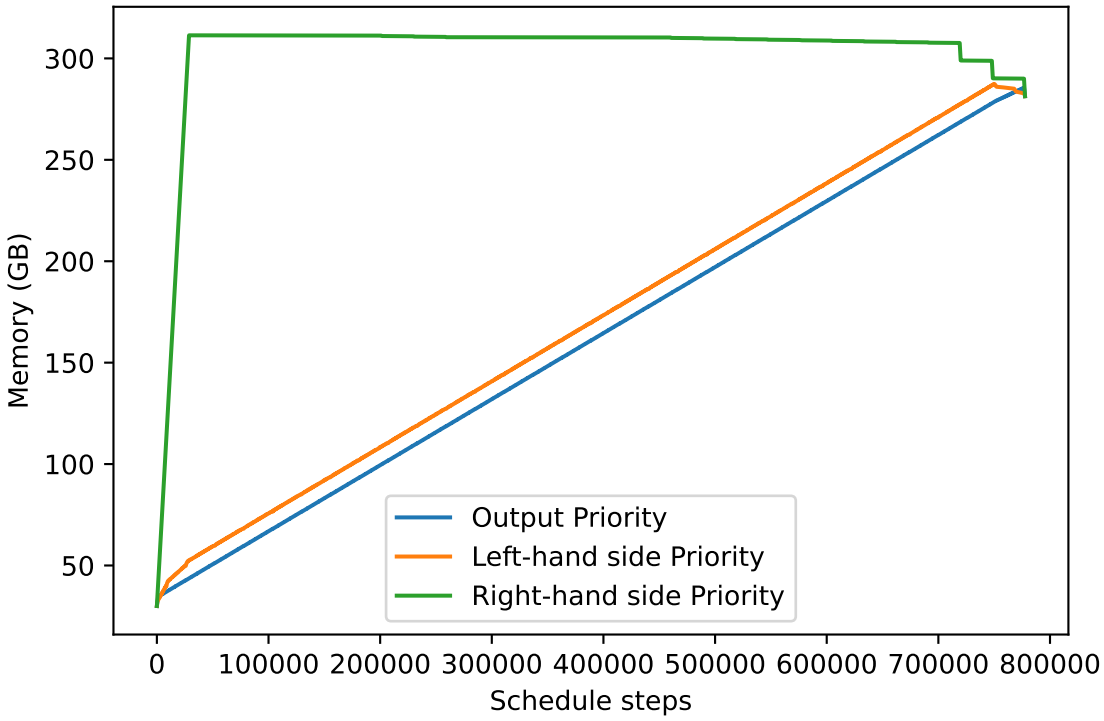


Figure 4.8: Comparison of the memory requirements at each step for three schedule permutations of the Conv 2D (1) layer.

For the second layer, Conv 2D (2), shown in Fig. 4.9, the right-hand side priority schedule requires the least memory at 176 GB compared to the 203 GB of the output priority schedule, 206 of the left-hand side priority schedule. This is slightly more than the worst case of 169 GB in Section 4.5.

We see the most significant difference with the Conv 2D (3) layer, Fig. 4.10. The right-hand side priority schedule requires only 33 GB, compared to the 203 GB of the output priority and 201 GB of the left-hand side priority schedule. We measure 215 GB for the worst case in Section 4.5. Our worst-case estimate is still better than the measured worst-case.

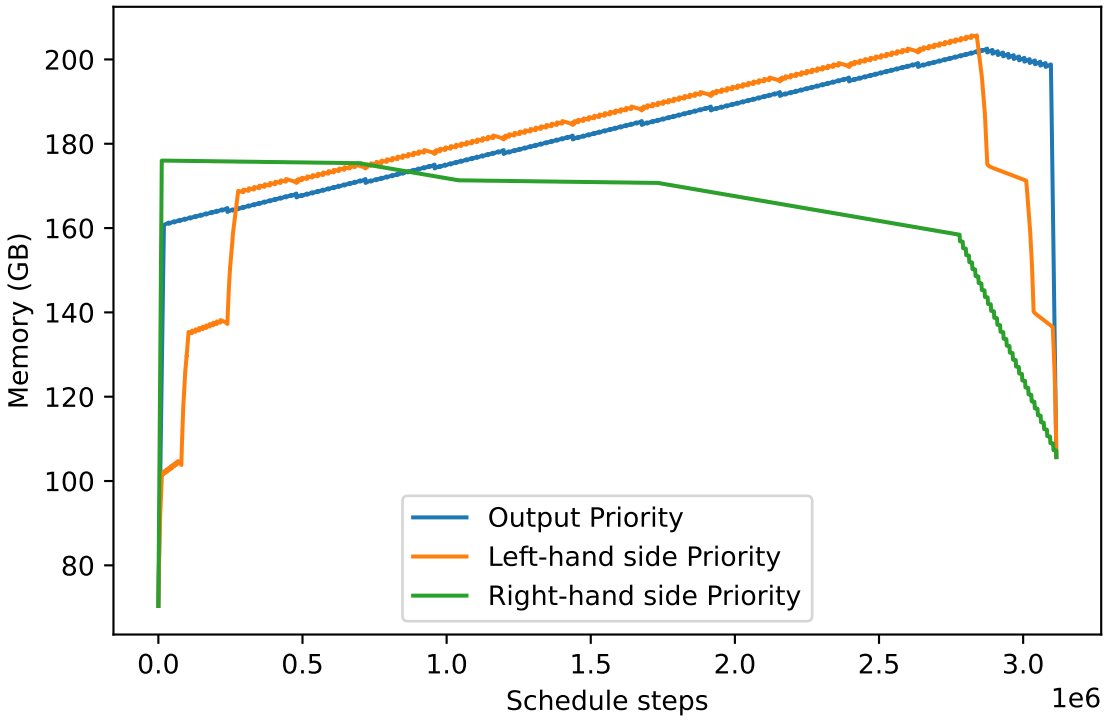


Figure 4.9: Comparison of the memory requirements at each step for three schedule permutations of the Conv 2D (2) layer.

However, we can not simply choose the "best" schedule for each layer and run the network that way. We are bound by the maximum memory of the most expensive layer. There is no benefit to choosing the right-hand side priority for the second and third layers since the first layer requires more memory already. We can switch the first layer to one of the schedules presented earlier in this chapter; however, this creates the issue that these schedules do not have their results in memory, while the schedules in this section rely on that. Finding the optimal combinations of layers most likely requires some trial and error. However, we provide the tools and heuristics here to narrow down the possible candidates.

We are also not limited to the three schedules, output, right-hand side, and left-hand

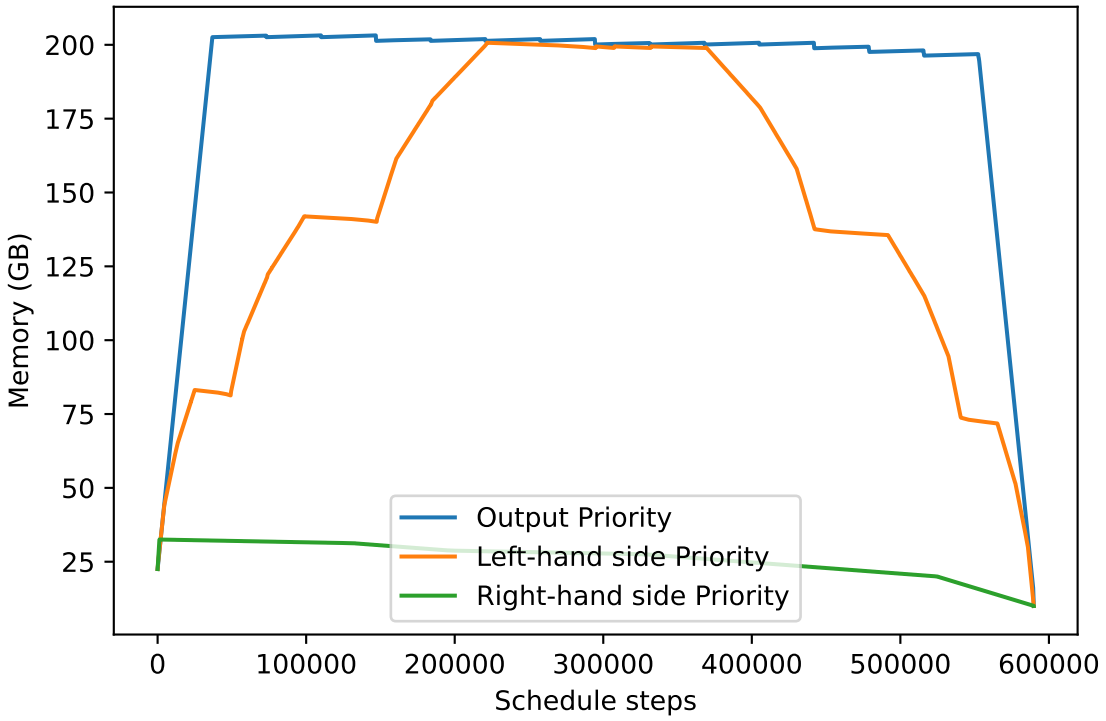


Figure 4.10: Comparison of the memory requirements at each step for three schedule permutations of the Conv 2D (1) layer.

side priority, presented in this section. More optimal permutations with a lower memory requirement might exist. However, finding better permutations is not trivial. In initial experiments, we had some limited success using a genetic algorithm. However, we are only able to find a better schedule for a small schedule with less than 400 steps, and the result is only better by the size of one weight. For large schedules, the genetic algorithm did not find any solution with millions of iterations. This problem requires further investigation in the future.

CHAPTER 5

CLASSIFICATION OF ENCRYPTED WORD EMBEDDINGS USING RECURRENT NEURAL NETWORKS

In this chapter, we present and evaluate an interactive approach for RNNs over encrypted data. RNNs excel at sequence processing, by taking the sequential relation of the data into account. However, when using homomorphic encryption the sequential processing becomes an issue. In a client-server setting where the client owns the data and the server owns the model, we can use interactive phases to reset the noise level. The content of this chapter was previously published as: *Classification of Encrypted Word Embeddings using Recurrent Neural Networks*

Robert Podschwadt and Daniel Takabi in *PrivateNLP 2020: Workshop on Privacy in Natural Language Processing, 2020* [2]

5.1 Introduction

Artificial neural networks have been very successful and popular over the last few years in a variety of domains. CNNs have shown better than human performance in image classification tasks [133, 134] and have also been applied to language processing tasks [135]. RNNs, another type of neural networks, are specifically designed to work with sequences. Unlike other types of networks, RNNs take the output of the previous sequence step into consideration. There are different types of RNN architectures such as Long Short Term Memory (LSTM), Gated Recurrent Unit (GRU) and a simple fully connected variant or Elman Network [86]. Here, we work with Elman Networks and unless specified otherwise will use the term RNN instead

of Elman Network. Recurrent architectures are very popular in natural language processing (NLP) due to the sequential nature of language. There are many different sub-fields in NLP. Here, we investigate the task of sentiment classification.

Many companies have built a business around offering MLaaS. In MLaaS the model is hosted in the cloud. The service provider has the infrastructure and know-how to build the models. The client owns the data and sends it to the provider (also called server) for processing.

A concern for the client of MLaaS is the privacy of the data. To process the data the server needs access to the data. This is often unwanted or unacceptable depending on the sensitivity of the data. There are three main techniques for preserving the privacy of the data while still allowing for ML algorithms to work: 1) Secure Multiparty Computation (SMC), 2) Differential Privacy (DP) and 3) Homomorphic Encryption (HE).

In previous work, a variety of different machine learning algorithms have been adapted for privacy-preserving processing such as linear regression [136], linear classifiers [137, 138], decision trees [139, 138] or neural networks [27, 136, 140]. Solutions based on SMC [136, 140] come with a huge communication overhead.

We propose an approach that is based on homomorphic encryption and recurrent neural networks. It does not require interactive communication between client and server like SMC approaches but in the case of longer sequences, we use interactive communication to control the noise introduced by HE. Very little prior work deals with recurrent neural networks. Much of the work is done on CNNs in the image domain [27, 125, 59] and more. [68]

perform encrypted speech recognition which is an NLP task but the model used is also a CNN. Badwai et al. [62] research privacy-preserving text classification which is the task that we also use as well but the authors do not use an RNN. To the best of our knowledge, there is only one prior paper working with a recurrent architecture. Qian and Lei propose a system [58] that is capable of implementing LSTM networks based on TFHE [24]. Their LSTM model suffers from a small drop in accuracy though when running on encrypted data. Our solution is able to maintain the same accuracy as the plain text model. We present a solution that can process RNNs with arbitrary length input sequences in a privacy-preserving manner and introduce a way of using word embeddings with encrypted data. To ensure the privacy of the data we rely on the CKKS [49] crypto scheme. We evaluate our system on a text classification task. The basic idea of our proposed approach is running RNNs on encrypted data by taking advantage of HE schemes. The server hosts the trained model, and the client transmits the encrypted data for processing and receives an encrypted result. The training of the model is done on plaintext. In this work, we make the following main contributions:

- We propose an approach that combines RNNs, specifically Elman Networks, and homomorphic encryption to perform inference over encrypted data in natural language processing tasks.
- We present an innovative approach to work with word embeddings for encrypted data.
- We perform thorough benchmarking of our system both with respect to run time performance and communication cost. Our results demonstrate that we are able to run

RNNs over encrypted data without sacrificing accuracy and with reasonable performance and communication cost.

5.1.1 Threat Model and Problem Statement

In this chapter, we apply privacy-preserving machine learning techniques based on HE to RNNs. We focus on a client server setting such as MLaaS in which the client has full control over the data and the server has full control over the model. We assume that the model has been trained on plaintext data and the server offers inference as a service to the clients. The clients want to use the inference service and wish to keep their data private while the server wishes to keep its model private .

Threat Model: We assume that all parties are honest but curios. They will not deviate from the protocol but will attempt to learn any information possible in the process. The server does not share information about the architecture of the model with the client. The client encrypts the data and sends it to the server for processing. If it is possible, the server will process the data and send back the final result in encrypted format. In some cases data will be sent back to the client where it is decrypted, encrypted again to remove the built-up noise and sent back to the server to continue processing. In addition to the privacy of the data we have the goal to achieve accurate predictions. This means the predictions made on encrypted data should be as close as possible to predictions made on plaintext data.

5.1.2 NLP with Neural Networks

Recurrent neural networks are widely used for addressing challenges in Natural Language Processing. Recurrent neural network reached state-of-the-art performance for different tasks

such as: Speech Recognition, [141] and [142], Generating Image Descriptions, [143] and [144], Machine Translation, [145] and [146], Language Modeling, [147] and [148]. The implementation of an NLP pipeline using RNNs can be broken done into four major parts: 1) Designing the network, 2) Encoding the data, 3) Training the model, and 4) Inference of new instances.

In the next section we will look at the individual steps in detail and describe the changes that are necessary for computation in a privacy preserving setting.

5.2 Related Work

Badwai et al. [149] presented PrivFT a system for privacy-preserving text classification built on Facebook's *fasttext* [150] (Joulin et al. [150]). The main difference to our work is that we use a recurrent architecture. In PrivFT, the embedding operation is also not outsourced to the client. The client needs to one-hot encode each word, encrypt it, and send it to the server where the embedding operation is performed as a matrix multiplication. The message size is similar. The inference time for a single instance on the IMDb is higher in our scenario but using larger batch sizes allows us to get a lower per instance time. In contrast to our work, PrivFT features schemes for training on encrypted data and a *CKKS* implementation with GPU acceleration. Lou and Jiang created SHE [151] a privacy-preserving neural network framework based on TFHE. It offers support for LSTM cells. The authors replace the computationally expensive and high noise-introducing matrix operations normally required by LSTMs with much cheaper shift operations. Zhang et al. [152] perform a different NLP task namely encrypted speech recognition based on a CNN. The last step of the network

that matches the output to actual text is performed on the client side.

5.3 The Proposed Privacy-preserving Classification for Recurrent Neural Networks

Looking at the components of the RNN pipeline described in Section 5.1.2 we determine what changes need to be made to adhere to the constraints of homomorphic encryption.

Network Design. As long as we only use fully connected and recurrent layers the only consideration we need to make are the activation functions that are being used. All other operations inside an RNN can be performed over encrypted data using HE schemes. However, it is not possible to implement common activation functions within current HE schemes. We aim to find the best low degree polynomial approximation to replace the activation functions within the RNN.

Data Encoding. Here, we use word embeddings as an encoding scheme for textual data. We describe our approach to handling embeddings in more detail in Section 5.3.1.

Model Training. We assume that the training of the model is performed by the server on plain training data.

Inference. This is the part of the pipeline in our system that is run on encrypted data. At no point during this process is the data decrypted on the server thus ensuring its privacy is protected. During processing by the model, the encrypted data accumulates noise. We describe a way of circumventing the problem of the noise crossing the threshold after which correct decryption is no longer possible in Section 5.3.2. Once the data has been processed by the entire network, the result of the classification is sent back to the client. The result of

the classification is still encrypted and needs to be decrypted by the client.

A variety of activation functions have been proposed as replacements for common activation functions used in NNs. Dowlin et al. [153] use polynomials of degree 2 to substitute the Sigmoid function in CNNs, and Shortell and Shokoufandeh [154] use a polynomial of degree 3 to approximate the natural logarithm function. Hesamifard et. al [83] use Chebyshev polynomials to approximate activation functions such as ReLU, Sigmoid and Tanh. We will be using the approach of [83] to approximate Tanh which is the most popular activation function in RNNs. The Softmax function can not be performed over encrypted data but since it is typically used as the very last function of neural network, we move it to the client side. The server computes the neural network all the way to the inputs of the Softmax function. The Softmax function is performed by the client after decryption to obtain the classification results.

5.3.1 Encrypted word embeddings

Word embeddings are a way to turn words into real valued vectors. The embedding layer basically is a lookup table that maps any word in a dictionary to a real valued vector. The lookup of an embedding for a given word cannot be performed efficiently in HE schemes. We address this problem by moving the embedding layer out of the RNN and to the client where it can be performed in plaintext. After performing the embedding lookup, the client encrypts the embeddings and sends the result to the server. To enhance the privacy of the model, the model owner can use one of the many pretrained embeddings such as GloVe [155], Elmo [156], Bert [157] or XLNet [158] and share those with the client .

5.3.2 Noise growth in HE

In an RNN architecture, a sequence is processed by feeding its entries into a fully connected layer which also takes the output of that layer produced for the previous sequence entry. The current output and the previous output are combined into the new output. Due to the noise build-up in HE we need to keep track of the number of operations performed on ciphertexts. To process a sequence of length n with an RNN layer the resulting ciphertext needs to pass the layer n times. That means n dot products and activation functions are applied. It is not always possible to process all of the sequence entries due to the noise that is accumulated. Our approach is to send the encrypted data back to the client where it is decrypted and re-encrypted thereby removing the built up noise.

5.3.3 Implementation

We use CKKS to protect the privacy of the client data. The server trains a plaintext model and shares the embedding matrix with the client. The activation in the model needs to be compatible with HE. This is achieved by approximating Tanh using the method by Hesamifard et al. [83]. The client performs the embedding process and encrypts the result. The encrypted embeddings are sent to the server where it is processed. When the noise, built up during computation, reaches the limit it the data is sent back to client where it is decrypted, thereby removing all noise, rencrypted and sent back to the server. Once the model is completed processed the server sends the still encrypted resutl back to the client where it can be decrypted. We implement our proposed solution in C++11. We train the

model using *Keras* [159] and the homomorphic encryption primitives are provided by HElib [160]. On the plaintext, we tried different activation functions and found out that Tahn and Tanh approximations work best. Other activation functions such as x^2 or the linear function cause the model not to train properly. We find that best replacement for our purposes is: $-0.0016357430301874 8x^3 + 0.249476365628036x$.

5.4 Evaluation and Experimental Results

5.4.1 Data and Preprocessing

The IMDb [161] dataset contains 50,000 movie reviews labeled as either positive or negative, of which 25,000 are used as training and 25,000 as test data. The tokenization is performed by Keras. We train a model to perform sentiment classification, which is classifying a review as either positive or negative. Out of the 25,000 training instances, we use 2,000 as validation data for hyperparameter tuning. We use a vocabulary of the top 20,000 words. We pad or truncate the reviews to be 200 words long. Our model consists of an embedding layer that turns words in the reviews into real-valued vectors of dimension 128. The embedding matrix is randomly initialized and updated during the training process. The embedding layer is followed by an RNN layer with 128 units. We use the Tanh approximation from Section 5.3.3 as an activation function. The last layer is a fully connected layer with two units and Softmax activation. The training is performed on the plain data using *Keras* and yields 86.47% accuracy on the unseen test data. We achieve the same accuracy on the encrypted data.

5.4.2 Results on Encrypted Data

The experiments were performed on a Ubuntu 18.04 64bit machine with an AMD Ryzen 5 2600 @ 3.5GHz processor and 32GB of RAM. We extract the learned weights and run experiments with different batch sizes. In our experiments the noise growth exceeds the workable threshold after 27 timesteps. This means we need to add communication between the client and server seven times to refresh the noise in order to classify the IMDb sequences of length 200.

The amount of data that needs to be transmitted depends on the batch size. The encrypted embeddings are larger than the plaintext data by a factor of 1,280. See Table 5.1 for different batch sizes. The *Embeddings* column is the amount of data that is initially transferred from the client to server. *Noisy ciphertext* gives the size of the data the server sends to the client to be refreshed and *Refreshed ciphertext* is the reencrypted answer. These are the values for only one refresh operation. The *Batch* column is the total amount of data transferred between client and server during classification of one batch which requires seven refresh rounds.

The amount of data that needs to be transmitted initially makes up the largest portion of the transfer. To run our network seven noise removal communications are required. At a batch size of 256 the server sends 106MB to the client and the client responds with 70MB. One round of noise removal therefore requires 176 MB to be transferred. All seven rounds take 1,232MB. Which is less than 10% of the initial transfer. The increase in size of the ciphertexts is nearly linear. Smaller ciphertexts sizes carry more overhead per instance than

Table 5.1: Data transferred during encrypted classification

Batchsize	Embeddings	Noisy ciphertext	Refreshed ciphertext	Batch
1	125MB	0.939MB	0.623MB	135MB
4	287MB	2.2MB	1.5MB	312MB
32	1,843MB	14MB	9MB	2,004MB
64	3,548MB	27MB	18MB	3,863MB
128	7,065MB	54MB	35MB	7,869MB
256	14,336MB	106MB	70MB	15,568MB

larger ones.

Table 5.2 lists the execution time for different batch sizes. The times are given for encrypted, plain data and for the actual time it takes to processes the batch as well as the resulting time per instance. The noise removal is not performed by the client though. It is simulated on the server. The measurements also do not include the encryption and transfer of the embeddings. We can see that increasing the batch size leads to lower per instance classification time. The effect is lost when increasing the batch size from 128 to 256. On the plain data we still can see improvement after that point. To get an accurate comparison the plain text measurements are performed on the same implementation as the encrypted experiments. It looks like the growth in execution time for the encrypted values is exponential while the plain version appears to be logarithmic. Our implementation performs best on encrypted data with a batch size of 128 and worst with a batch size of one if we look at the time per sample. The overhead is smallest though for one instance per batch. Here the encrypted version is 40 times slower than the plain version. For our optimal batch size of 128 the encrypted version is 92 times slower. This is due to the different growth rates of execution time for the encrypted and plain data.

Table 5.2: Run times of inference on IMDb test set.

Batch Size	Encrypted (sample/batch)	Plain (sample/batch)
1	70.6s / 70.6s	1.83s / 1.8s
4	20.2s / 80.7s	0.496s / 1.99s
32	5.8s / 184.6s	0.072s / 2.30s
64	4.3s / 272.7s	0.055s / 3.52s
128	4.2s / 547.6s	0.046s / 5.89s
256	6.5s / 1658.7s	0.039s / 9.96s

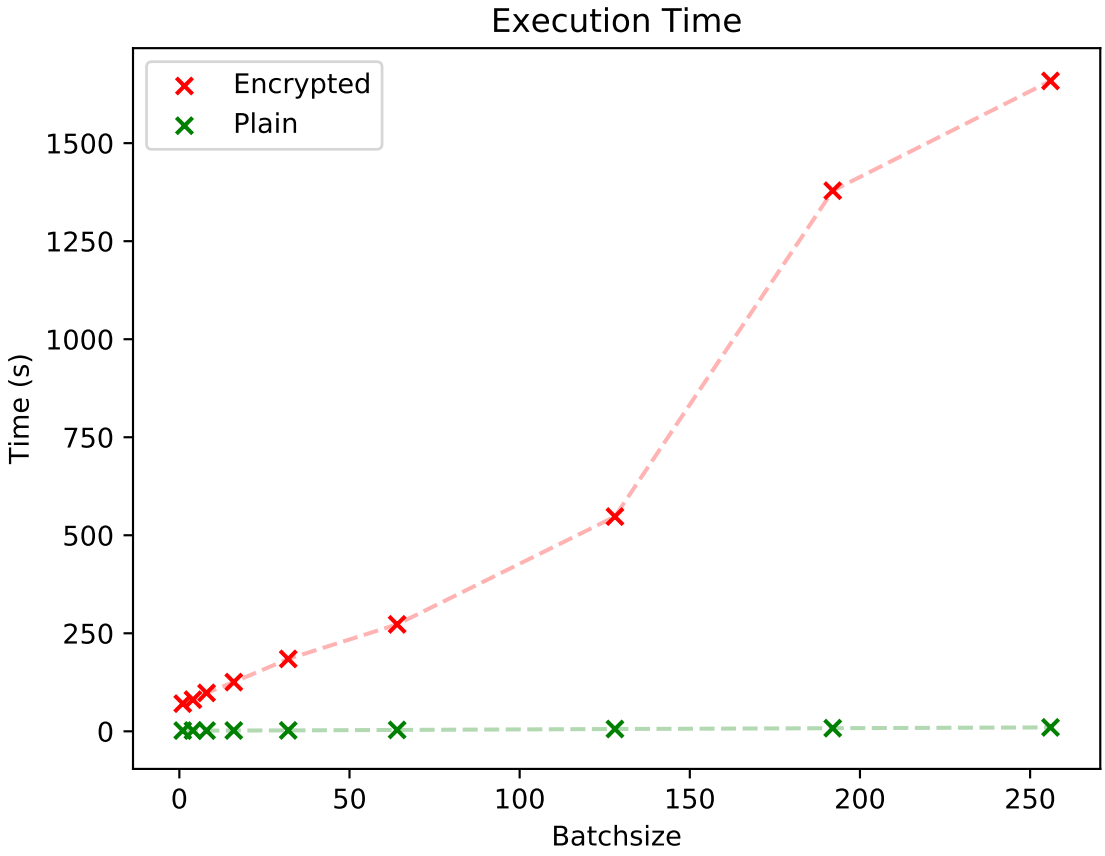


Figure 5.1: Inference time per batch on the IMDb data

5.5 Summary and Discussion

In this chapter, we present an approach that allows the use of recurrent neural networks on homomorphically encrypted data based on the CKKS scheme. We present a solution to perform NLP tasks over encrypted data using recurrent neural networks, in our case sentiment analysis on the IMDb dataset. We can achieve this with no loss in accuracy compared to the plaintext model. This is made possible by introducing communication between client and server to refresh the noise. We trade network traffic for the ability to efficiently use word embeddings.

The approach for privacy-preserving RNNs presented in this chapter suffers from several drawbacks. (1) The approach relies on communication between the server and client, requiring the client to stay online during computation and introducing network latency as a bottleneck. (2) The crypto parameters we use during evaluation are not secure. We chose them as a proof of concept. Secure parameters would increase the size of ciphertexts, reduce the number of steps we can evaluate before requiring client interaction, and generally increase resource requirements. (3) The interactive phases expose the internal of the network to the client. The client could use the information to learn the weights of the network. This puts the privacy of the server-side model at risk.

We will address these issues in the next chapter by evaluating the approach on secure parameters and proposing an architecture that eliminates the need for interactive phases by reducing the depth of the network. We can use an alternative approach to preserving the server model's privacy. We can mask the internal state of the network by adding some

randomness to the data before sending it to the client and removing it once the client sends the data back. This adds two homomorphic additions, one to add the randomness and one to remove it, to each interactive phase. We can do this in our approach since the client does not alter the data; the client only decrypts and re-encrypts the data. Bakshi and Last [70] evaluate an approach where the client computes the evaluation function during the interactive phase. This allows the authors to use a wide range of activations and does not limit them to HE-friendly functions. However, the use of non-HE-friendly functions requires interaction after every time step. This increases the communication overhead However, it also allows the use of smaller crypto parameters, since there is an interactive phase after every dot product. This means that the authors only need parameters that support a single multiplication, substantially reducing the size of objects that need to be transferred. However, these advantages come at the cost of model privacy. Here, the client receives the internal state after every time step and prior to the activation, providing them with more detailed and frequent information than in our proposed approach. Furthermore, in this setting, we can not add randomness to the data sent to the client. The non-linear activation function changes the value so we can no longer simply remove it on the server side.

CHAPTER 6

NON-INTERACTIVE PRIVACY PRESERVING RECURRENT NEURAL NETWORK PREDICTION WITH HOMOMORPHIC ENCRYPTION

In the previous chapter, we present an interactive approach for privacy-preserving RNNs. There, we also outline some issues with the interactive approach. In this chapter, we present a new architecture that reduces the depth of the network while retaining some of the recurrent properties. A lower multiplicative depth allows us to still use leveled homomorphic encryption and not rely on interactive phases or bootstrapping. For example, Jiang et al. [162] propose an approach for running Gated Running Units (GRU) over homomorphic encryption. Their approach relies on a custom bootstrapping procedure, which takes 2.8 minutes with 48 threads and needs to be applied every 4th timestep. Assuming a sequence length of 64, the approach requires roughly 45 minutes of bootstrapping. The authors report an additional 90 seconds per timestep, not considering bootstrapping, for a GRU layer with 64 hidden units, totaling 8,448 seconds for the 64 time steps. However, their ciphertext packing scheme only packs a single instance into a run.

The content of this chapter was previously published as *Non-interactive Privacy Preserving Recurrent Neural Network Prediction with Homomorphic Encryption*

Robert Podschwadt and Daniel Takabi in *IEEE 14th International Conference on Cloud Computing (CLOUD), Pages 65-70, 2021* [3], and has been slightly edited here.

6.1 Introduction

Machine learning (ML) is often performed in the Cloud, using machine learning as service (MLaaS) like Amazon Web Services [163], Google Cloud [164], and Microsoft Azure [165]. In MLaaS the client owns the data and computation is performed in the Cloud. The downside of MLaaS is it requires access to the clients' data. This creates security and privacy concerns making it unacceptable in many situations. E.g., medical data enjoys strong legal protections and can not easily be shared. In such a scenario, using MLaaS is nearly impossible. Even without legal issues, privacy concerns still exist. Many privacy preserving machine learning (PPML) approaches, based on differential privacy (DP), secure multiparty computation (SMC), Homomorphic Encryption (HE), and hybrid approaches, have been proposed to address this. HE allows mathematical operations on encrypted data without the need for decryption. Although many HE-based approaches have been studied, most of them focus on CNNs [27, 125, 61, 67], with RNNs largely being ignored due to their more complex nature. Different types of RNNs exist, such as Long Short Term Memory (LSTM) [71], Gated Recurrent Unit (GRU) [166] and a fully connected variant [86] often called simple RNNs. Here, we focus on simple RNNs. In prior work, Lou and Jiang proposed a system [58] that implements an LSTM based on TFHE [167]. They used fixed point encoding, which leads to a drop in accuracy. Podschwadt and Takabi [2] and Bakshi and Last [70] proposed systems for simple RNNs, based on CKKS [49]. Both require interactive phases. The particular challenge of running RNNs over HE lies in their multiplicative depth (MD). MD is the number of multiplications required to evaluate an algorithm and is the

main limiting factor of HE-based solutions. The MD of RNNs is often much larger than the MD of feed forward networks with the same number layers. RNNs require polynomial activation functions (PAF) of at least degree 3, which further increases the MD. We present an approach for privacy-preserving RNN inference based on HE, in which the data owner encrypts the data and sends it to the Cloud for processing. The Cloud performs inference on the encrypted data and sends the encrypted result back to the client. Our approach follows the ideas presented in [2]. Unlike [2], we do not require any interaction during computation, allowing for offline processing. Our proposed RNN architecture, called parallel RNN blocks (PRB), reduces the dimensionality of the inputs to the RNN by splitting them into smaller chunks which are processed by an RNN layer. This reduces MD of the network. We show that our architecture can perform well on plain and encrypted data on sentiment analysis tasks. We make the following contributions:

- we present an approach for privacy-preserving, non-interactive RNN inference scalable to real-world datasets.
- we introduce a new architecture which reduces the MD of the RNNs by splitting inputs into evenly sized chunks that can be processed by less deep RNN blocks and recombined later. This new architecture allows for non-interactive privacy preserving RNN inference using HE.
- we evaluate the proposed approach on two different data sets, one large real word dataset and a commonly used benchmark dataset, to show the feasibility of our proposed approach.

- we experimentally compare our approach with previous interactive approaches
- we investigate how training RNNs, on plaintext, with PAFs impacts performance

6.2 Related Work

CryptoNets [27] was one of the earliest work on privacy-preserving neural networks based on HE, further improved by Chou et al. [59] by pruning the network parameters. Hesamifard et al. [168] focus on training and evaluating polynomial CNNs over plaintext, and HE encrypted data. Zhang et al. [68] perform encrypted speech recognition using a CNN. The last part of the network that matches the output to actual text is performed on the client side. However, most of these approaches are focused on CNNs only and do not work for RNNs. Feng et al. [169] perform sequence to sequence transformation based on LSTM models, using SMC to protect the privacy of the data. This approach might be less attractive to Cloud providers since it requires them to share part of the model with the client. Most other work on privacy preserving RNNs require communication between the server and the client. Bakshi and Last [70] have the client compute the activations on plaintext. Unlike our work the authors considers only small networks with few units and short input sequences. Liu et al. [18] transform a regular LSTM into a privacy preserving LSTM by replacing activation functions with approximations, which are evaluated by the client. This leaks some information about the model to client. In Podschwadt and Takabi [2] the client does not perform computation but is involved for noise removal. While using the client for computation reduces the computational load on the server, it increases communication and can also reveal information about model to the client. Using the client only for noise removal

reduces the amount of communication required and information leaking, but it increases the computational cost on the server side. Our proposed approach removes all the communications between client and server during the computation. Lou and Jiang [58] also created a non-interactive, privacy-preserving neural network framework based on TFHE [167]. This allows for the evaluation of LSTM cells over encrypted data. They achieve this by replacing the expensive matrix operations required by LSTMs with much cheaper shift operations. Unlike our approach their work loses accuracy over a plaintext model due to the use of fixed point encoding.

6.3 The Proposed Approach

Our goal is to enable privacy preserving inference using RNNs in a non-interactive manner. The main concern is the privacy of the user's data. The Cloud should not be able to learn anything about the client's data. In turn the client should learn as little as possible of the Cloud's model. The system should be as efficient as possible and deliver high quality predictions. The last two requirements are more flexible. We are not willing to trade privacy for either performance or efficiency but we can trade off between prediction quality and efficiency.

6.3.1 Threat Model

Our system preserves the privacy of the data assuming all parties are honest but curios. They will execute the protocol and not deviate from it but they will try learn any information they can. The client owns the data and private key. The Cloud has a trained model that

can be run on encrypted data. The word embedding lookup can not be performed efficiently on encrypted data. As in [2] the client performs the embedding operation and sends the encrypted embeddings to the Cloud. By using pretrained embeddings this does not leak information to the client, since the embeddings are publicly available and are not a secret of the model provider. They are trained independently from the Cloud model, and do not contain any information about the model or the training data. Therefore, they can be shared without revealing any information.

6.3.2 Multiplicative Depth and Noise growth

The MD of an HE algorithm is the number of sequential multiplications that are necessary to complete it. In a simplified way of looking at an RNN, it can be expressed as the repeated application of the activation function f to the internal state $s_t{=}f(x_t \cdot w + s_{t-1} \cdot v)$ of a neuron. The inputs x_t are fresh ciphertexts that no computation has been performed on yet. We can ignore them for this analysis, giving us: $s_t{=}f(s_{t-1} \cdot v)$. We can expand s_{t-1} to $s_{t-1}{=}f(s_{t-2} \cdot v)$. Repeatedly applying this expansion yields $s_t{=}f(\dots f(f(x_0 \cdot w) \cdot v) \dots) \cdot v)$. The inner product $\cdot$ and the activation function f are applied t times to x_0. Our activation has an MD of three and the inner product has an MD of one. An input sequence with t time steps has an MD of $4t$. A feed forward network with the same PAF would need t layers to have the same MD as the RNN. This is often too deep to run with HE. In [2] the authors solve the problem by sending the data to client to reset the noise level. Our proposed system is able to perform the computation without client interaction. Further, we use stronger CKKS parameters than [2], to give us 128-bit security. Due to the noise growth, the training

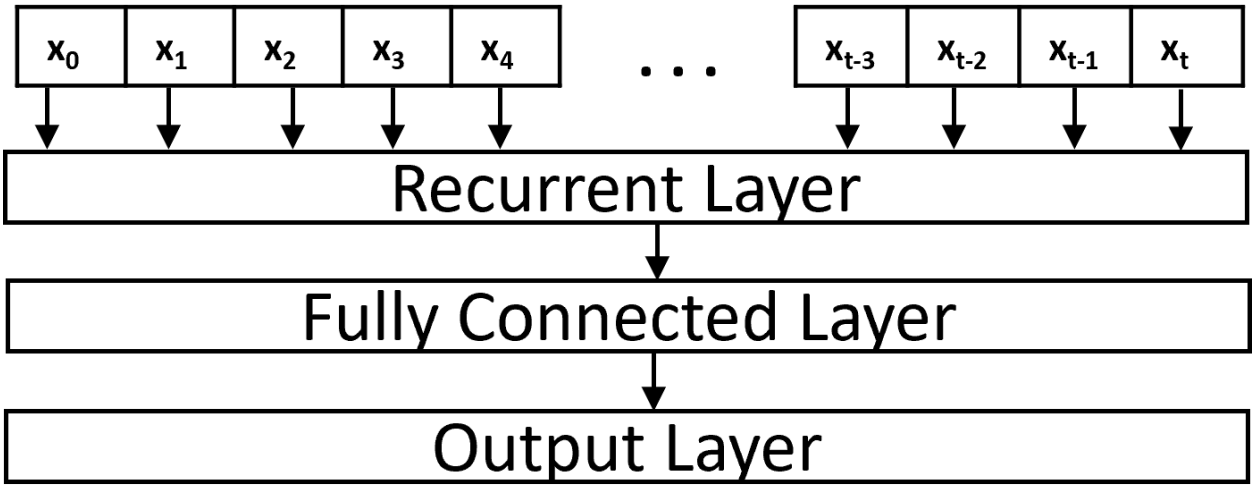


Figure 6.1: A standard RNN

of the model is done on plaintext data. Training on encrypted data has been shown to be computationally expensive [81].

6.3.3 Parallel RNN Blocks

As discussed in Section 6.3.2, the MD of RNNs grows with the number of elements in the input sequence. Reducing the depth of an RNN by reducing the length of the input sequence will result in the loss of information. We propose an alternative solution for running RNNs over encrypted data, based on the observation that shorter sequences reduce the MD of RNNs. We want to keep the strengths of RNNs and not discard information. To achieve this we split the input sequence into shorter sub-sequences of equal length. The sub-sequences are fed into an RNN layer. The outputs of the RNN layer is concatenated and fed into a fully connected layer, see Fig. 6.2. We refer to the RNN layers processing the sub-sequences as parallel blocks since they are completely independent from each other and can be computed

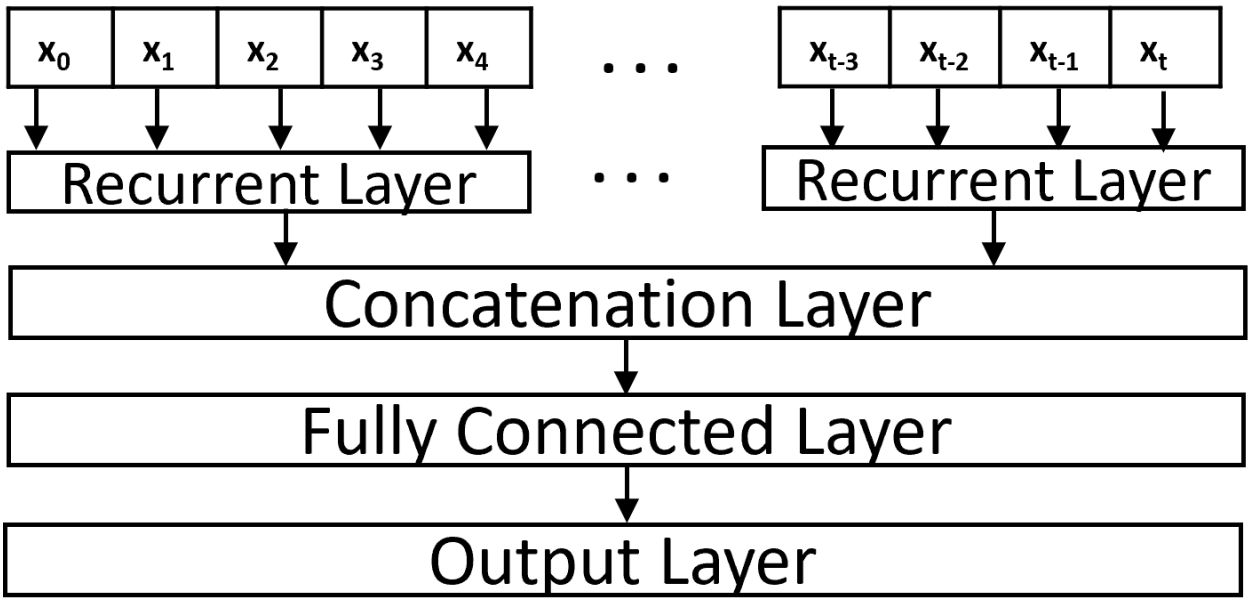


Figure 6.2: Parallel RNN blocks

Figure 6.3: Architecture diagram

at the same time. A traditional RNN (Fig. 6.1) can not be executed in parallel since every step requires input from the previous one. This is true for our architecture as well but this interdependence is limited to within a block. There are at least two ways of setting up the RNNs within the blocks. 1) every RNN has its own independent weights. The weights within that block would only be trained on the particular sub-sequence of the training data associated with this block. 2) the weights are shared between all the blocks. In this case, all blocks would train on every sub-sequence of the training data. While no data is discarded from the input sequence there is some loss of information at the block boundaries. In a regular RNN the network has the entirety of the input sequence in its internal state. In our architecture the blocks have no input from the other blocks. However, our experiments show

that the fully connected layer following the concatenation layer mitigates the effect. The number of blocks depends on the length of the input sequence and the desired MD of the network. The splitting has not impact on the security of the system. It is performed on the ciphertexts in the Cloud and reveals no further information.

6.4 Evaluation and Experimental Results

6.4.1 Training with polynomial approximations

Using PAFs in neural networks presents a problem during training as they have different properties than standard activation functions. Some activation functions, e.g Sigmoid and Tanh, are bounded. Most activation functions are monotonic increasing. This not true for polynomials. Polynomials can not be bounded. They can only be bounded from either above or below. A function $f(x)$ is bounded from above if $\forall x \exists B(f(x) \leq B)$ and bounded from below if $\forall x \exists B(f(x) \geq B)$. Non-monotonic increasing and unbounded functions are harder to optimize due to exploding/vanishing gradients [170] and a none smooth error surface. In our experiments training RNNs with functions such as x^2 or ReLU lead to the network not converging at all. Therefore, we choose a degree 3 polynomial $f(x)$=$-0.0016357430301874 8x^3 + 0.249476365628036x$ as our activation, derived using the method described in [26]. To prevent exploding gradients we keep the input values of f between the local extremes of f. We do this by applying L_2 regularization with λ=0.02 to the weights of the RNN. We found a good heuristic for λ is 2×10^{-4} times the number of units in the recurrent layer.

6.4.2 Data and Preprocessing

We evaluate our proposed solution on a sentiment analysis task. The goal is to predict if a given text is negative or positive. We use a data set of user reviews of movies and TV shows taken from the Amazon online store [171]. The dataset contains 1,697,533 reviews with ratings. A one star rating is the lowest possible and a five star rating the best possible rating. We use the review text as the input to our system and the star rating as the labels. We simplify the task by grouping ratings into two larger groups. One and two star reviews are combined into a negative class and four and five star reviews form the positive class. We drop reviews with three stars. We use the 20k most frequent words as our dictionary. As a word vector representation we choose pretrained 100-D GloVe [102] embeddings. The length of an instance is 64 tokens and we truncate or pad as necessary. As evaluation metrics, we use balanced accuracy and F1 score. We evaluate each model architecture's performance using 5 fold cross validation. Additionally we evaluate our models on the IMDb [172] test data, containing 25k movie reviews, labeled as either positive or negative. We perform the same preprocessing steps and use the same vocabulary that we use on the Amazon data. Performance on IMDb is expected to be lower, since the data is not from the same distribution as the training data. But it can be useful as an indicator of the model's generalizability.

6.4.3 Plaintext Performance

We do all the training and plaintext evaluation using TensorFlow 2.4. As a baseline we train a simple RNN model consisting of an embedding layer with 128 and Tanh activation,

a fully connected layer with 630 units and ReLU activation, and a fully connected layer with 1 unit and Sigmoid activation. We use the Adam optimizer with default parameters and binary cross entropy as the loss function and train for 32 epochs. This gives us a baseline against which we can compare our proposed solution. To make a fair comparison we try to keep the number of trainable parameters as similar as possible. All models have a similar number of parameters unless stated otherwise. The baseline model achieves a median balanced accuracy of 81.86% and a median F1 of 91.85%. It is not possible to run the baseline model over encrypted data in its original form. Neither the Tanh function nor the ReLU function can be applied to encrypted data. We use the same model architecture but replace the Tanh function with f and the ReLU function with the square function. Additionally, we add L_2 weight regularization with a factor of 0.02 to prevent the training from collapsing. The regularization factor is experimentally determined. After 32 epochs the model achieves 87.36% median F1 (see Table 6.1 for more details). The model suffers a 5% drop over the baseline. This shows nonetheless that RNNs can be trained on larger data sets with PAFs. However, it is necessary to make the correct adjustments in order to prevent exploding gradients. We found the level of regularization needs to increase with the number of neurons of in the simple RNN. A RNN model with 512 neurons in the RNN layer needs an L_2 regularization factor of at least 0.1 to prevent a training collapse. The number of neurons isn't the only component that factors into the amount of regularization that is required. It also depends on the PAF used.

Although all components are HE friendly, the model can not be run on encrypted data

due to its MD, as shown in [2, 70]. These approaches require communication to remove the noise during computation. Our proposed architecture does not require communication. For the first experiment we choose ten blocks, leading to sub-sequences of length 6.4. By using length 14 sub-sequences and an overlap of two between the blocks we can make sub-sequences integer length and the same size. To evaluate the feasibility of this architecture, we first train models with Tanh and ReLU activation. We find 128 neurons in the RNN layers works best. We run experiments with shared weights and individual weights. Since models with individual weights have more trainable parameters, they have theoretically the capacity to learn more. We use parameters for both architectures that give the models a similar number of parameters to the baseline model. Additionally we also train models with individual weights that have the same number of neurons in the RNN blocks and the fully connected layer as the models with shared weights, resulting in roughly 3x the number of parameters. We refer to these models as large. The model with shared weights consists of 10 blocks with 128 units and an overlap of two. The following fully connected layer has 64 units. The model with individual weights has 65 units in the RNN blocks and 6 units in the fully connected layer. To run on encrypted data we replace the activation functions with HE friendly polynomials. We train more models with the same parameters using PAFs. The RNN blocks models with PAFs perform similarly to the RNN blocks with Tanh.

Table 6.1 shows that the F1 scores of the models without PAFs are within 1 percentage point of each other at 87.08%, 87.96% and 87.73%. On IMDb the large model with the individual weights has a slight advantage, with 74.83%, over the other models, at 73.51%

Table 6.1: Median F1 score and median balanced accuracy (BA) of the different architectures on different data sets

Model	Amazon		IMDb Test	
	F1(%)	BA(%)	F1(%)	BA (%)
Simple RNN (base line)	91.85	81.86	77.65	78.53
Simple RNN f	87.36	77.91	72.29	73.94
PRB	87.08	79.80	73.51	75.89
PRB individual weights	87.96	77.56	66.67	69.75
PRB individual weights (large)	87.73	82.18	74.83	77.64
PRB f	86.31	78.36	72.99	74.79
PRB individual weights f	86.02	78.05	68.10	72.52
PRB individual weights f (large)	85.90	78.51	70.33	73.99

and 66.67%. These results are 5% below the baseline and on par with RNNs with PAFs. This shows that our architecture can achieve similar results to a traditional architecture. Using PAFs (f) reduces the performance slightly. The model with shared weights and PAFs performs closest to the baseline at 86.31% on the Amazon test data and 74.79 % on the IMDb data. Models with individual weights suffer a greater drop on IMDb. When using individual weights a specific block is only ever trained on a specific section of the input texts. It is unlikely that text from other distributions have the same information in that specific section. Going forward, we will focus on models with shared weights.

To assess the impact of the number of blocks we create 6 model architectures with a different number of splits. The RNN block consists of 128 units. We once again control for number of trainable parameters in the network, by adjusting the number of units in the fully connected layer. The number of units is calculated as $\frac{1280}{\#splits}$. We use 5 fold crossvalidation and train the models for 32 epochs on the Amazon move reviews. Additionally we compare impact of Tanh activation and PAFs in conjunction with the number of splits. The results are depicted in Fig. 6.4. There are a number of noteworthy observations. 1) Models using Tanh

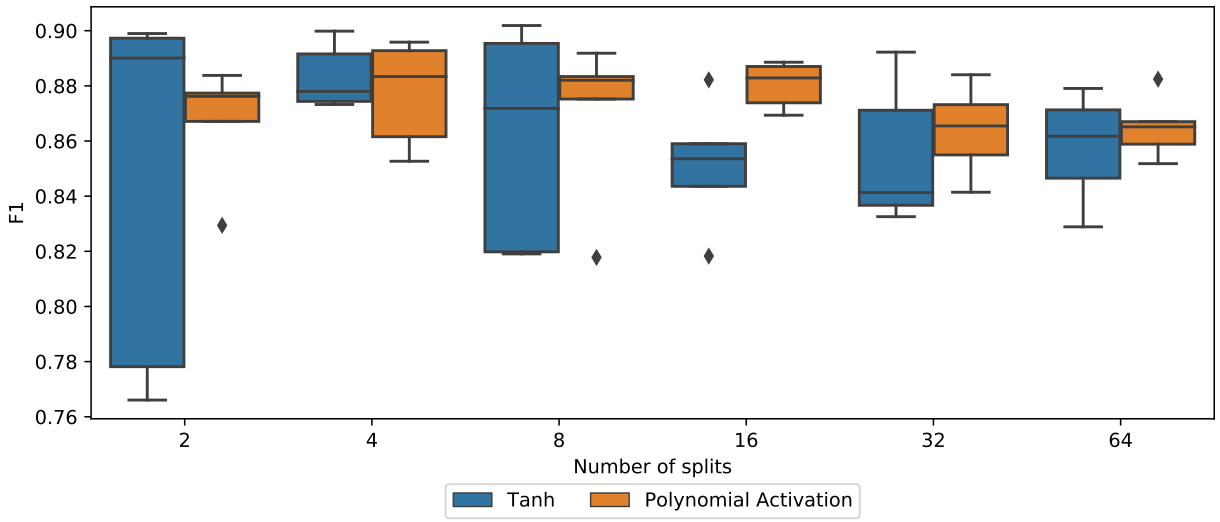


Figure 6.4: Impact of the number of splits on model performance

vary widely in terms of performance compared to PAFs, and the variance in performance is often greater than that of PAFs models. 2) The performance gets lower as the number of splits increases. This seems intuitive since the greater the number of splits the shallower the models becomes. 3) The sweet-spot, for performance on polynomial models, is 16 splits. All experiments show that it is crucial with PAFs to keep the input values into the activation functions in a very specific interval, which can be achieved using L_2 regularization.

6.4.4 Results on Encrypted Data

To test the performance of our framework on encrypted data, we implement the above-mentioned networks in C++17. We use the CKKS implementation of HElib [173] with 32 bit of precision for all our networks. Our solution is designed to process large batches of data, using SIMD batching to, offset the overhead of HE. The data owner needs to create 6400 ciphertexts and encodes multiple instances, up to the batchsize, in the ciphertexts.

Each ciphertext holds one dimension of all instances in the batch as described in [27]. The batchsize depends on the crypto parameters. To guarantee privacy, we choose the crypto parameters that provide a security level of at least 128 bit. We use: L=300, M=2^{16} for 64 splits; L=450, M=2^{16} for 32 splits; and L=900, M=2^{17} for 16 splits, resulting in a security level of 218, 153 and 157 bits respectively. The batch size is $\frac{M}{4}$. Fewer splits require too much memory. The experiments are preformed on a Microsoft Azure Standard E32-16s_v3 VM running Ubuntu 18.04 with 16 CPU cores and 256 GB of RAM. Truncating and the embedding operation are performed by the data owner prior to encryption. The encrypted embeddings are sent to the Cloud where the privacy preserving inference is performed. After the computation is complete the encrypted result is sent back to the client. The model with 64 splits runs for 19.5 minutes, processing 16,384 instance at once, almost 14 $\frac{\text{instances}}{\text{second}}$. The model with 32 splits can uses the same batch size but takes almost 35 minutes, or 7.88 $\frac{\text{instances}}{\text{second}}$. The model with 16 splits needs 2 hours and 55 minutes to process 32768 instances, or 3.1 $\frac{\text{instances}}{\text{second}}$.

Over 90% of the execution time is spent in the RNN blocks. Our system is highly parallel. With enough cores available we could run every neuron in every block in parallel. In this particular architecture we could make use of up to 2048 cores. In a distributed setting each block could be run on it's own node. With communication about 2 GB per block, this could be transferred easily in a data center. Memory is more limiting. The 16 split model takes up to 215 GB of RAM during processing. Models with fewer splits can not be run on our machine due to memory limitations. Even the smaller 64 splits model requires over a 100

Table 6.2: Resource requirements for encrypted (Ctxt) and plain text (Ptxt) execution

Splits	Running-time (s)			RAM (GB)		
	Ptxt	Ctxt	increase	Ptxt	Ctxt	increase
64	25	1179	47x	3.2	120.5	37x
32	40	2080	52x	2.5	106.2	42x
16	97	10527	109x	3.3	215.6	65x

GB of RAM. The crypto parameters influence the size of the data transferred. The client needs to upload 18.75 GB, 25 GB and 100 GB for 64, 32 and 16 splits model respectively. The response from the Cloud is comparatively small with 3, 4 and 16 MB respectively. The computational and memory overhead is shown in Table 6.2. Running on encrypted data introduces 37-109 times overhead.

6.4.5 Comparison with interactive approaches

Prior works on privacy-preserving RNN inference using CKKS [70, 2], in contrast to our work, rely on interactive phases for noise removal. We implement the interactive approach described in [2, 70]. For comparison we use the *Simple RNN f* model (Table 6.1) and crypto parameters discussed earlier. The model performance (87.36%) is close to our model using parallel RNN blocks (86.32%). The number of interactive refresh rounds, for a given set of crypto parameters, is the same as the number of splits. We do not use an actual client and simulate the interactive phase by performing the decryption and reencryption on the same machine. This eliminates any network latency, leading to a lower running time of the interactive approach than it would be in practice. Table 6.3 shows that our approach performs faster, even in this ideal setting. On the same batch of data, on the same hardware, our non-interactive approach performs 1.3 to 2.6 times faster. However, this improvement

Table 6.3: Resource requirements for RNNs using an interactive phase (ia) and our non-interactive approach (nia)

Splits/	Running-time (s)			RAM (GB)		
ia Phases	ia	nia	increase	ia	nia	increase
64	3076	1179	0.38x	24.1	120.5	5x
32	3895	2080	0.53x	32.8	106.2	3.2x
16	14036	10527	0.75x	145.3	215.6	1.5x

in running time comes at a cost of 1.5 to 5 times increase in memory consumption, which could likely be narrowed with optimization. The most important difference of interactive approaches and our approach is the need to transfer data during the computation. Client and server need to transfer an extra 64, 42, 94 GB when using 64, 32 and 16 refresh rounds respectively. The transfer from the Cloud to the client accounts for roughly 60%. Data transfer using interactive phases increases by a factor of 2 to 3.

6.5 Summary and Discussion

In this chapter, we presented and evaluated a new architecture, parallel RNN blocks, that allows us to run RNNs over HE data in a non-interactive manner. Previous solutions have not been able to achieve this. There is a small loss in performance compared to the baseline. The memory requirements could be reduced using different ciphertext packing techniques.

In this chapter, we propose an architecture for RNNs over encrypted that does not require interactive phases or bootstrapping. We achieve this by splitting the input sequence into shorter subsequences and processing these by an individual RNN, called a block. We combine the outputs of the blocks and feed them into a fully connected layer. In this chapter, we use concatenation to combine the block outputs. There are two issues with concatenation. First, as the number of splits increases, so does the dimension of the concatenation. In this chapter,

we use a maximum of 64 splits, each producing 128 output values. The concatenation has 8,192 values, resulting in a following fully connected layer with a large number of weights. While a large number of weights in a fully connected layer does not increase the depth, it does increase the computation time. Secondly, on plain data, RNNs can be run on inputs of varying lengths. The block architecture, in itself, permits this. Particularly when the blocks share the weights, we can split a sequence of any length into subsequences and process them using a block. The issue arises in the concatenation layer and the subsequent fully connected layer. The fully connected layer expects an input of fixed dimension. This can be produced by the concatenation layer only if it has a fixed number of inputs, making it incompatible with varying-length inputs. In the following chapter, we investigate two other approaches for combining the block outputs.

6.5.1 Statistical significance Test

To establish if the difference in performance between the various models with different activation functions and different numbers of splits is statistically significant, we perform a Mann-Whitney U test. We compute the pairwise p value for all model combinations on the results of the 5-fold cross-validation. In this case, the null hypothesis is that two models have different performances, which we can reject at $p < 0.05$. Fig. 6.5 shows the pair-wise p-values for the models with Tanh and polynomial activation. Using the model Tanh model with 0 splits, a vanilla RNN, as the baseline, we can conclude that most models employing splits have a statistically different performance. However, the performance difference for all models against the polynomial baseline is statistically significant. The same is true when we

compare the models with the same number of splits but different activations. We do not find a significant difference here either. This leads us to conclude the approach of RNN blocks with concatenation, presented in this chapter, achieves comparable results to the polynomial baseline RNN and performs slightly worse than the Tanh baseline.

6.5.2 Relation to Truncated Backpropagation Through Time

Truncated Backpropagation Through Time (TBPTT)[174] is a technique for training RNNs that can make training efficient for long input sequences. The idea behind TBPTT is to run a gradient computation and weight update after a fixed number of steps k_1 instead of just at the end of the sequences. Additionally, the gradient is not computed all the way to the beginning of the sequence but only for k_2 steps and is truncated there, hence the name. Our approach is similar in that we also truncate the gradient computation. However, we truncate the computation by explicitly splitting the sequence into subsequences. This splitting breaks apart some of the temporal dependencies at block boundaries. TBPTT can control this by choosing the values of k_1 and k_2, which control how far back temporal dependencies should be considered. During inference, TBPTT does not impact the computation; it only influences training. Therefore, it does not impact how many previous steps have gone into the current state during inference. However, this is exactly what our approach seeks to limit. The number of steps that go into a state impacts the multiplicative depth of the evaluation. We, therefore, need to truncate the forward pass, and we need to truncate it to short subsequences. Due to the constraints of HE, we can not handle long subsequences. Because of this truncation, we lose inter-block temporal dependencies.

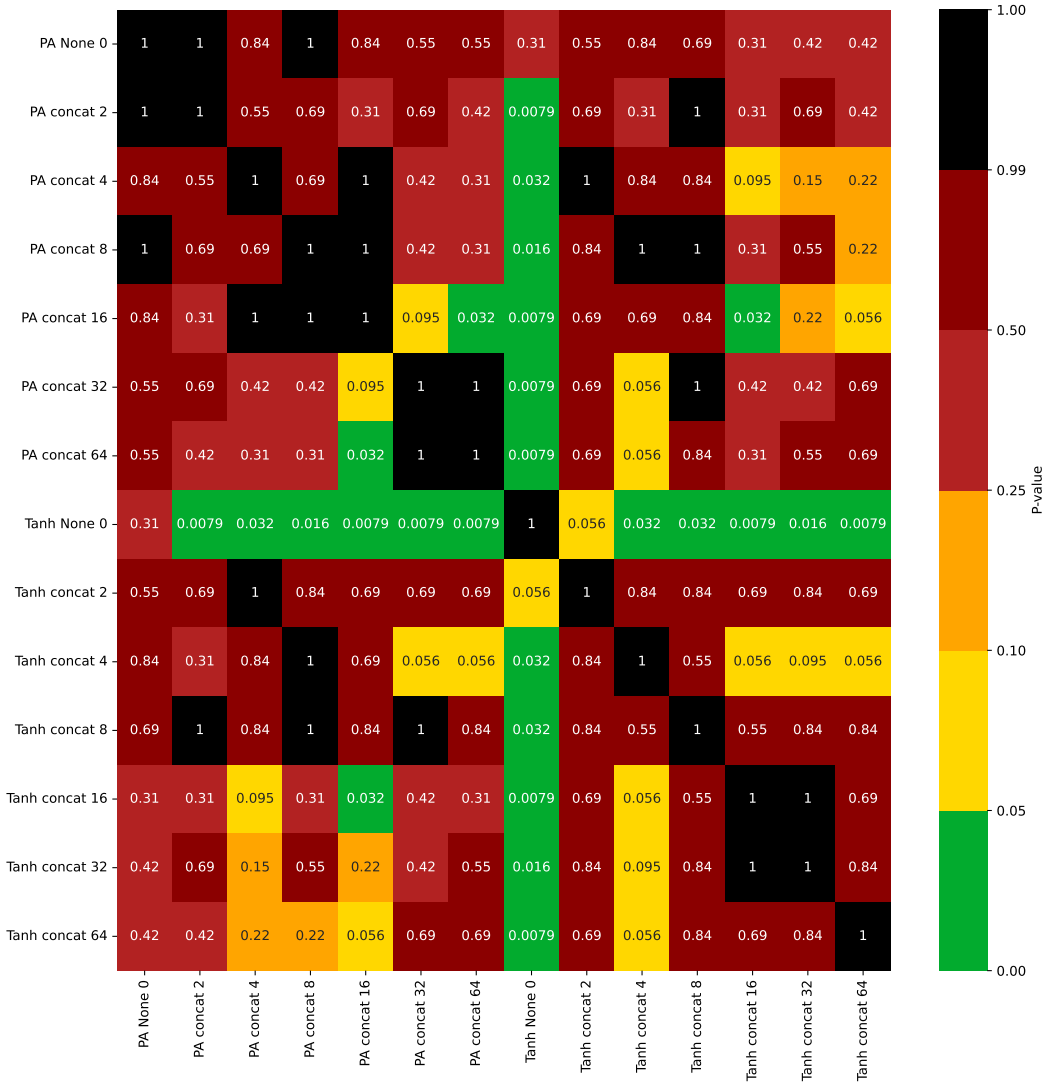


Figure 6.5: P-values computed by the Mann-Whitney U test on the F1 scores of 5-fold cross-validation. Computed for all model pairs using the concatenation strategy and the two baselines PA None 0 and Tanh None 0.

CHAPTER 7
ADDITIONAL RNN-BLOCKS ARCHITECTURES

As discussed in the last chapter, the use of concatenation to combine the results of the RNN blocks comes with some drawbacks. Fig 7.1 shows the architecture without the fully connected layer following the concatenation layers. The advantage of using concatenation is that it does not require any computation on encrypted data since we use batch packing. It only requires a rearranging of the data structure that holds the ciphertext objects. However, as discussed previously, it creates a large vector of data, which is needed in its entirety for computing the next layer. If we assume that we are limited in the number of time steps a block can process due to the multiplicative depth, the contamination vector will grow longer as the input sequences to the networks grow longer.

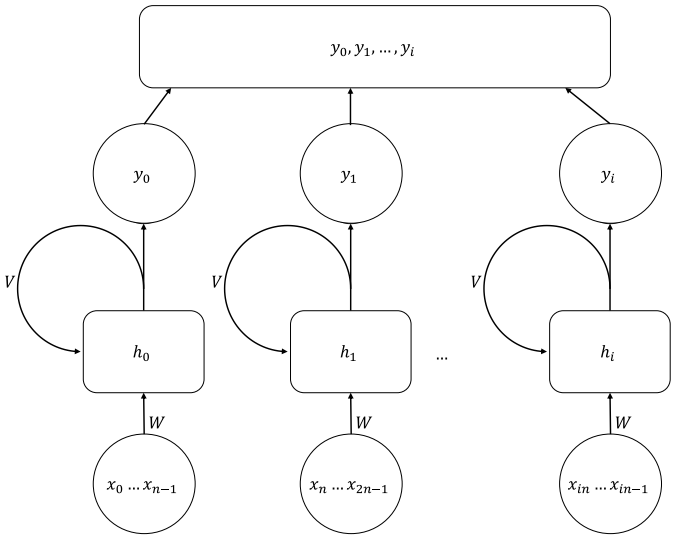


Figure 7.1: The RNN Blocks architecture. The input sequence $x_0, x_1, ...x_l$ is split into i chunks of length n. We process each chunk with an RNN and combine the outputs $y_0, y_1, ..., y_i$ for further processing.

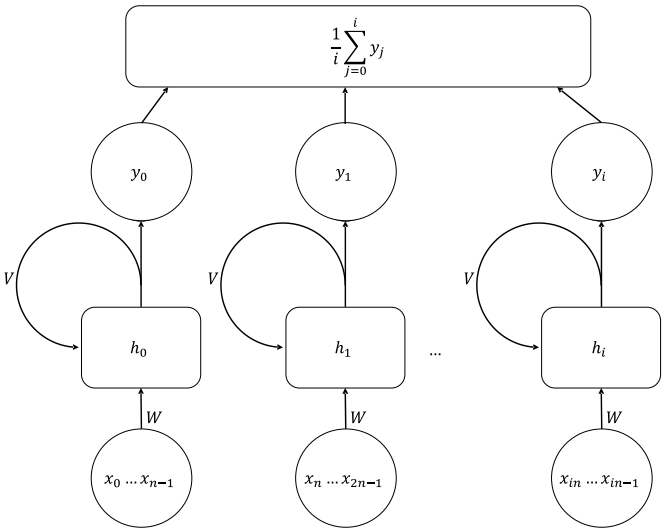


Figure 7.2: Averaging the outputs $y_0, \ldots, y_i$ of $i+1$ RNNs blocks. The input to each block is a subsequence of length n.

7.1 Averaging block output

An approach to addressing the issues mentioned earlier is to average the output of all the blocks. To do so we add all the outputs of blocks $y_0, \ldots, y_i$ elementwise and divide it by the number of blocks i, shown in Fig. 7.2. Averaging the outputs has several advantages. It requires little additional computation: only i additions and one multiplication. Additionally, we can perform the additions successively, allowing us to keep the number of objects we need to keep in memory small. With concatenation, we need to keep all $y_0, \ldots, y_i$ for further computation. Averaging also allows us to use an arbitrary number of input blocks, placing no limitations on the length of the sequences that we want to process while also not increasing the multiplicative depth. However, it could be the case that averaging loses us information.

7.1.1 Evaluation

We use the same data set and evaluation strategy as in Section 6.4. We use the Amazon Movie Review Data [171], which we transform into a binary classification problem from five classes. We again run 5-fold cross-validation for 64 epochs but stop early if the loss validation loss does not improve for 12 epochs. We use ten percent of the training data for validation. We use 128 units in the RNNs and share weights across the blocks. The averaging layer feeds into a fully connected layer with 630 units, followed by a single output neuron. We build a set of none HE-friendly models using Tanh as the activation function inside the RNNs and ReLU in the fully connected layer. The HE-friendly model uses a degree three polynomial activation (Eq. 7.1) inside the RNNs and x^2 in the fully connected layer. To prevent exploding gradients, especially with the polynomial activations, we use L_2 weight regularization with a strength of 0.02 for the recurrent layer.

$$-0.00163574303018748x^3 + 0.249476365628036x \tag{7.1}$$

Table 7.1 and Fig. 7.3 show the results of the experiments. As expected, as the number of splits increases, the performance decreases slightly. The drop is expected since we lose some temporal dependency at the block boundaries. Considering the RNN using Tanh as the baseline, we lose a maximum of 8.7% average F1 score between the baseline and the 64 split model using polynomial activation. Based on a Mann-Whitney U test, we conclude that the difference in performance between these two models is statically significant, with $p = 0.0079 < 0.05$. Fig. 7.4 shows the pairwise p values for all averaging models and the

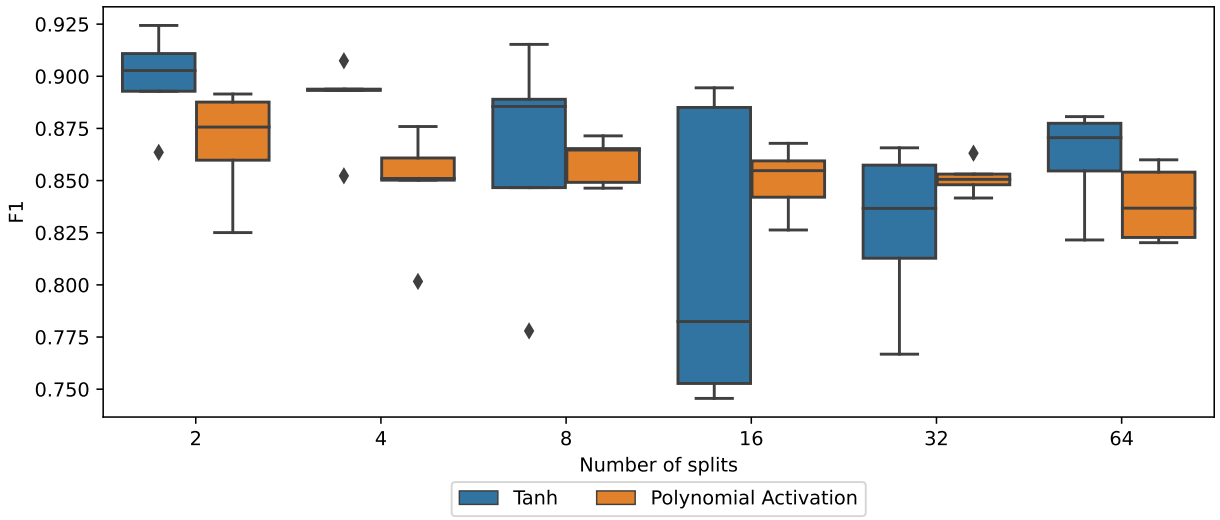


Figure 7.3: Result of 5-fold cross-validation for RNN blocks with averaging using different numbers of splits and comparing the HE-friendly model (PA) to the non-HE-friendly model (Tanh).

baselines computed on the F1 scores of the 5-fold cross-validation. Using the same criteria, we can conclude the difference in performance between the baseline and the Tanh models with two and four splits is not statically significant; the difference between the Tanh model with eight splits and the baseline probably might be statically significant $p = 0.056$, and the difference for 16, 32, 64 splits Tanh models is statically significant. When comparing the HE-friendly models (polynomial activation) to the Tanh baseline, the difference in performance is significant for all models but the HE-friendly baseline. Comparing the averaging models with Tanh to the averaging models with polynomial activation, we can conclude that the difference in performance is probably significant for 2 and 4 splits and not significant for a larger number of splits.

While there is a significant drop in performance between the baseline and models with a larger number of splits, models with a larger number of splits are attractive to us. A larger

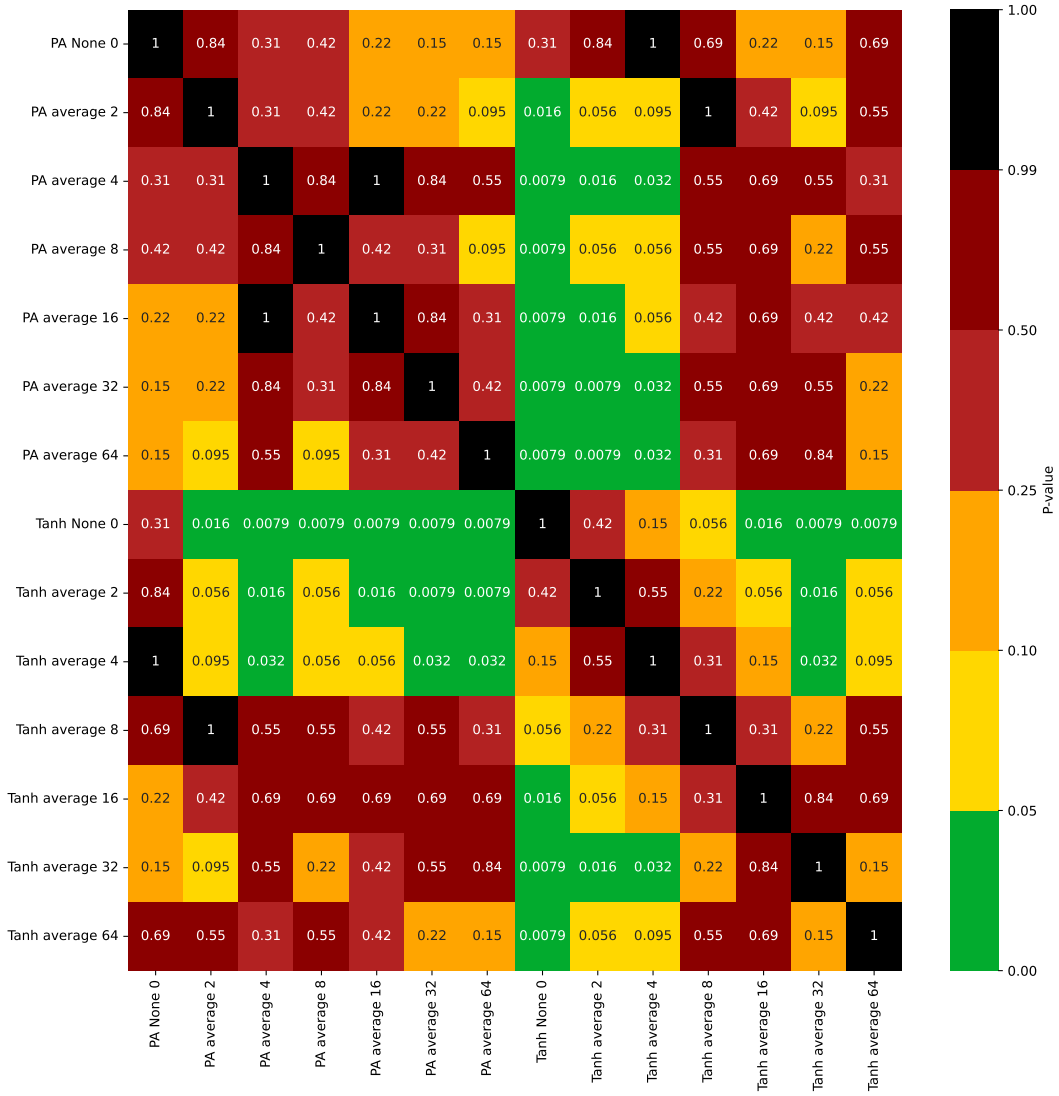


Figure 7.4: P-values computed by the Mann-Whitney U test on the F1 scores of 5-fold cross-validation. Computed for all model pairs using the averaging strategy and the two baselines PA None 0 and Tanh None 0.

Table 7.1: Average F1 score for 5-fold cross-validation of the averaging models with Polynomial Activation and Tanh activation, and a varying number of splits. The length of a subsequence is 64/#Splits. Performed on the Amazon Movie Review Data. 0 splits indicate the baseline RNN model that is not using blocks.

# Splits	Activation	F1
0	Polynomial Activation	0.874
	Tanh	0.912
2	Polynomial Activation	0.868
	Tanh	0.899
4	Polynomial Activation	0.848
	Tanh	0.888
8	Polynomial Activation	0.859
	Tanh	0.863
16	Polynomial Activation	0.850
	Tanh	0.812
32	Polynomial Activation	0.851
	Tanh	0.828
64	Polynomial Activation	0.839
	Tanh	0.861

number of splits reduces the multiplicative depth and, thereby, the resource requirements. The performance difference of models with 8 or more splits is not statistically significant, even down to one split, which is technicity, not an RNN block, since it only processes one sequence element. This could suggest that we lose too many temporal dependencies when the number of splits gets too large. However, it does provide us with an HE-friendly way for sequence processing. In the next section, we will evaluate an alternative approach to combining RNN blocks that aims to preserve the recurrent structure more.

7.2 Hierarchical Blocks

In the previous section, we evaluated averaging as a strategy for combining the outputs of blocks. While it features some desirable properties, the evaluation suggests that we might lose too many temporal dependencies when we split the input sequence into too many subsequences. To address this, we now present an architecture that uses hierarchical RNNs

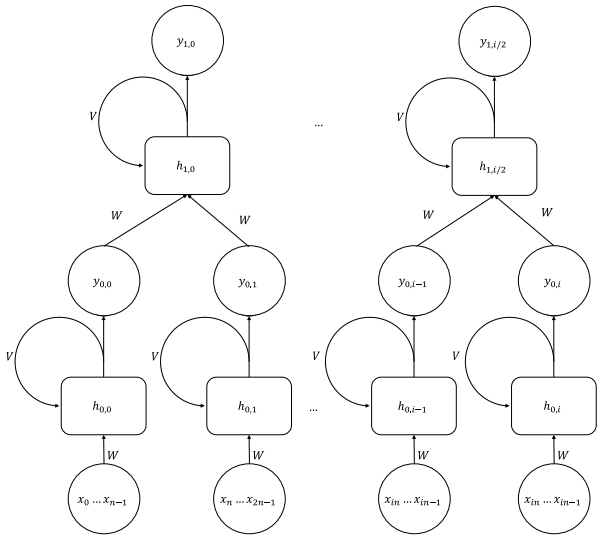


Figure 7.5: Hierarchical RNN Blocks architecture. We stack the output of multiple blocks (2 in this figure) and run them through another block.

to perform the combination of block outputs. Let n be the length of subsequences and l the length of the input sequence. In this architecture, we gather the output of up to n adjacent blocks and stack them together, forming a new sequence. An RNN block once again processes this new sequence; see Fig. 7.5. This arranges the blocks into layers. Each layer contains blocks at the same depth of computation. The number of blocks in each layer decreases until we reach a single block, giving the network a funnel-like shape. The first layer consists of l/n blocks, and the second layer consists of $\lceil l/n^2 \rceil$ blocks. We can generalize this to the number of blocks in the k-th layer is $\lceil l/n^k \rceil$. For simplicity, we constrain l and n to be powers of 2. This constraint is not strictly required; however, it simplifies the shape and split computation.

On encrypted data, the most essential characteristic of the network is the multiplicative depth. To analyze the depth of an RNN or an RNN block, we can unroll it. Once unrolled, it

essentially becomes a stack of fully connected layers. A vanilla RNN, on sequence of length l, turns into a stack of l fully connected layers. When using blocks with a subsequence length of n, each block unrolls into n fully connected layers. Let d_r be the depth of one of the unrolled layers, including activation functions. The depth of a vanilla RNN is then $d_r l$. The approaches presented in Chapter 6 and Section 7.1 have a depth of $d_r n + d_f + 1$ and $d_r n + d_f + 2$, respectively, where d_f is the depth is of a fully connected layer with square activation. In the hierarchical model, each layer of blocks has a depth of at most $d_r n$. We can compute the depth of the hierarchical model using the Algorithm 7.

Algorithm 7 Computing the depth of a hierarchical RNN blocks model

Inputs: Input sequence length l, subsequence length n, depth of a timestep in a block d_r
Outputs: Depth of the RNN blocks model d_h

1: $d_h := 1$ ▷ 1 for the output neuron
2: **while** $l > 1$ **do**
3: **if** $l \geq n$ **then**
4: $d_h := d_h + n d_r$
5: **else**
6: $d_h := d_h + l d_r$
7: **end if**
8: $l := l/n$
9: **end while**

We can easily see that the depth of the hierarchical models is higher than the depth of models with averaging or contamination. However, it is also lower than the depth of a vanilla RNN by at least a factor of two.

7.2.1 Evaluation

We now evaluate the performance of hierarchical RNN blocks. We use the same setup as in Section 7.1.1 for the averaging models. The length of the input sequence is 64. And we use

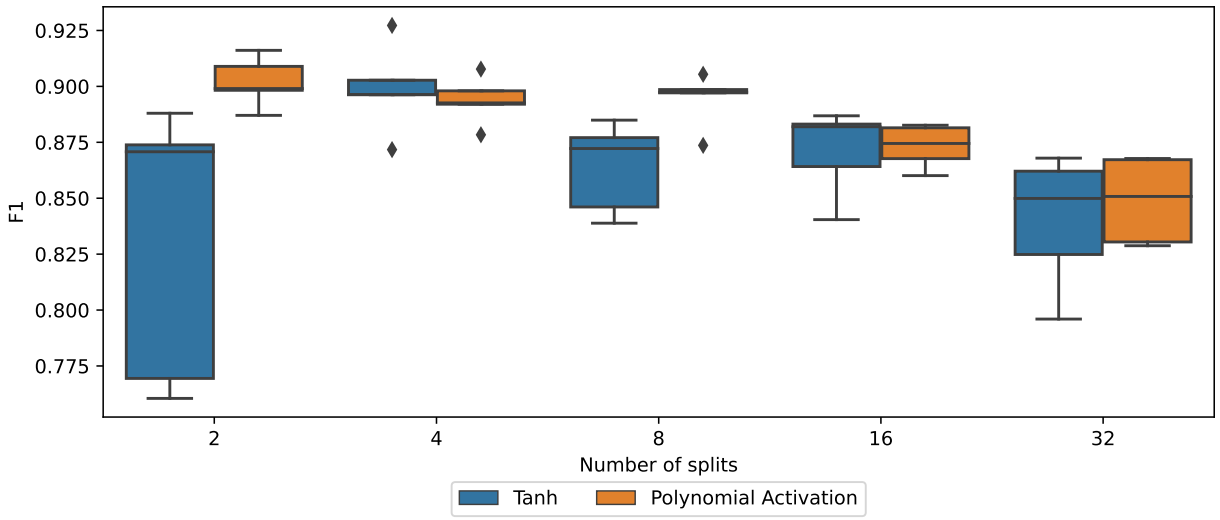


Figure 7.6: Result of 5-fold cross-validation for hierarchical RNN blocks using different numbers of splits and comparing the HE-friendly model (PA) to the non-HE-friendly model (Tanh).

128 units in the RNN blocks. The blocks in the first layer share the weights with each other, and the blocks we use for combining the outputs share their weights. However, we leave out 64 splits since this results in a subsequence length of one. The architecture needs at least a subsequence length of two to combine the outputs of at least two blocks as the input for the next layer. We use subsequence lengths of 2, 4, 8, 16, and 32, which results in 32, 16, 8, 4, and 2 splits respectively. We again use the same 5-fold cross-validation we used earlier, with the same folds.

Table 7.2 and Fig. 7.6 show the result of the experiments. We observe that the models using polynomial activations consistently have a higher average F1 score than the Tanh models with the same number of splits. We again perform a Mann-Whitney U test to establish the statistical significance of the performance differences. The results are shown in Fig. 7.7. We can see that there is no statistically significant difference in the performance of

Table 7.2: Average F1 score for 5-fold cross-validation of the hierarchical models with Polynomial Activation and Tanh activation, and a varying number of splits. The length of a subsequence is 64/#Splits. Performed on the Amazon Movie Review Data. 0 splits indicate the baseline RNN model that is not using blocks.

# Splits	Activation	F1
0	Polynomial Activation	0.874
	Tanh	0.912
2	Polynomial Activation	0.902
	Tanh	0.833
4	Polynomial Activation	0.894
	Tanh	0.899
8	Polynomial Activation	0.894
	Tanh	0.864
16	Polynomial Activation	0.873
	Tanh	0.871
32	Polynomial Activation	0.849
	Tanh	0.840

the models using polynomial activation with 2, 4, and 8 splits compared to the Tanh baseline. The models with 16 splits perform similarly to each other regardless of the activation function, and so do the models with 32 splits. The polynomial activation models with 2 and 8 splits show a significant performance improvement over their Tanh counterparts. At the same time, the models with 4 splits perform very similarly. This shows that this architecture can compensate for some of the temporal dependency loss we saw earlier and achieve close to baseline performance. Additionally, the results suggest that using polynomial activations does not harm the performance of the model since polynomials perform at least as well as Tanh with the same number of splits. However, we still need to compare all three strategies to each other, which we will do in the next section.

The hierarchical architecture shows near baseline performance; however, it exhibits higher multiplicative depths. Using Algorithm 7, we can compute the depth factors d to be 13, 13, 17, 21, and 34. To get the actual multiplicative depth of the model, we calculate dd_r.

We again use the activation function shown in Eq. 7.1. This gives us $d_r = 3$, leading to a total multiplicative of the models between 39 and 66. These depths are too high to be practical without bootstrapping; however, it would reduce the number of required bootstrapping operations. If we take Jang et al. [73] as a guideline and estimate that we need to perform bootstrapping every four layers (note: these are unrolled layers, not blocks). This gives between 2 and 8 bootstrapping operations depending on the split length. In a vanilla RNN with the same activation function and input sequence of length 64, we would need 16 bootstrapping operations.

7.3 Comparing Architectures

In this and the previous chapter, we presented three variants of the RNN blocks architecture: concatenation, averaging, and hierarchical. Each one has its strengths and drawbacks. Assuming the same number of splits, concatenation has the lowest multiplicative depth, averaging requires one more multiplication than concatenation, and the hierarchical structure requires the most multiplications. Theoretically, averaging should be faster and require less memory than concatenation since the last hidden layer has fewer connections, and the number of objects we need to keep in memory is significantly reduced. The hierarchical approach is, in all likelihood, the slowest since we need to incorporate bootstrapping. In terms of memory, it has a similar profile to concatenation if implemented naively. We can theoretically reduce the memory footprint by computing the network in a dept first approach. This should lead to a lower number of ciphertexts that we need to keep since each block reduces

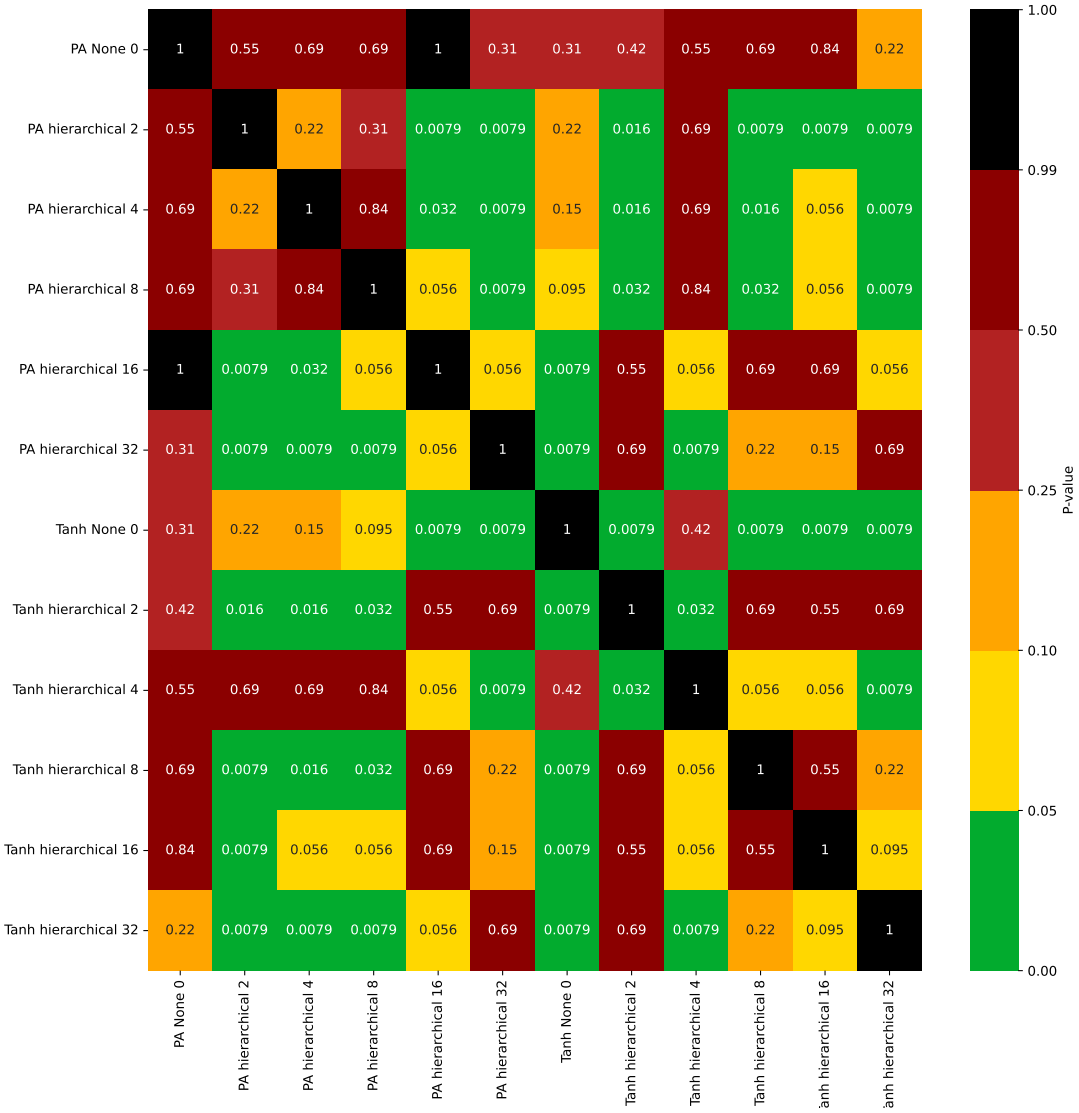
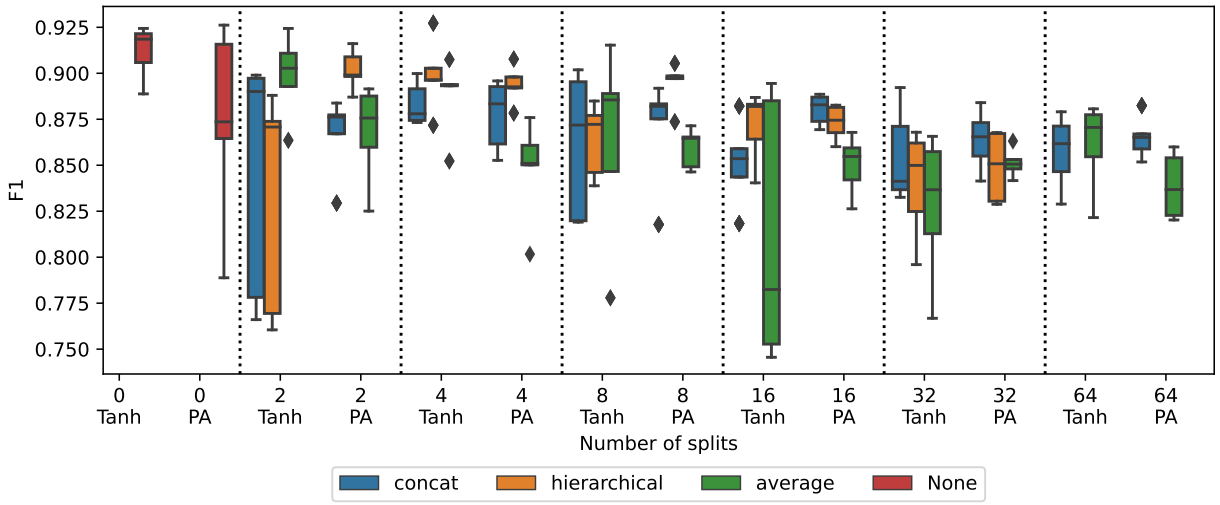


Figure 7.7: P-values computed by the Mann-Whitney U test on the F1 scores of 5-fold cross-validation. Computed for all model pairs using the hierarchical blocks strategy and the two baselines PA None 0 and Tanh None 0.

Figure 7.8: Performance comparison of all three RNN block architectures, concatenation (concat), average, and hierarchal. None indicates the baseline vanilla RNN with 0 splits. Tanh and Polynomial Activation (PA) indicate the activation function used. Dotted lines are purely visual dividers for clarity.



a subsequence into a single output.

The resource requirements of the model are not the only important factor when selecting a model. Perhaps more important is the model performance. We now compare the models presented in Chapter 6 and Sections 7.1 and 7.2 with each other in terms of performance. For all models, we perform 5-fold cross-validation on the Amazon Movie Review dataset [171], and use the same preprocessing described in Chapter 6. Table 7.3 and Fig. 7.8 show the results of the experiments.

We again perform a Mann-Whintey U test to establish the statistical significance of the difference in performance. We perform pairwise for all models with polynomial activation and the baseline models. Fig. 7.9, 7.10, and 7.11 show the result of the tests. By average F1 score, the best-performing model is the vanilla RNN with Tanh activation. However,

Table 7.3: Performance comparison of all combination strategies depending on the number of splits. F1 is the average F1 score on the test fold for 5-fold cross-validation. The highest average F1 score of a given number of splits is bold.

# Splits	Combination	Activation	F1
0	None	Polynomial Activation	0.874
		Tanh	**0.912**
2	average	Polynomial Activation	0.868
		Tanh	0.899
	concat	Polynomial Activation	0.867
		Tanh	0.846
	hierarchical	Polynomial Activation	**0.902**
		Tanh	0.833
4	average	Polynomial Activation	0.848
		Tanh	0.888
	concat	Polynomial Activation	0.877
		Tanh	0.883
	hierarchical	Polynomial Activation	0.894
		Tanh	**0.899**
8	average	Polynomial Activation	0.859
		Tanh	0.863
	concat	Polynomial Activation	0.870
		Tanh	0.862
	hierarchical	Polynomial Activation	**0.894**
		Tanh	0.864
16	average	Polynomial Activation	0.850
		Tanh	0.812
	concat	Polynomial Activation	**0.880**
		Tanh	0.851
	hierarchical	Polynomial Activation	0.873
		Tanh	0.871
32	average	Polynomial Activation	0.851
		Tanh	0.828
	concat	Polynomial Activation	**0.864**
		Tanh	0.855
	hierarchical	Polynomial Activation	0.849
		Tanh	0.840
64	average	Polynomial Activation	0.839
		Tanh	0.861
	concat	Polynomial Activation	**0.865**
		Tanh	0.857

as shown earlier, the hierarchical models with polynomial activation with 2, 4, and 8 splits show no statistically significant difference in performance. When we compare the difference between the hierarchical models and the averaging models, we find that the hierarchical models perform significantly better, up to 16 splits. At 32 splits, there is no significant difference between the performance. We further find that averaging with 2 splits performs

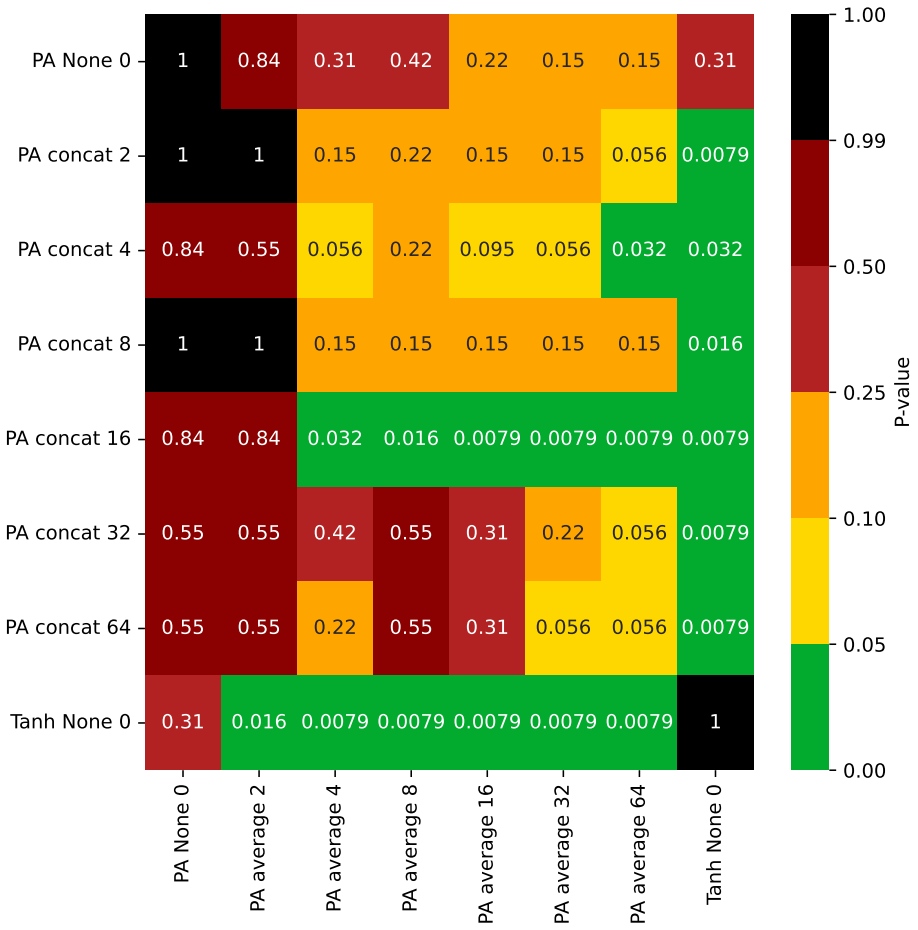


Figure 7.9: P-values computed by the Mann-Whitney U test on the F1 scores of 5-fold cross-validation. Computed for all model pairs using the concatenation and averaging strategy and the two baselines PA None 0 and Tanh None 0.

as well as the hierarchical model with 16 splits.

When comparing the hierarchical models with the concatenation models, we see a slightly different picture. The hierarchical model performs significantly better only with 2 splits. In all other cases, with the same number of splits, there is no significant difference in performance. In comparison, the hierarchical models outperform the averaging model in more cases.

Lastly, we compare the averaging and concatenation models. Interestingly, there is no significant difference in performance with the same number of splits, except for 16 splits. Here, the concatenation performs better. As the number of splits increases, the concatenation models can retain performance the best. The hierarchical models outperform them when the number of splits is low; however, they cannot keep this performance as the splits increase. We hypothesize that the concatenation layer retains the most temporal dependencies of the architecture presented here.

7.4 Summary and Discussion

In this chapter, we presented two additional architectures that use the RNN blocks idea. We analyzed their performance, advantages, and drawbacks and compared them with each other. If computational cost is not essential, hierarchical RNN blocks can offer a good theoretical trade-off of depth and performance. They outperform other architecture at a small number of splits and have no significant drop in performance compared to the baseline. However, they require significantly fewer bootstrapping operations than simply using a vanilla RNN. They might prove to be more beneficial on longer input sequences. When we can not use bootstrapping, concatenation offers a decent performance-depth tradeoff at a larger number of splits.

While we evaluate the architectures presented here on vanilla RNNs, they are not limited to those. In fact, we could substitute those for other layers like LSTMs, GRUs, or one-dimensional convolutions. It would be reasonable to suspect that we can see similar depth

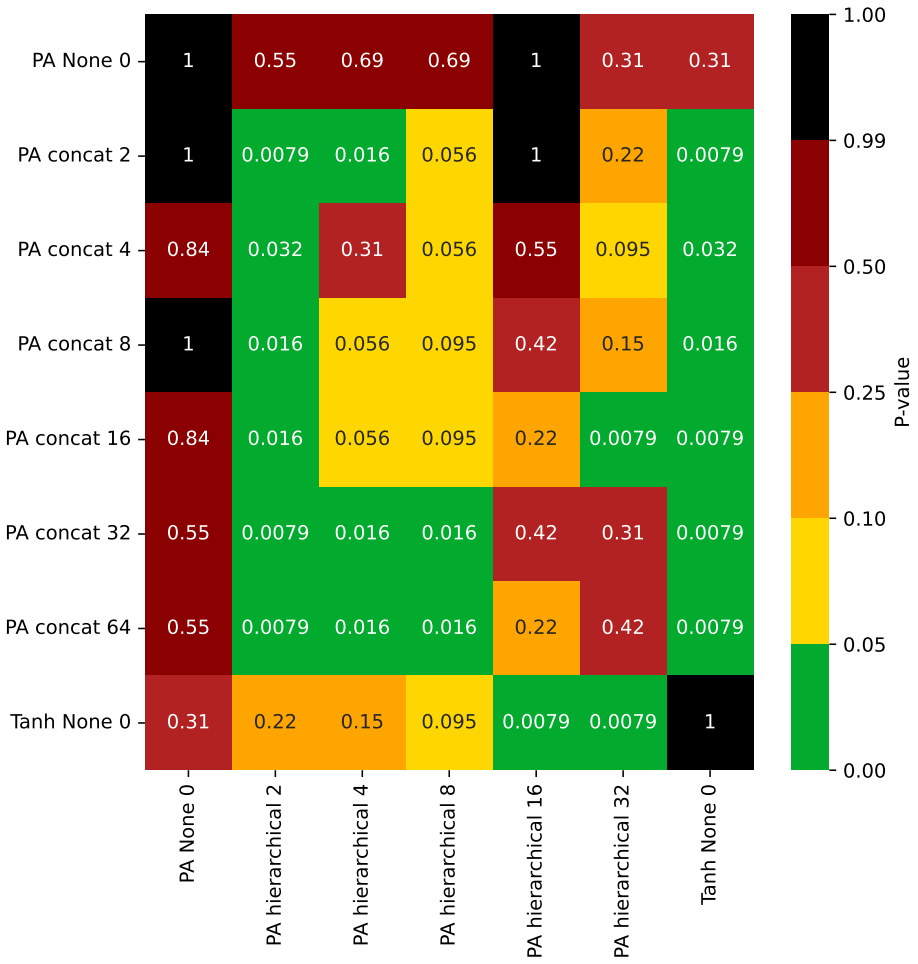


Figure 7.10: P-values computed by the Mann-Whitney U test on the F1 scores of 5-fold cross-validation. Computed for all model pairs using the concatenation and hierarchical strategy and the two baselines PA None 0 and Tanh None 0.

reduction using those layers.

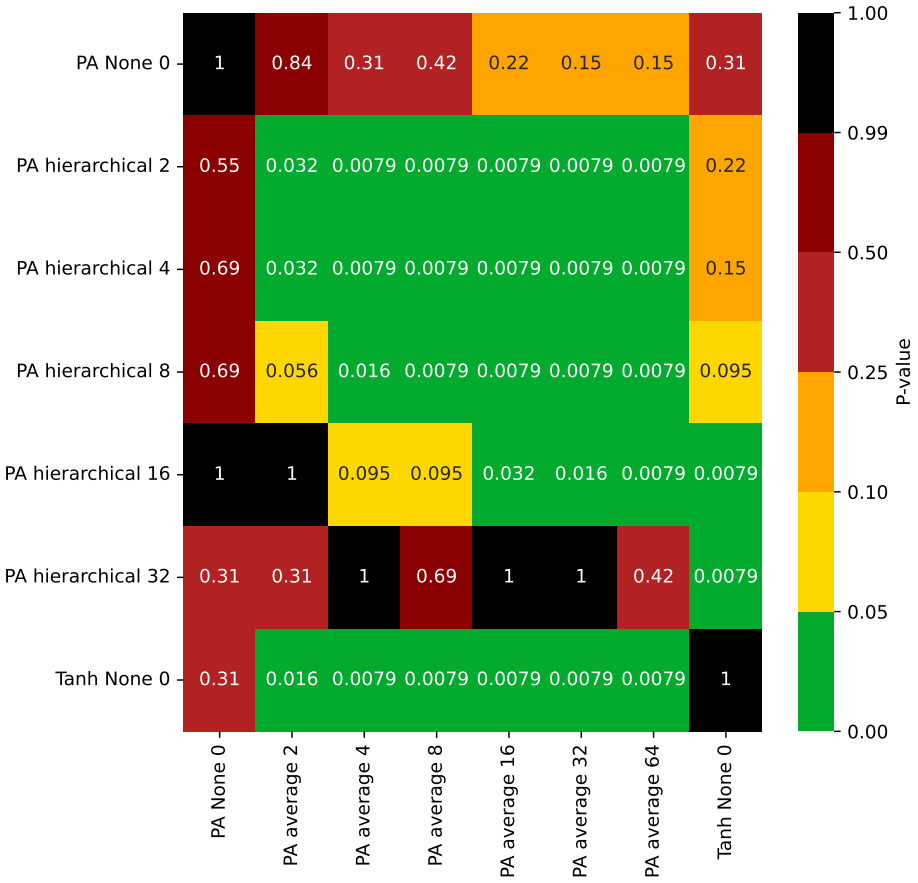


Figure 7.11: P-values computed by the Mann-Whitney U test on the F1 scores of 5-fold cross-validation. Computed for all model pairs using the averaging and hierarchical strategy and the two baselines PA None 0 and Tanh None 0.

CHAPTER 8
CONCLUSION AND FUTURE WORK

8.1 Conclusion

In this dissertation, we investigate issues of privacy-preserving neural networks with HE. To facilitate our experiments, we implement an optimizing compiler based on XLA that transforms TensorFlow models into HE operations. The compiler performs various optimizations on the graph, such as polynomial detection, efficient polynomial evaluation, and precomputation of operations with known inputs. However, foremost, it reduces the programming overhead of running models on encrypted data by providing a simple Python interface. Additionally, it provides us the access we need to the low-level functions and implementations we need to implement our research. Other closed-source tools, like HELayer, do not provide their source code, making it impossible for us to extend them.

One of the issues we investigate in-depth is the memory overhead of batch-packed convolutions on encrypted data. We introduce a schedule representation that allows us to analyze the convolution on a basic operation level. Along with the representation, we present an algorithm to execute a schedule in parallel using multiple execution threads. We propose multiple schedules that swap ciphertexts and plaintexts in and out of memory to optimize memory usage while minimizing execution time. Using these optimized schedules, we can greatly reduce the memory requirements at little runtime cost.

The other issue we investigate is RNNs over encrypted data. When using HE, the depth of RNNs poses a problem. We present and evaluate multiple strategies for dealing with

the depth. One of the strategies relies on communication between the client and server to reset the noise level during computation; however, this can expose the internals of the server-side network to the client. A malicious client could use this information to steal the model weights. To mitigate this, we present an architecture that uses recurrent elements, which we call RNN blocks, to eliminate communication. It does this by reducing the depth of the network. We can reduce the network depth by splitting the input sequence into subsequences and processing the subsequences with RNN blocks. We need to combine the output of the blocks for further processing by the network. We analyze three combining approaches: concatenation, averaging, and a hierarchical approach. In our experiments, we find that concatenation works best for a large number of splits and short subsequences, and the hierarchical approach works best for very few splits. However, the hierarchical approach produces deeper models than concatenation and averaging and, therefore, needs bootstrapping on encrypted data. Despite this, the hierarchical models are only half as deep as vanilla RNN would be. The averaging approach likely performs better when we work with input sequences of varying lengths since its structure does not rely on the exact number of splits.

8.2 Future Work

In the future, we want to continue the research problems addressed in this dissertation. Focusing on addressing the resource requirements of HE-based PPML and privacy-preserving neural networks for sequence processing.

8.2.1 Future work 1: Addressing the resource overhead

The work presented here relies on batch-packed SIMD processing. However, other ciphertext-packing approaches can reduce latency and memory overhead further by introducing improved algorithms that rely on rotation. During graph analysis, we can also determine the best ciphertext-packing. The naive and batched packing suffers from large computational overhead and high inference latency but offers excellent throughput. More efficient packing techniques like the ones proposed by Jiang et al. [61] and Brutzkus et al. [63] use fewer ciphertexts, reducing the inference latency. However, selecting the most efficient packing depends on the model. Determining the best layout for a given model is still an open problem. Furthermore, it is possible that for a given model, none of the published layouts is the most efficient. An exhaustive search of the space of proposed layouts can be feasible. However, searching all possible layouts is probably intractable. We need to investigate strategies for searching the layout space. A possible avenue is inspired by the recent success of deep reinforcement learning for circuit design [175] and improved matrix multiplication [176]. We want to investigate their potential application to cipher-packing design for HE.

8.2.2 Future work 2: Privacy-preserving Neural Networks for Sequence Processing

In this dissertation, we exclusively use vanilla RNNs for sequence processing. However, vanilla RNNs have been replaced mainly by other recurrent layers like LSTMs or GRUs. These do not suffer from vanishing gradients during training the same way vanilla RNNs do. However, they are computationally more complex. With improvements in ciphertext-

packing and bootstrapping, some prior work realizes GRUs over encrypted data [162]. It seems likely the number of bootstrapping operations can be reduced by combining the RNN block architecture presented here with layers like LSTMs or GRUs. We want to investigate how RNN blocks interact with LSTMs and GRUs.

Furthermore, especially in language processing, RNNs are often replaced with Transformers[177] nowadays. Little work is dedicated to privacy-preserving transformers using HE [178]. Here, the authors make extensive modifications to the transformer architecture by dropping some parts of the model and replacing others with approximations. An additional issue is that the inference relies on server-client interaction. We want to investigate strategies for eliminating this interaction and making inference non-interactive.

REFERENCES

[1] Robert Podschwadt et al. "A Survey of Deep Learning Architectures for Privacy-Preserving Machine Learning With Fully Homomorphic Encryption". In: *IEEE Access* 10 (2022). Conference Name: IEEE Access, pp. 117477–117500. ISSN: 2169-3536. DOI: 10.1109/ACCESS.2022.3219049.

[2] Robert Podschwadt and Daniel Takabi. "Classification of Encrypted Word Embeddings using Recurrent Neural Networks." In: *PrivateNLP@ WSDM*. 2020, pp. 27–31.

[3] Robert Podschwadt and Daniel Takabi. "Non-interactive Privacy Preserving Recurrent Neural Network Prediction with Homomorphic Encryption". In: *2021 IEEE 14th International Conference on Cloud Computing (CLOUD)*. IEEE, 2021, pp. 65–70.

[4] "Perspective — Alexa has been eavesdropping on you this whole time". en-US. In: *Washington Post* (May 2019). ISSN: 0190-8286. URL: https://www.washingtonpost.com/technology/2019/05/06/alexa-has-been-eavesdropping-you-this-whole-time/ (visited on 09/15/2022).

[5] Rina Torchinsky. "How period tracking apps and data privacy fit into a post-Roe v. Wade climate". In: *NPR* (June 24, 2022). URL: https://www.npr.org/2022/05/10/1097482967/roe-v-wade-supreme-court-abortion-period-apps (visited on 10/12/2022).

[6] By Rina Chandran and Diana Baptista. "Analysis: After Roe v. Wade, healthcare data privacy fears grow worldwide". In: *Reuters* (July 12, 2022). URL: https://www.reuters.com/legal/litigation/after-roe-v-wade-healthcare-data-privacy-fears-grow-worldwide-2022-07-12/ (visited on 10/12/2022).

[7] Cynthia Dwork and Aaron Roth. "The algorithmic foundations of differential privacy." In: *Foundations and Trends in Theoretical Computer Science* 9.3-4 (2014), pp. 211–407.

[8] David Evans, Vladimir Kolesnikov, and Mike Rosulek. "A pragmatic introduction to secure multi-party computation". In: *Foundations and Trends® in Privacy and Security* 2.2-3 (2018). Publisher: Now Publishers, Inc., pp. 70–246.

[9] Amit Sahai and Brent Waters. "Fuzzy identity-based encryption". In: *Annual international conference on the theory and applications of cryptographic techniques*. Springer, 2005, pp. 457–473.

[10] Dan Boneh, Amit Sahai, and Brent Waters. "Functional encryption: Definitions and challenges". In: *Theory of Cryptography Conference*. Springer, 2011, pp. 253–273.

[11] Martin Abadi et al. "Deep learning with differential privacy". In: *Proceedings of the 2016 ACM SIGSAC conference on computer and communications security*. 2016, pp. 308–318.

[12] Nicolas Papernot et al. "Scalable private learning with pate". In: *arXiv preprint arXiv:1802.08908* (2018).

[13] Andrew C. Yao. "Protocols for secure computations". In: *23rd annual symposium on foundations of computer science (sfcs 1982)*. IEEE, 1982, pp. 160–164.

[14] Oded Goldreich, Silvio Micali, and Avi Wigderson. "How to play any mental game, or a completeness theorem for protocols with honest majority". In: *Providing Sound Foundations for Cryptography: On the Work of Shafi Goldwasser and Silvio Micali*. 2019, pp. 307–328.

[15] Michael O. Rabin. "How to exchange secrets with oblivious transfer". In: *Cryptology ePrint Archive* (2005).

[16] Woo-Seok Choi et al. "Impala: Low-Latency, Communication-Efficient Private Deep Learning Inference". In: *arXiv preprint arXiv:2205.06437* (2022).

[17] Shaohua Li et al. "FALCON: A Fourier Transform Based Approach for Fast and Secure Convolutional Neural Network Predictions". en. In: *2020 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*. Seattle, WA, USA: IEEE, June 2020, pp. 8702–8711. ISBN: 978-1-72817-168-5. DOI: 10.1109/CVPR42600.2020.00873. URL: https://ieeexplore.ieee.org/document/9156980/ (visited on 02/09/2021).

[18] Jian Liu et al. "Oblivious neural network predictions via minionn transformations". In: *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*. 2017, pp. 619–631.

[19] Donald Beaver. “Efficient multiparty protocols using circuit randomization”. In: *Annual International Cryptology Conference*. Springer, 1991, pp. 420–432.

[20] Donald Beaver. “Precomputing oblivious transfer”. In: *Annual International Cryptology Conference*. Springer, 1995, pp. 97–109.

[21] Florian Tramèr et al. “Stealing Machine Learning Models via Prediction ${$APIs$}$”. In: *25th USENIX security symposium (USENIX Security 16)*. 2016, pp. 601–618.

[22] Prajwal Panzade and Daniel Takabi. “Towards faster functional encryption for privacy-preserving machine learning”. In: *2021 Third IEEE International Conference on Trust, Privacy and Security in Intelligent Systems and Applications (TPS-ISA)*. IEEE, 2021, pp. 21–30.

[23] Théo Ryffel et al. “Partially encrypted deep learning using functional encryption”. In: *Advances in Neural Information Processing Systems* 32 (2019).

[24] Ilaria Chillotti et al. “Faster packed homomorphic operations and efficient circuit bootstrapping for TFHE”. In: *International Conference on the Theory and Application of Cryptology and Information Security*. Springer, 2017, pp. 377–408.

[25] Shruthi Gorantala et al. “A General Purpose Transpiler for Fully Homomorphic Encryption”. In: *arXiv preprint arXiv:2106.07893* (2021).

[26] Ehsan Hesamifard, Hassan Takabi, and Mehdi Ghasemi. “Cryptodl: Deep neural networks over encrypted data”. In: *arXiv preprint arXiv:1711.05189* (2017).

[27] Nathan Dowlin et al. “Cryptonets: Applying neural networks to encrypted data with high throughput and accuracy”. In: *International Conference on Machine Learning*. 2016, pp. 201–210.

[28] Joon-Woo Lee et al. “Privacy-Preserving Machine Learning With Fully Homomorphic Encryption for Deep Neural Network”. In: *IEEE Access* 10 (2022). Conference Name: IEEE Access, pp. 30039–30054. ISSN: 2169-3536. DOI: 10.1109/ACCESS.2022.3159694.

[29] Razvan Pascanu, Tomas Mikolov, and Yoshua Bengio. “Understanding the exploding gradient problem”. In: *CoRR* abs/1211.5063 (2012). arXiv: 1211.5063. URL: http://arxiv.org/abs/1211.5063.

[30] Nicolas Papernot. “Machine Learning at Scale with Differential Privacy in TensorFlow”. In: *2019 USENIX Conference on Privacy Engineering Practice and Respect (PEPR 19)*. 2019.

[31] Ashkan Yousefpour et al. “Opacus: User-Friendly Differential Privacy Library in PyTorch”. In: *arXiv preprint arXiv:2109.12298* (2021).

[32] Nishant Kumar et al. “CrypTFlow: Secure TensorFlow Inference”. In: *IEEE Symposium on Security and Privacy*. IEEE, May 2020. URL: https://www.microsoft.com/en-us/research/publication/cryptflow-secure-tensorflow-inference/.

[33] Brian Knott et al. *CrypTen: Secure Multi-Party Computation Meets Machine Learning*. Tech. rep. arXiv:2109.00984. arXiv:2109.00984 [cs] type: article. arXiv, Sept. 2021.

DOI: 10.48550/arXiv.2109.00984. URL: http://arxiv.org/abs/2109.00984 (visited on 05/24/2022).

[34] W. Diffie and M. Hellman. "New directions in cryptography". In: *IEEE Transactions on Information Theory* 22.6 (Nov. 1976). Conference Name: IEEE Transactions on Information Theory, pp. 644–654. ISSN: 1557-9654. DOI: 10.1109/TIT.1976.1055638.

[35] R. L. Rivest, A. Shamir, and L. Adleman. "A method for obtaining digital signatures and public-key cryptosystems". In: *Communications of the ACM* 21.2 (Feb. 1978), pp. 120–126. ISSN: 0001-0782. DOI: 10.1145/359340.359342. URL: https://doi.org/10.1145/359340.359342 (visited on 08/16/2022).

[36] Ronald L. Rivest, Len Adleman, and Michael L. Dertouzos. "On data banks and privacy homomorphisms". In: *Foundations of secure computation* 4.11 (1978). Publisher: Citeseer, pp. 169–180.

[37] Frederik Armknecht et al. "A Guide to Fully Homomorphic Encryption". en. In: (2015), p. 35.

[38] Pascal Paillier. "Public-key cryptosystems based on composite degree residuosity classes". In: *International conference on the theory and applications of cryptographic techniques*. Springer, 1999, pp. 223–238.

[39] Taher Elgamal. "A Public Key Cryptosystem and a Signature Scheme Based on Discrete Logarithms". en. In: *IEEE TRANSACTIONS ON INFORMATION THEORY* 4 (1985), p. 4.

[40] Zvika Brakerski and Vinod Vaikuntanathan. "Fully homomorphic encryption from ring-LWE and security for key dependent messages". In: *Annual cryptology conference*. Springer, 2011, pp. 505–524.

[41] Zvika Brakerski, Craig Gentry, and Vinod Vaikuntanathan. "(Leveled) fully homomorphic encryption without bootstrapping". In: *ACM Transactions on Computation Theory (TOCT)* 6.3 (2014). Publisher: ACM New York, NY, USA, pp. 1–36.

[42] Marten van Dijk et al. "Fully homomorphic encryption over the integers". In: *Annual international conference on the theory and applications of cryptographic techniques*. Springer, 2010, pp. 24–43.

[43] Craig Gentry. *A fully homomorphic encryption scheme*. Vol. 20. 9. Stanford university Stanford, 2009.

[44] Zvika Brakerski, Shafi Goldwasser, and Yael Tauman Kalai. "Black-box circular-secure encryption beyond affine functions". In: *Theory of Cryptography Conference*. Springer, 2011, pp. 201–218.

[45] Zvika Brakerski. "Fundamentals of fully homomorphic encryption". In: *Providing Sound Foundations for Cryptography: On the Work of Shafi Goldwasser and Silvio Micali*. 2019, pp. 543–563.

[46] Kyoohyung Han and Dohyeong Ki. "Better bootstrapping for approximate homomorphic encryption". In: *Cryptographers' Track at the RSA Conference*. Springer, 2020, pp. 364–390.

[47] Michael J. Flynn. "Some Computer Organizations and Their Effectiveness". In: *IEEE Transactions on Computers* C-21.9 (Sept. 1972). Conference Name: IEEE Transactions on Computers, pp. 948–960. ISSN: 1557-9956. DOI: 10.1109/TC.1972.5009071.

[48] Nigel P. Smart and Frederik Vercauteren. "Fully homomorphic SIMD operations". In: *Designs, codes and cryptography* 71.1 (2014). Publisher: Springer, pp. 57–81.

[49] Jung Hee Cheon et al. "Homomorphic encryption for arithmetic of approximate numbers". In: *International Conference on the Theory and Application of Cryptology and Information Security*. Springer, 2017, pp. 409–437.

[50] Vadim Lyubashevsky, Oded Regev, and Chris Peikert. "A Toolkit for Ring-LWE Cryptography". en. In: *Advances in Cryptology – EUROCRYPT 2013*. Ed. by David Hutchison et al. Vol. 7881. Series Title: Lecture Notes in Computer Science. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 35–54. ISBN: 978-3-642-38347-2 978-3-642-38348-9. DOI: 10.1007/978-3-642-38348-9_3. URL: http://link.springer.com/10.1007/978-3-642-38348-9_3 (visited on 10/16/2023).

[51] Vadim Lyubashevsky, Chris Peikert, and Oded Regev. "On Ideal Lattices and Learning with Errors over Rings". In: *Journal of the ACM* 60.6 (Nov. 2013), 43:1–43:35. ISSN: 0004-5411. DOI: 10.1145/2535925. URL: https://doi.org/10.1145/2535925 (visited on 08/09/2022).

[52] Junfeng Fan and Frederik Vercauteren. "Somewhat Practical Fully Homomorphic Encryption." In: *IACR Cryptol. ePrint Arch.* 2012 (2012). Publisher: Citeseer, p. 144.

[53] Martin Albrecht et al. *Homomorphic Encryption Security Standard*. Tech. rep. Backup Publisher: HomomorphicEncryption.org. Toronto, Canada, Nov. 2018. URL: HomomorphicEncryption.org.

[54] Martin R. Albrecht, Rachel Player, and Sam Scott. "On the concrete hardness of Learning with Errors". en. In: *Journal of Mathematical Cryptology* 9.3 (Oct. 2015), pp. 169–203. ISSN: 1862-2984, 1862-2976. DOI: 10.1515/jmc-2015-0016. URL: https://www.degruyter.com/document/doi/10.1515/jmc-2015-0016/html (visited on 10/19/2023).

[55] *Microsoft SEAL (release 3.6)*. Nov. 2020. URL: https://github.com/Microsoft/SEAL.

[56] Ahmad Al Badawi et al. "OpenFHE: Open-Source Fully Homomorphic Encryption Library". In: *Proceedings of the 10th Workshop on Encrypted Computing & Applied Homomorphic Cryptography*. WAHC'22. event-place: Los Angeles, CA, USA. New York, NY, USA: Association for Computing Machinery, 2022, pp. 53–63. DOI: 10.1145/3560827.3563379. URL: https://doi.org/10.1145/3560827.3563379.

[57] Jung Hee Cheon et al. "Bootstrapping for approximate homomorphic encryption". In: *Advances in Cryptology–EUROCRYPT 2018: 37th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Tel Aviv, Israel, April 29-May 3, 2018 Proceedings, Part I 37*. Springer, 2018, pp. 360–384.

[58] Qian Lou and Lei Jiang. "SHE: A Fast and Accurate Deep Neural Network for Encrypted Data". In: *Advances in Neural Information Processing Systems*. 2019, pp. 10035–10043.

[59] Edward Chou et al. "Faster cryptonets: Leveraging sparsity for real-world encrypted inference". In: *arXiv preprint arXiv:1811.09953* (2018).

[60] Aojun Zhou et al. "Incremental network quantization: Towards lossless cnns with low-precision weights". In: *arXiv preprint arXiv:1702.03044* (2017).

[61] Xiaoqian Jiang et al. "Secure Outsourced Matrix Computation and Application to Neural Networks". en. In: *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*. Toronto Canada: ACM, Oct. 2018, pp. 1209–1222. ISBN: 978-1-4503-5693-0. DOI: 10.1145/3243734.3243837. URL: https://dl.acm.org/doi/10.1145/3243734.3243837 (visited on 01/14/2021).

[62] Ahmad Al Badawi et al. "PrivFT: Private and Fast Text Classification With Homomorphic Encryption". In: *IEEE Access* 8 (2020). Conference Name: IEEE Access, pp. 226544–226556. ISSN: 2169-3536. DOI: 10.1109/ACCESS.2020.3045465.

[63] Alon Brutzkus, Ran Gilad-Bachrach, and Oren Elisha. "Low latency privacy preserving inference". In: *International Conference on Machine Learning*. PMLR, 2019, pp. 812–821.

[64] Roshan Dathathri et al. "CHET: an optimizing compiler for fully-homomorphic neural-network inferencing". In: *Proceedings of the 40th ACM SIGPLAN Conference*

on Programming Language Design and Implementation. PLDI 2019. New York, NY, USA: Association for Computing Machinery, June 2019, pp. 142–156. ISBN: 978-1-4503-6712-7. DOI: 10.1145/3314221.3314628. URL: https://doi.org/10.1145/3314221.3314628 (visited on 11/19/2020).

[65] Kentaro Mihara et al. "Neural Network Training With Homomorphic Encryption". In: *arXiv preprint arXiv:2012.13552* (2020).

[66] Hervé Chabanne et al. "Privacy-Preserving Classification on Deep Neural Network." In: *IACR Cryptol. ePrint Arch.* 2017 (2017), p. 35.

[67] A. Al Badawi et al. "Towards the AlexNet Moment for Homomorphic Encryption: HCNN, the First Homomorphic CNN on Encrypted Data with GPUs". In: *IEEE Transactions on Emerging Topics in Computing* (2020). Conference Name: IEEE Transactions on Emerging Topics in Computing, pp. 1–1. ISSN: 2168-6750. DOI: 10.1109/TETC.2020.3014636.

[68] Shi-Xiong Zhang, Yifan Gong, and Dong Yu. "Encrypted speech recognition using deep polynomial networks". In: *ICASSP 2019-2019 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*. IEEE, 2019, pp. 5691–5695.

[69] Fabian Boemer et al. "ngraph-he: A graph compiler for deep learning on homomorphically encrypted data". In: *Proceedings of the 16th ACM International Conference on Computing Frontiers*. 2019, pp. 3–13.

[70] Maya Bakshi and Mark Last. "Cryptornn-privacy-preserving recurrent neural networks using homomorphic encryption". In: *International Symposium on Cyber Security Cryptography and Machine Learning*. Springer, 2020, pp. 245–253.

[71] Sepp Hochreiter and Jürgen Schmidhuber. "Long short-term memory". In: *Neural computation* 9.8 (1997). Publisher: MIT Press, pp. 1735–1780.

[72] Kyunghyun Cho et al. "On the properties of neural machine translation: Encoder-decoder approaches". In: *arXiv preprint arXiv:1409.1259* (2014).

[73] Jaehee Jang et al. "Privacy-Preserving Deep Sequential Model with Matrix Homomorphic Encryption". In: *Proceedings of the 2022 ACM on Asia Conference on Computer and Communications Security*. ASIA CCS '22. New York, NY, USA: Association for Computing Machinery, May 2022, pp. 377–391. ISBN: 978-1-4503-9140-5. DOI: 10.1145/3488932.3523253. URL: https://doi.org/10.1145/3488932.3523253 (visited on 08/12/2022).

[74] Armand Joulin et al. "Bag of tricks for efficient text classification". In: *arXiv preprint arXiv:1607.01759* (2016).

[75] Florian Bourse et al. "Fast homomorphic evaluation of deep discretized neural networks". In: *Annual International Cryptology Conference*. Springer, 2018, pp. 483–512.

[76] Marshall H. Stone. "The generalized Weierstrass approximation theorem". In: *Mathematics Magazine* 21.5 (1948). Publisher: JSTOR, pp. 237–254.

[77] Theodore J. Rivlin. *Chebyshev polynomials*. Courier Dover Publications, 2020.

[78] Eunsang Lee et al. “Minimax Approximation of Sign Function by Composite Polynomial for Homomorphic Comparison”. In: *IEEE Transactions on Dependable and Secure Computing* (2021). Conference Name: IEEE Transactions on Dependable and Secure Computing, pp. 1–1. ISSN: 1941-0018. DOI: 10.1109/TDSC.2021.3105111.

[79] Robert E. Goldschmidt. “Applications of division by convergence”. PhD Thesis. Massachusetts Institute of Technology, 1964.

[80] Günter Meinardus. *Approximation of functions: Theory and numerical methods*. Vol. 13. Springer Science & Business Media, 2012.

[81] K. Nandakumar et al. “Towards Deep Neural Network Training on Encrypted Data”. In: *2019 IEEE/CVF Conference on Computer Vision and Pattern Recognition Workshops (CVPRW)*. ISSN: 2160-7516. June 2019, pp. 40–48. DOI: 10.1109/CVPRW.2019.00011.

[82] Jack LH Crawford et al. “Doing real work with FHE: the case of logistic regression”. In: *Proceedings of the 6th Workshop on Encrypted Computing & Applied Homomorphic Cryptography*. 2018, pp. 1–12.

[83] Ehsan Hesamifard, Hassan Takabi, and Mehdi Ghasemi. “CryptoDL: Towards Deep Learning over Encrypted Data”. In: *Annual Computer Security Applications Conference (ACSAC)*. 2016.

[84] Ehud Aharoni et al. “HeLayers: A Tile Tensors Framework for Large Neural Networks on Encrypted Data”. In: *Proceedings on Privacy Enhancing Technologies* 2023.1 (Jan.

2023). arXiv:2011.01805 [cs], pp. 325–342. ISSN: 2299-0984. DOI: 10.56553/popets-2023-0020. URL: http://arxiv.org/abs/2011.01805 (visited on 09/27/2023).

[85] Alexander Viand, Patrick Jattke, and Anwar Hithnawi. "Sok: Fully homomorphic encryption compilers". In: *2021 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2021, pp. 1092–1108.

[86] Jeffrey L. Elman. "Finding structure in time". In: *Cognitive science* 14.2 (1990). Publisher: Wiley Online Library, pp. 179–211.

[87] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. http://www.deeplearningbook.org. MIT Press, 2016.

[88] INSPIRE Lab. *CryptoDL*. original-date: 2020-01-15T23:26:29Z. Aug. 2022. URL: https://github.com/inspire-lab/CryptoDL (visited on 09/12/2022).

[89] Shai Halevi and Victor Shoup. "Design and implementation of HElib: a homomorphic encryption library". en. In: (), p. 42.

[90] Yann LeCun et al. "Gradient-based learning applied to document recognition". In: *Proceedings of the IEEE* 86.11 (1998). Publisher: Ieee, pp. 2278–2324.

[91] *Homomorphic Encryption Standardization – An Open Industry / Government / Academic Consortium to Advance Secure Computation*. en-US. URL: https://homomorphicencryption.org/ (visited on 04/26/2021).

[92] F. Pedregosa et al. "Scikit-learn: Machine Learning in Python". In: *Journal of Machine Learning Research* 12 (2011), pp. 2825–2830.

[93] Martín Abadi et al. "${$TensorFlow$}$: A System for ${$Large-Scale$}$ Machine Learning". In: *12th USENIX symposium on operating systems design and implementation (OSDI 16)*. 2016, pp. 265–283.

[94] Adam Paszke et al. "Pytorch: An imperative style, high-performance deep learning library". In: *Advances in neural information processing systems* 32 (2019).

[95] Eunsang Lee et al. "Low-Complexity Deep Convolutional Neural Networks on Fully Homomorphic Encryption Using Multiplexed Parallel Convolutions". In: *International Conference on Machine Learning*. PMLR, 2022, pp. 12403–12422.

[96] Pedro Geraldo MR Alves, Jheyne N. Ortiz, and Diego F. Aranha. "Faster Homomorphic Encryption over GPGPUs via hierarchical DGT". In: ().

[97] Kaiming Nan et al. "Deep model compression for mobile platforms: A survey". In: *Tsinghua Science and Technology* 24.6 (2019). Publisher: TUP, pp. 677–693.

[98] Yihui He et al. "Amc: Automl for model compression and acceleration on mobile devices". In: *Proceedings of the European conference on computer vision (ECCV)*. 2018, pp. 784–800.

[99] Ehud Aharoni et al. "HE-PEx: Efficient Machine Learning under Homomorphic Encryption using Pruning, Permutation and Expansion". In: *arXiv preprint arXiv:2207.03384* (2022).

[100] Inbar Helbitz and Shai Avidan. "Reducing ReLU Count for Privacy-Preserving CNN Speedup". en. In: *arXiv:2101.11835 [cs]* (Jan. 2021). arXiv: 2101.11835. URL: http://arxiv.org/abs/2101.11835 (visited on 02/01/2021).

[101] Mika Juuti et al. "PRADA: Protecting Against DNN Model Stealing Attacks". In: *2019 IEEE European Symposium on Security and Privacy (EuroS&P)*. June 2019, pp. 512–527. DOI: 10.1109/EuroSP.2019.00044.

[102] Jeffrey Pennington, Richard Socher, and Christopher D. Manning. "Glove: Global vectors for word representation". In: *Proceedings of the 2014 conference on empirical methods in natural language processing (EMNLP)*. 2014, pp. 1532–1543.

[103] Eric Crockett. "Simply safe lattice cryptography". PhD Thesis. Georgia Institute of Technology, 2017.

[104] Sergiu Carpov, Paul Dubrulle, and Renaud Sirdey. "Armadillo: a compilation chain for privacy preserving applications". In: *Proceedings of the 3rd International Workshop on Security in Cloud Computing*. 2015, pp. 13–19.

[105] Eduardo Chielle et al. "E3: A framework for compiling C++ programs with encrypted operands". In: *Cryptology ePrint Archive* (2018).

[106] Alexander Viand and Hossein Shafagh. "Marble: Making fully homomorphic encryption accessible to all". In: *Proceedings of the 6th Workshop on Encrypted Computing & Applied Homomorphic Cryptography*. 2018, pp. 49–60.

[107] David W. Archer et al. "Ramparts: A programmer-friendly system for building homomorphic encryption applications". In: *Proceedings of the 7th ACM Workshop on Encrypted Computing & Applied Homomorphic Cryptography*. 2019, pp. 57–68.

[108] Fabian Boemer et al. "ngraph-he2: A high-throughput framework for neural network inference on encrypted data". In: *Proceedings of the 7th ACM Workshop on Encrypted Computing & Applied Homomorphic Cryptography*. 2019, pp. 45–56.

[109] Tim van Elsloo, Giorgio Patrini, and Hamish Ivey-Law. "SEALion: A framework for neural network inference on encrypted data". In: *arXiv preprint arXiv:1904.12840* (2019).

[110] Ayoub Benaissa et al. *TenSEAL: A Library for Encrypted Tensor Operations Using Homomorphic Encryption*. Tech. rep. arXiv:2104.03152. arXiv:2104.03152 [cs] type: article. arXiv, Apr. 2021. DOI: 10.48550/arXiv.2104.03152. URL: http://arxiv.org/abs/2104.03152 (visited on 05/24/2022).

[111] Tensorflow. https://www.tensorflow.org/xla. Accessed: 2023-10-20.

[112] Kim Laine and Rachel Player. "Simple encrypted arithmetic library-seal (v2. 0)". In: *Technical report, Technical report* (2016).

[113] Chris Lattner and Vikram Adve. "LLVM: A compilation framework for lifelong program analysis & transformation". In: *International Symposium on Code Generation and Optimization, 2004. CGO 2004.* IEEE, 2004, pp. 75–86.

[114] Yann LeCunn, Corinna Cortes, and Christopher JC Burges. *The MNIST database of handwritten digits*. 1998. URL: http://yann.lecun.com/exdb/mnist/.

[115] Alex Krizhevsky and Geoffrey Hinton. "Learning multiple layers of features from tiny images". In: (2009). Publisher: Toronto, ON, Canada.

[116] Apple Inc. *Siri - Apple*. https://www.apple.com/siri/. Accessed: 2023-10-17.

[117] Inc Amazon.com. *Amazon Alexa Voice AI, Alexa Developer Offical Site*. https://developer.amazon.com/en-US/alexa. Accessed: 2023-10-17.

[118] Inc Google. *Google Assitant, your own personal Google*. https://assistant.google.com/. Accessed: 2023-10-17.

[119] Inc Grammarly. *Grammarly: Free Writing AI Assistance*. https://www.grammarly.com/. Accessed: 2023-10-17.

[120] Inc Google. *Bard - Chat Based AI Tool from Google, Powered by PaLM2*. https://bard.google.com/. Accessed: 2023-10-17.

[121] OpenAI. *ChatGPT*. https://openai.com/chatgpt. Accessed: 2023-10-17.

[122] Steven E Dilsizian and Eliot L Siegel. "Artificial intelligence in medicine and cardiac imaging: harnessing big data and advanced computing to provide personalized medical diagnosis and treatment". In: *Current cardiology reports* 16 (2014), pp. 1–8.

[123] Amrit Kashyap et al. "A deep learning approach to estimating initial conditions of brain network models in reference to measured fMRI data". In: *Frontiers in Neuroscience* 17 (2023).

[124] Donghwan Kim et al. "HyPHEN: A Hybrid Packing Method and Optimizations for Homomorphic Encryption-Based Neural Networks". In: *arXiv preprint arXiv:2302.02407* (2023).

[125] Ehsan Hesamifard, Hassan Takabi, and Mehdi Ghasemi. "Deep Neural Networks Classification over Encrypted Data". en. In: *Proceedings of the Ninth ACM Conference on Data and Application Security and Privacy*. Richardson Texas USA: ACM, Mar. 2019, pp. 97–108. ISBN: 978-1-4503-6099-9. DOI: 10.1145/3292006.3300044. URL: https://dl.acm.org/doi/10.1145/3292006.3300044 (visited on 02/26/2021).

[126] Yifei Cai et al. "Hunter: He-friendly structured pruning for efficient privacy-preserving deep learning". In: *Proceedings of the 2022 ACM on Asia Conference on Computer and Communications Security*. 2022, pp. 931–945.

[127] Adi Akavia et al. "Compact storage for homomorphic encryption". In: *Cryptology ePrint Archive* (2022).

[128] Meng Hao et al. "Iron: Private Inference on Transformers". In: *Advances in Neural Information Processing Systems*. 2022.

[129] Zhicong Huang et al. "Cheetah: Lean and Fast Secure ${$Two-Party$}$ Deep Neural Network Inference". In: *31st USENIX Security Symposium (USENIX Security 22)*. 2022, pp. 809–826.

[130] Mengxin Zheng, Qian Lou, and Lei Jiang. *Primer: Fast Private Transformer Inference on Encrypted Data*. en. arXiv:2303.13679 [cs]. Mar. 2023. URL: http://arxiv.org/abs/2303.13679 (visited on 03/30/2023).

[131] Chiraag Juvekar, Vinod Vaikuntanathan, and Anantha Chandrakasan. "GAZELLE: A low latency framework for secure neural network inference". In: *27th USENIX Security Symposium (USENIX Security 18)*. 2018, pp. 1651–1669.

[132] Kaustubh Shivdikar et al. "Accelerating polynomial multiplication for homomorphic encryption on gpus". In: *2022 IEEE International Symposium on Secure and Private Execution Environment Design (SEED)*. IEEE, 2022, pp. 61–72.

[133] Terrance DeVries and Graham W. Taylor. "Improved regularization of convolutional neural networks with cutout". In: *arXiv preprint arXiv:1708.04552* (2017).

[134] Sergey Zagoruyko and Nikos Komodakis. "Wide residual networks". In: *arXiv preprint arXiv:1605.07146* (2016).

[135] Jonas Gehring et al. "Convolutional sequence to sequence learning". In: *International Conference on Machine Learning*. PMLR, 2017, pp. 1243–1252.

[136] P. Mohassel and Y. Zhang. "SecureML: A System for Scalable Privacy-Preserving Machine Learning". In: *2017 IEEE Symposium on Security and Privacy (SP)*. ISSN: 2375-1207. May 2017, pp. 19–38. DOI: 10.1109/SP.2017.12.

[137] Thore Graepel, Kristin Lauter, and Michael Naehrig. "ML confidential: Machine learning on encrypted data". In: *International Conference on Information Security and Cryptology*. Springer, 2012, pp. 1–21.

[138] Raphael Bost et al. "Machine learning classification over encrypted data." In: *NDSS*. Vol. 4324. 2015, p. 4325.

[139] Louis JM Aslett, Pedro M. Esperança, and Chris C. Holmes. "Encrypted statistical machine learning: new privacy preserving methods". In: *arXiv preprint arXiv:1508.06845* (2015).

[140] M. Sadegh Riazi et al. "Chameleon: A hybrid secure computation framework for machine learning applications". In: *Proceedings of the 2018 on Asia Conference on Computer and Communications Security*. 2018, pp. 707–721.

[141] Alex Graves, Abdel-rahman Mohamed, and Geoffrey Hinton. "Speech recognition with deep recurrent neural networks". In: *2013 IEEE international conference on acoustics, speech and signal processing*. Ieee, 2013, pp. 6645–6649. URL: https://ieeexplore.ieee.org/abstract/document/6638947/ (visited on 10/26/2023).

[142] Alex Graves and Navdeep Jaitly. "Towards end-to-end speech recognition with recurrent neural networks". In: *International conference on machine learning*. PMLR, 2014, pp. 1764–1772. URL: https://proceedings.mlr.press/v32/graves14.html (visited on 10/26/2023).

[143] Andrej Karpathy and Li Fei-Fei. "Deep visual-semantic alignments for generating image descriptions". In: *Proceedings of the IEEE conference on computer vision and pattern recognition*. 2015, pp. 3128–3137. URL: https://www.cv-foundation.org/openaccess/content_cvpr_2015/html/Karpathy_Deep_Visual-Semantic_Alignments_2015_CVPR_paper.html (visited on 10/26/2023).

[144] Kelvin Xu et al. "Show, Attend and Tell: Neural Image Caption Generation with Visual Attention". In: *Proceedings of the 32nd International Conference on Machine Learning*. Ed. by Francis Bach and David Blei. Vol. 37. Proceedings of Machine Learning Research. Lille, France: PMLR, July 2015, pp. 2048–2057. URL: http://proceedings.mlr.press/v37/xuc15.html.

[145] Dzmitry Bahdanau, Kyunghyun Cho, and Yoshua Bengio. "Neural Machine Translation by Jointly Learning to Align and Translate". In: *3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7-9, 2015, Conference Track Proceedings*. 2015. URL: http://arxiv.org/abs/1409.0473.

[146] Junyoung Chung et al. "Empirical evaluation of gated recurrent neural networks on sequence modeling". English (US). In: *NIPS 2014 Workshop on Deep Learning, December 2014*. 2014.

[147] Tomas Mikolov et al. "Recurrent neural network based language model". In: *INTERSPEECH*. 2010.

[148] Martin Sundermeyer, Ralf Schlüter, and Hermann Ney. "LSTM Neural Networks for Language Modeling". In: *INTERSPEECH*. 2012.

[149] Ahmad Al Badawi et al. "PrivFT: Private and Fast Text Classification with Homomorphic Encryption". In: *arXiv preprint arXiv:1908.06972* (2019).

[150] Armand Joulin et al. "Bag of Tricks for Efficient Text Classification". In: *Proceedings of the 15th Conference of the European Chapter of the Association for Computational Linguistics: Volume 2, Short Papers*. Valencia, Spain: Association for Computational Linguistics, Apr. 2017, pp. 427–431. URL: https://www.aclweb.org/anthology/E17-2068.

[151] Qian Lou and Lei Jiang. "SHE: A Fast and Accurate Deep Neural Network for Encrypted Data". In: *Advances in Neural Information Processing Systems*. 2019, pp. 10035–10043.

[152] S. Zhang, Y. Gong, and D. Yu. "Encrypted Speech Recognition Using Deep Polynomial Networks". In: *ICASSP 2019 - 2019 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*. May 2019, pp. 5691–5695. DOI: 10.1109/ICASSP.2019.8683721.

[153] Nathan Dowlin et al. *CryptoNets: Applying Neural Networks to Encrypted Data with High Throughput and Accuracy*. Tech. rep. MSR-TR-2016-3. 2016.

[154] Thomas Shortell and Ali Shokoufandeh. "Secure Signal Processing Using Fully Homomorphic Encryption". In: *Advanced Concepts for Intelligent Vision Systems - 16th International Conference, ACIVS, Italy, Proceedings*. 2015.

[155] Jeffrey Pennington, Richard Socher, and Christopher D. Manning. "GloVe: Global Vectors for Word Representation". In: *Empirical Methods in Natural Language Processing (EMNLP)*. 2014, pp. 1532–1543. URL: http://www.aclweb.org/anthology/D14-1162.

[156] Matthew E. Peters et al. "Deep contextualized word representations". In: *Proc. of NAACL*. 2018.

[157] Jacob Devlin et al. "BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding". In: *CoRR* abs/1810.04805 (2018). arXiv: 1810.04805. URL: http://arxiv.org/abs/1810.04805.

[158] Zhilin Yang et al. "XLNet: Generalized Autoregressive Pretraining for Language Understanding". In: *arXiv preprint arXiv:1906.08237* (2019).

[159] François Chollet et al. *Keras*. https://github.com/fchollet/keras. 2017.

[160] Shai Halevi and Victor Shoup. "Algorithms in HElib". In: *Advances in Cryptology - CRYPTO - 34th Annual Cryptology Conference, CA, USA, Proceedings*. 2014.

[161] Andrew L. Maas et al. "Learning Word Vectors for Sentiment Analysis". In: *Proceedings of the 49th Annual Meeting of the Association for Computational Linguistics: Human Language Technologies*. Portland, Oregon, USA: Association for Computational Linguistics, June 2011, pp. 142–150. URL: http://www.aclweb.org/anthology/P11-1015.

[162] Zoe L. Jiang et al. "Privacy-Preserving Distributed Machine Learning Made Faster". In: *arXiv preprint arXiv:2205.05825* (2022).

[163] Frederic P. Miller, Agnes F. Vandome, and John McBrewster. *Amazon Web Services*. Alpha Press, 2010. ISBN: 6131788367, 9786131788369.

[164] Google. "Google Prediction API". In: (2017). URL: https://cloud.google.com/prediction/.

[165] Microsft. "Microsoft Azure Machine Learning". In: (2017). URL: https://azure.microsoft.com/en-us/services/machine-learning/.

[166] Kyunghyun Cho et al. "Learning phrase representations using RNN encoder-decoder for statistical machine translation". In: *arXiv preprint arXiv:1406.1078* (2014).

[167] Ilaria Chillotti et al. *TFHE: Fast Fully Homomorphic Encryption Library*. https://tfhe.github.io/tfhe/. August 2016.

[168] Ehsan Hesamifard et al. "Privacy-preserving Machine Learning as a Service". en. In: *Proceedings on Privacy Enhancing Technologies* 2018.3 (June 2018). Publisher: Sciendo Section: Proceedings on Privacy Enhancing Technologies, pp. 123–142. DOI: 10.1515/popets-2018-0024. URL: https://content.sciendo.com/view/journals/popets/2018/3/article-p123.xml (visited on 12/11/2020).

[169] Q. Feng et al. "SecureNLP: A System for Multi-Party Privacy-Preserving Natural Language Processing". In: *IEEE Transactions on Information Forensics and Secu-*

rity 15 (2020). Conference Name: IEEE Transactions on Information Forensics and Security, pp. 3709–3721. ISSN: 1556-6021. DOI: 10.1109/TIFS.2020.2997134.

[170] Yoshua Bengio, Patrice Simard, and Paolo Frasconi. "Learning long-term dependencies with gradient descent is difficult". In: *IEEE transactions on neural networks* 5.2 (1994). Publisher: IEEE, pp. 157–166.

[171] Ruining He and Julian McAuley. "Ups and downs: Modeling the visual evolution of fashion trends with one-class collaborative filtering". In: *proceedings of the 25th international conference on world wide web*. 2016, pp. 507–517.

[172] Andrew Maas et al. "Learning word vectors for sentiment analysis". In: *Proceedings of the 49th annual meeting of the association for computational linguistics: Human language technologies*. 2011, pp. 142–150.

[173] Shai Halevi and Victor Shoup. "Algorithms in helib". In: *Annual Cryptology Conference*. Springer, 2014, pp. 554–571.

[174] Ronald J Williams and Jing Peng. "An efficient gradient-based algorithm for online training of recurrent network trajectories". In: *Neural computation* 2.4 (1990). Publisher: MIT Press One Rogers Street, Cambridge, MA 02142-1209, USA journals-info . . ., pp. 490–501.

[175] Rajarshi Roy et al. "PrefixRL: Optimization of Parallel Prefix Circuits using Deep Reinforcement Learning". In: *2021 58th ACM/IEEE Design Automation Conference*

(DAC). ISSN: 0738-100X. Dec. 2021, pp. 853–858. DOI: 10.1109/DAC18074.2021.9586094.

[176] Alhussein Fawzi et al. "Discovering faster matrix multiplication algorithms with reinforcement learning". en. In: *Nature* 610.7930 (Oct. 2022). Number: 7930 Publisher: Nature Publishing Group, pp. 47–53. ISSN: 1476-4687. DOI: 10.1038/s41586-022-05172-4. URL: https://www.nature.com/articles/s41586-022-05172-4 (visited on 10/20/2022).

[177] Ashish Vaswani et al. "Attention is all you need". In: *Advances in neural information processing systems* 30 (2017). URL: https://proceedings.neurips.cc/paper/7181-attention-is-all (visited on 11/01/2023).

[178] Tianyu Chen et al. "THE-X: Privacy-Preserving Transformer Inference with Homomorphic Encryption". In: *Findings of the Association for Computational Linguistics: ACL 2022*. Ed. by Smaranda Muresan, Preslav Nakov, and Aline Villavicencio. Dublin, Ireland: Association for Computational Linguistics, May 2022, pp. 3510–3520. DOI: 10.18653/v1/2022.findings-acl.277. URL: https://aclanthology.org/2022.findings-acl.277.