Predicting the Execution Time of Secure Neural Network Inference[⋆]

Eloise Zhang and Zoltán Ádám Mann

University of Amsterdam, Amsterdam, The Netherlands

Abstract. In the secure neural network inference (SNNI) problem, a service provider offers inference as a service with a pre-trained neural network (NN). Clients can use the service by providing an input and obtaining the output of the inference with the NN. For reasons of privacy and intellectual property protection, the service provider must not learn anything about the input or the output, and the client must not learn anything about the internal parameters of the NN. This is possible by applying techniques like multi-party computing (MPC) or homomorphic encryption (HE), although with a significant performance overhead.
One way to improve the efficiency of SNNI is by selecting NN architectures that can be evaluated faster using MPC or HE. For this, it would be important to predict how long SNNI with a given NN takes. This turns out to be challenging. Traditional predictors for NN inference time, like the number of parameters in the NN, are poor predictors of SNNI execution time, since they ignore the characteristics of cryptographic protocols. This paper is the first to address this problem. We propose three different prediction methods for SNNI execution time, and investigate experimentally their strengths and weaknesses. The results show that the proposed methods offer different advantages in terms of accuracy and speed.

Keywords: Privacy-preserving machine learning, Multi-party computation, Homomorphic encryption, Neural networks, Performance prediction

1 Introduction

Compared to traditional statistical methods, approaches based on artificial Neural Networks (NNs) offer distinct advantages. They possess the ability to autonomously learn patterns and capture intricate structures within data, all without necessitating an exhaustive comprehension of the underlying phenomena. As a result, NNs can be successfully applied in a variety of fields [1].

To achieve good accuracy, often huge NNs are used with millions or even billions of tuneable parameters. Training such NNs requires large amounts of training data, significant computational resources, and a high level of expertise. This makes the idea of Machine Learning as a Service (MLaaS) attractive: in

[⋆] This paper was published in the *Proceedings of the 39th International Conference on ICT Systems Security and Privacy Protection (IFIP SEC)*, pp. 481-494, 2024

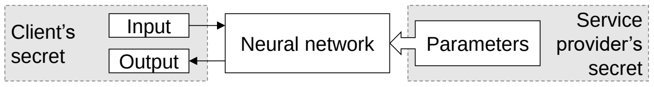

Fig. 1. Overview of the SNNI problem

MLaaS, the training is performed by a specialized service provider. After the service provider has trained the NN, it can offer inference as a service to clients. A client provides an input, and receives the NN's output on the given input.

Such a service may lead to concerns of confidentiality and privacy [26]. The client's input may be sensitive personal information that the client does not want to share with the service provider. On the other hand, the parameters of the NN are the intellectual property of the service provider, which the service provider may not want to share with the client. This leads to the secret NN inference (SNNI) problem, as shown in Fig. 1: how to compute the output of the NN on a given input such that (1) the input and the output remain the client's secret and (2) the parameters of the NN remain the service provider's secret.

In recent years, several cryptographic protocols have been proposed for solving the SNNI problem [20], using different cryptographic primitives, especially from the field of secure multi-party computation (MPC) and homomorphic encryption (HE). These cryptographic protocols lead to significant overhead [7]. One reason why SNNI tends to be very slow is that NNs optimized for traditional (i.e., non-secure) inference may make secure evaluation using MPC or HE very inefficient. For example, the ReLU function, defined as $f(x) = \max(0, x)$, is used extensively in modern NNs and is very fast in traditional evaluation. However, if x is secret-shared between multiple parties or if x is (homomorphically) encrypted, then computing $f(x)$ in a secure way becomes challenging, requiring a sophisticated protocol. Thus, to make SNNI faster, it is important to select NNs that can be efficiently evaluated in a secure manner. For this, it would be useful if we could predict how long SNNI with a given NN would take. Unfortunately, we have currently no way to tell how long SNNI will take on a given NN, other than by actually running it, which may be too costly.

The aim of this paper is to remedy this problem by proposing methods for quickly predicting the approximate runtime of a given SNNI approach on a given NN. We devise three different methods for this purpose: one based on analytical models for different types of NN layers, one using a dedicated NN for each layer of the original NN, and one using a recurrent neural network (RNN). We perform controlled experiments to evaluate the strengths and weaknesses of the three methods. Our results show that they have complementary advantages and disadvantages in terms of accuracy and speed.

2 Preliminaries

Neural networks. A feed-forward NN is a sequence of layers. Layer $j = 1, \ldots, L$ realizes a function $f_j : \mathbb{R}^{n_j} \to \mathbb{R}^{n_{j+1}}$. The input of layer 1 is the input of the NN,

Table 1. Parameters determining the size of different types of layers

Symbol	Meaning	Related layers				
		FC	CONV	MP/AvP	ReLU	BN
N_I, N_O	number of inputs / outputs	✓				
C_I, C_O	number of input / output channels		✓			
H_I, W_I	input height / width		✓	✓		
H_O, W_O	output height / width		✓	✓		
H_F, W_F	filter height / width		✓	✓		
N	number of neurons				✓	
C	number of channels			✓		✓
H, W	height / width of both input & output					✓

the output of layer L is the output of the NN. The output of layer $j = 1, \ldots, L-1$ is the input of layer $j+1$. The layers can be of different types, depending on the function they realize. Some commonly used layer types are the following (the dimension parameters are explained in Table 1; for a positive integer N, the notation $[N]$ is a shorthand for $\{1, \ldots, N\}$):
Fully-connected (FC) computes the function $y = x^\intercal W + b$, where $x \in \mathbb{R}^{N_I}$ is the input, $W \in \mathbb{R}^{N_I \times N_O}$ and $b \in \mathbb{R}^{N_O}$ are parameters, and $y \in \mathbb{R}^{N_O}$ is the output.
Convolutional (CONV) computes the function

$$y[h', w', c'] = \sum_{c \in [C], a \in [H_F], b \in [W_F]} x[h's + a, w's + b, c] \cdot k[c, a, b, c'],$$

where $x \in \mathbb{R}^{H_I \times W_I \times C_I}$ is the input, $k \in \mathbb{R}^{C_I \times H_F \times W_F \times C_O}$ and $s \in \mathbb{R}_+$ are parameters, and $y \in \mathbb{R}^{H_O \times W_O \times C_O}$ is the output.
Max pooling (MP) computes the function

$$y[h, w, c] = \max\{x[hs + a, ws + b, c] : a \in [H_F],\ b \in [W_F]\},$$

where $x \in \mathbb{R}^{H_I \times W_I \times C}$ is the input and $y \in \mathbb{R}^{H_O \times W_O \times C}$ is the output.
Average pooling (AvP) computes the function

$$y[h, w, c] = \frac{1}{|H_F||W_F|} \cdot \sum_{a \in [H_F],\ b \in [W_F]} x[hs + a, ws + b, c],$$

where $x \in \mathbb{R}^{H_I \times W_I \times C}$ is the input and $y \in \mathbb{R}^{H_O \times W_O \times C}$ is the output.
ReLU activation computes the function $y[j] = \max(0, x[j])$, where $x \in \mathbb{R}^N$ is the input and $y \in \mathbb{R}^N$ is the output.
Batch normalization (BN) computes the function $y[h, w, c] = \mu[c] \cdot x[h, w, c] + \theta[c]$, where $x \in \mathbb{R}^{H \times W \times C}$ is the input, $\mu \in \mathbb{R}^C$ and $\theta \in \mathbb{R}^C$ are parameters, and $y \in \mathbb{R}^{H \times W \times C}$ is the output.

Secure NN inference. Current SNNI approaches make use of various cryptographic primitives, particularly additive secret sharing (A-SS), oblivious transfer (OT), garbled circuits (GC), and homomorphic encryption (HE). Based on the primitives, sophisticated protocols can be built for specific layer types [20]. For different types of NN layers, different protocols may be the most appropriate. Current SNNI approaches evaluate the NN layer by layer, employing the protocol deemed most appropriate per layer.

The evaluation of *linear layers* – such as FC, CONV, and AvP – only requires additions and multiplications. These operations are directly supported by additive secret-sharing (using Beaver's multiplication algorithm), as well as by most fully homomorphic encryption schemes.

The evaluation of *nonlinear layers* – such as MP and ReLU – is more complicated and requires other cryptographic primitives to securely implement the used basic nonlinear operations like comparisons. One possibility is to build Boolean circuits for these nonlinear operations, and apply Yao's Garbled Circuits protocol or the Goldreich-Micali-Wigderson protocol to securely evaluate the Boolean circuits. In any case, the secure evaluation of nonlinear layers typically requires a large amount of computation and communications.

3 Problem Statement

Our aim is to predict the execution time of a given SNNI protocol for different NNs. The *input* to the prediction process is the description of a feed-forward NN, given as a sequence of layer descriptions. Each layer description specifies the type and size of a layer. The size of the layer may be characterized by one number (e.g., for a ReLU layer) or by a tuple of numbers (e.g., for a CONV layer). The *output* is an estimate of the time it takes to execute SNNI for one sample, using the given NN.

This prediction functionality can be used, for example, in the context of Neural Architecture Search (NAS). When designing an SNNI service, the service provider can use NAS to identify a NN architecture realizing a good trade-off between accuracy and SNNI time. NAS can be very time-consuming if many candidate architectures have to be evaluated. If SNNI time can be quickly predicted, architectures that would lead to too high SNNI time can be immediately discarded, thus accelerating the search.

For the prediction to be useful, it should ideally be both quick and accurate.

4 Prediction Approaches

We consider three prediction approaches. A possibility is to calculate the SNNI execution time from the time needed to evaluate individual layers, and using different models for each layer type[1]. Considering the used model, we have multiple promising options. We can use an **analytical model** per layer, based on the approximate amount of computation and communication performed in the

[1] In this paper, we focus on the layer types described in Section 2. In future work, our approach could be extended to other layer types, such as those used in transformer architectures (e.g., multi-head attention).

given layer. Alternatively, we can use an **NN**[2] per layer, based on various features of the given layer. Finally, we consider using a single model to predict the SNNI execution time as a whole. Since the length of the input description is not fixed, a recurrent neural network (**RNN**) is suitable for this. These approaches are described in more detail in the following subsections.

4.1 Per-layer-type Analytical Model

SNNI execution time can be estimated as $T = \sum_{i=1}^{L} T_i$, where T_i denotes the execution time of layer i, $1 \leq i \leq L$. To estimate T_i, we observe that all protocols for all types of layers follow the same high-level scheme:

1. The input and the parameters of the layer are prepared. Depending on the specific protocol, this preparation can mean different activities (e.g., encryption, secret sharing, garbling), but the execution time of this step is linear in the size of the input and the size of the parameter set of the layer.
2. The actual evaluation of the layer takes place. Again, depending on the protocol, this can take many forms, but the execution time is in all cases linear in the number of steps of the normal (non-secure) evaluation of the layer. For example, calculating the dot product of two d-dimensional vectors in a linear layer takes $O(d)$ time in the non-secure evaluation, and it also requires $O(d)$ amount of computation and $O(d)$ amount of data transfer using MPC or HE, only with different constants.
3. The output of the layer may need to be extracted. Depending on the cryptographic protocol, this may involve decryption, re-sharing, conversion to a different number representation etc. In any case, the time for this is linear in the output size.

This leads to the following formula: $T_i = c_1 \cdot S_{\text{in}} + c_2 \cdot S_{\text{par}} + c_3 \cdot N_{\text{op}} + c_4 \cdot S_{\text{out}} + c_5$. Here, S_{in}, S_{par}, and S_{out} denote the size of the input, of the parameter set, and of the output, respectively; N_{op} is the number of operations in the non-secure evaluation of the layer; and $c_1, \ldots, c_5$ are non-negative coefficients that depend on the protocol and on the system configuration. The values of S_{in}, S_{par}, N_{op}, and S_{out} can be estimated for different types of layers as in Table 2 [6, 2].

As shown in Fig. 2, for each layer of the original NN, we extract the raw features (e.g., N_I and N_O for a FC layer) and calculate the composite features (S_{in}, S_{par}, N_{op}, S_{out}). These composite features are normalized and fed into a linear regression model to get the correct weighting. The resulting predicted time per layer is then aggregated for all layers.

4.2 Per-layer-type NN

The analytical models introduced above rely on simplifying assumptions that may not always hold in practice. Sophisticated SNNI protocols may use optimizations that break the linearity assumptions used in the analytical model.

[2] Note that this is not the same as the original NN of Fig. 1. The NN of Fig. 1 is a large and complex NN, evaluated securely. Here, we mean multiple (one per layer of the original NN), relatively small NNs, evaluated without secrecy constraints.

Table 2. Factors determining the execution time of different types of layers. The dimension parameters are explained in Table 1.

Layer	S_{in}	S_{par}	N_{op}	S_{out}
FC	N_I	$(N_I+1)N_O$	$2N_I N_O + N_O$	N_O
CONV [3]	$H_I W_I C_I$	$C_I H_F W_F C_O$	$2C_I H_F W_F H_O W_O C_O$	$H_O W_O C_O$
MP/AvP	$H_I W_I C$	0	$H_F W_F H_O W_O C$	$H_O W_O C$
ReLU	N	0	N	N
BN	HWC	$2C$	$2HWC$	HWC

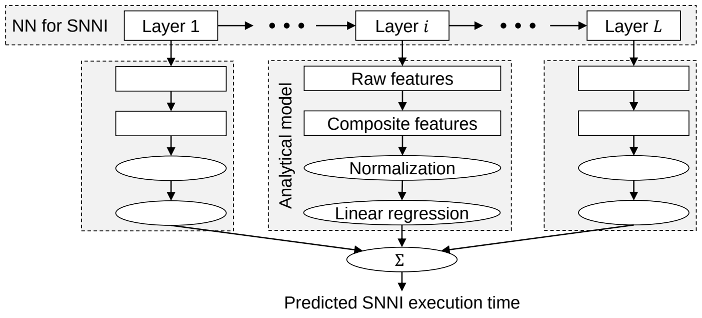

Fig. 2. Prediction using an analytical model per layer of the original NN

E.g., OT extension protocols lead to non-uniform cost for OTs (the first OTs are expensive, later OTs are cheaper). As a result, the analytical models cannot fully capture the complexity of the dependence of SNNI execution time on the size of the layers. To achieve more accurate prediction of SNNI execution time for a given layer, we propose using NNs. We train a separate NN for each layer type of the original NN. For this purpose, we use NNs with fully-connected layers and ReLU activation function, and we use the raw features of Table 1. The approach is visualized in Fig. 3.

To improve prediction accuracy, we leverage hyperparameter tuning techniques like grid search and randomized search [8] to systematically optimize the NN configurations. Through adjustments in hyperparameters such as the learn-

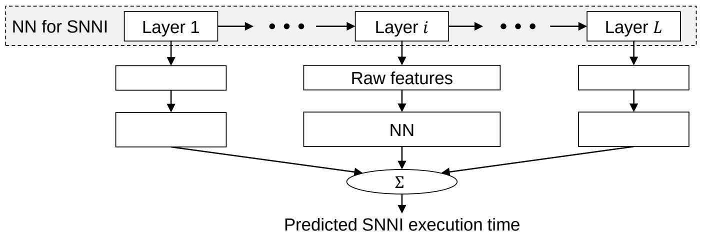

Fig. 3. Prediction using one NN per layer of the original NN

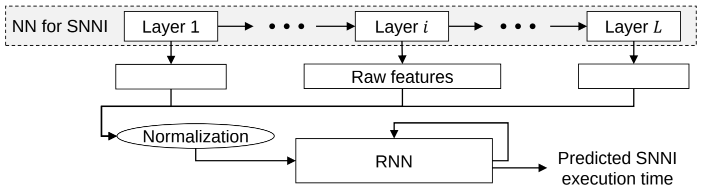

Fig. 4. Prediction using a single RNN

ing rate, number of hidden layers, as well as the number of neurons per layer, we aim to identify the most suitable NN setup.

4.3 RNN for Whole SNNI Execution Time

Recurrent Neural Networks (RNNs) are designed to handle sequential data. They maintain a form of memory, enabling them to capture temporal dependencies and patterns over extended sequences [25]. In our context, the execution of each layer in the SNNI process occurs sequentially. The runtime of one layer can be influenced by preceding layers, reinforcing the suitability of RNNs.

As sketched in Fig. 4, we consider the whole SNNI as one sample, with each layer of the NN representing a discrete step, executed sequentially. The feature for each step corresponds to the features of the layer. Since we aggregate features from different types of layers, each layer has a feature length equal to the total features of all layers. Only some of these features are relevant to the particular layer, with the remaining being zero. As a result, each sample is represented by a sparse matrix. Moreover, due to varying layer lengths across different NNs, we apply padding to standardize the samples to the maximum step length.

First, we implement a masking layer in our RNN, a crucial step in handling sequences of varying lengths. This layer employs a mask value to skip steps, enabling the model to effectively ignore padding values (represented by zeros) and to concentrate on the actual information in each sequence. This ensures that the model learns meaningful patterns and dependencies in the data without being influenced by the padding.

We integrate Long Short-Term Memory (LSTM) Networks into our RNN architecture. LSTM is a variant of RNNs equipped with a memory cell capable of retaining information across extensive sequences. This empowers them to grasp complex temporal relationships and patterns [12, 10]. LSTMs can process input sequences with differing lengths, aligning with our scenario, where the lengths of layers vary across NN architectures. Next, we incorporate multiple FC layers with ReLU activation into the RNN to thoroughly process nonlinear relationships.

To determine the optimal combination of hyperparameters, including the number of layers, units per layer, and learning rate, we leverage automated hyperparameter tuning.

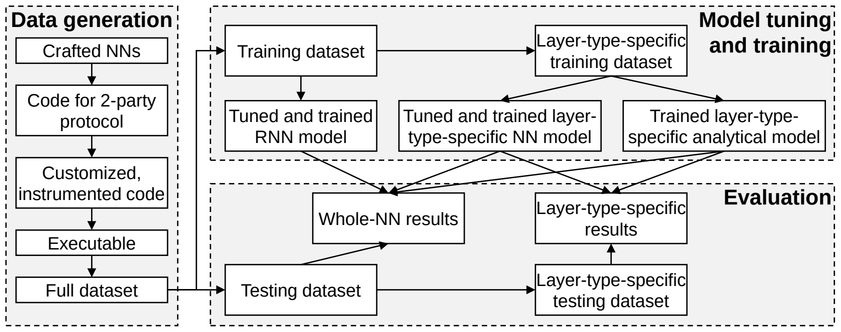
Data generation
Crafted NNs
Code for 2-party protocol
Customized, instrumented code
Executable
Full dataset
Model tuning and training
Training dataset
Layer-type-specific training dataset
Tuned and trained RNN model
Tuned and trained layer-type-specific NN model
Trained layer-type-specific analytical model
Evaluation
Whole-NN results
Layer-type-specific results
Testing dataset
Layer-type-specific testing dataset

Fig. 5. Overview of the experimentation process

5 Experiments

Our experiments involve three phases: data generation, model tuning and training, and evaluation (see Fig. 5). The subsequent paragraphs give details of each phase. The data and scripts of our experiments are publicly available[3].

Data generation. To train our prediction models, we first acquire data of SNNI execution time for various layer sizes for each layer type. This process entails the creation of comprehensive NNs for profiling SNNI execution time. When crafting these NNs, we deliberately diversify their composition by incorporating layers with varying configurations. This provides us with a richer dataset in a given amount of SNNI time.

To run SNNI on the created NNs, we first compile them into code for the appropriate protocol, using a configuration file. The code is further customized to align with a specific monitoring protocol. Then, we instrument the code to track runtime and the influencing factors shown in Table 2 for each layer. Afterwards, we compile and link the code to generate an executable for SNNI. Following this preparatory phase, we execute the secure protocol on the NNs and capture the relevant metrics. The result is a comprehensive dataset containing layer-specific descriptions of the neural network architecture, along with the corresponding execution times. We split this dataset, allocating 80% of it for training, and the remaining 20% for testing purposes.

Model tuning and training. From the training dataset, we construct layer-type-specific datasets for the per-layer prediction approaches. This involves extracting the layer-specific data from the training dataset, grouping the data by layer type, and splitting the data into features, representing the characteristics of each layer, and labels, representing the time for executing each layer.

Models are trained for each method from Section 4. The RNN is trained using the whole training dataset. For training the analytical models and NNs per layer type, the corresponding layer-type-specific training datasets are used.

[3] https://github.com/Eloise2000/SNNI-Performance-Evaluator

To enhance the performance of the NN and RNN models, we use hyperparameter tuning. This involves conducting multiple trials with the RandomSearch tuner from the Keras Tuner library[4] to explore a range of hyperparameters, including learning rate, batch size, and network architecture parameters. After each trial, we extract the hyperparameters that result in the lowest validation loss, ensuring optimal model performance. The complexity of the search space varies across different layer types, as more complex layers may require more hidden layers in the prediction NNs. To uphold computational efficiency, we restrict the number of hidden layers to a maximum of 5. For more information on the explored hyperparameter values, please consult our online repository. Lastly, we conduct model fitting for all three models, fine-tuning the model parameters to achieve the best fit with the training data and minimize prediction errors.

Evaluation. As with the training dataset, also the testing dataset is regrouped into a set of layer-type-specific datasets.

We assess the effectiveness of our prediction methods through a comprehensive evaluation process. For the per-layer prediction approaches, we evaluate the prediction performance for each type of layer independently, using the layer-type-specific testing datasets. We employ 5-fold cross-validation during testing for the analytical model. In addition, we assess the prediction performance of all methods on complete NNs, using the complete testing dataset. The results of these evaluations are presented in Section 5.2.

5.1 Experimental Setup

We use the EzPC framework[5] and the state-of-the-art SNNI library Cheetah[6] for executing SNNI. We use the Athos compiler to compile our tensorflow2-based NNs into code compatible with SNNI. The input dimensions are fixed to 224x224x3, a common image size in the ImageNet dataset. For CONV, MP, and AvP layers, we use square kernels with odd width (1, 3, 5, 7), a prevalent practice. As possible values for the number of filters, we use powers of 2, ranging from 16 to 1024. For execution, both model weights and input data are converted from floating-point to fixed-point representation. We use a bitlength of 41 and a precision of 12 bits, a commonly used configuration [21]. Our prediction models are evaluated on three popular NNs: SqueezeNet, ResNet-50, and DenseNet-121. SqueezeNet is a compact deep learning architecture designed for efficient image classification, achieving high accuracy with a significantly reduced model size [15]. ResNet-50 is known for its effectiveness in training very deep networks [11]. DenseNet-121, characterized by dense connections between layers, offers advantages in feature propagation and parameter efficiency [13].

Our experiments are conducted on two virtual machines, each equipped with 8 GB of RAM. The experiments are carried out in two distinct technical settings. In the *LAN* configuration, the server and client processes operate on separate

[4] https://github.com/keras-team/keras-tuner
[5] https://github.com/mpc-msri/EzPC
[6] https://github.com/Alibaba-Gemini-Lab/OpenCheetah

Table 3. Results for different types of layers. C: Client, S: Server. t_{layer}: average SNNI time for the given type of layer in our dataset. *MAE*: mean absolute error of predicted value. *R2*: coefficient of determination. t_{pred}: average time needed for the prediction.

		LAN experiment				WAN experiment			
Layer	Method	t_{layer} [ms]	MAE [ms]	R2	t_{pred} [ms]	t_{layer} [ms]	MAE [ms]	R2	t_{pred} [ms]
FC C	Analytical	459	72	0.958	4	491	85	0.953	4
	NN		105	0.952	153		83	0.968	139
FC S	Analytical	448	71	0.957	4	472	82	0.953	4
	NN		105	0.952	152		90	0.965	138
CONV C	Analytical	2549	1402	0.642	4	2745	1449	0.646	4
	NN		165	0.989	154		135	0.996	164
CONV S	Analytical	2159	1401	0.521	4	2350	1455	0.544	4
	NN		73	0.998	154		74	0.998	167
MP C	Analytical	3211	286	0.995	3	3757	576	0.992	2
	NN		296	0.990	149		374	0.983	164
MP S	Analytical	3300	314	0.995	2	3855	597	0.992	3
	NN		221	0.995	158		284	0.987	188
AvP C	Analytical	727	111	0.960	3	909	170	0.958	3
	NN		106	0.965	158		132	0.960	152
AvP S	Analytical	766	128	0.954	3	939	182	0.952	3
	NN		113	0.967	153		114	0.971	145
ReLU C	Analytical	1523	202	0.986	2	1655	214	0.985	2
	NN		195	0.985	160		210	0.984	185
ReLU S	Analytical	1796	351	0.937	2	1936	357	0.943	2
	NN		303	0.947	145		343	0.934	175
BN C	Analytical	821	94	0.942	2	1076	163	0.894	2
	NN		62	0.976	166		84	0.948	163
BN S	Analytical	582	90	0.878	2	831	163	0.825	2
	NN		53	0.955	146		86	0.915	170

machines, connected via a local network. Additionally, we simulate a *WAN* setting using the Linux Traffic Control (TC) subsystem. The bandwidth between server and client is approximately 7 Gbits/s for LAN and 380 Mbits/s for WAN. Round-trip times are about 0.5ms for LAN and 10ms for WAN, respectively.

5.2 Results

Comparison per layer type. Table 3 shows the results for two prediction methods: the analytical model and the NN-based approach, across various layer types. The NN-based approach consistently achieves R2 (coefficient of determination) values exceeding 0.9, indicating a robust correlation between predicted

Table 4. Results of the three prediction methods (*Analytical*, *NN per layer*, *RNN*) for different NNs. *NN* is the NN that SNNI is applied to. t_{SNNI} is the duration of SNNI on the NN. *Error* is given as the absolute error of the predicted value, and as the absolute error divided by the actual value. t_{pred} is the time needed for running the prediction.

		LAN experiment				WAN experiment			
NN	Method	t_{SNNI} [s]	Error [s]	Error [%]	t_{pred} [ms]	t_{SNNI} [s]	Error [s]	Error [%]	t_{pred} [ms]
SqueezeNet Client	Analytical		5.5	5.3	21		7.5	6.6	23
	NN	104.0	0.6	0.6	1647	113.7	8.1	7.12	1521
	RNN		0.3	0.3	2994		4.9	4.3	3116
SqueezeNet Server	Analytical		5.1	4.9	22		6.9	6.1	23
	NN	104.0	1.6	1.5	1728	113.7	8.0	7.0	1576
	RNN		4.4	4.2	2817		5.5	4.8	3717
ResNet-50 Client	Analytical		158.7	28.3	35		156.0	26.5	63
	NN	559.8	3.3	0.6	2473	587.7	0.7	0.1	1966
	RNN		10.5	1.9	3051		10.4	1.7	3083
ResNet-50 Server	Analytical		161.1	28.8	38		158.8	27.0	57
	NN	559.8	19.3	3.4	2264	587.7	4.7	0.8	2284
	RNN		5.2	0.9	2950		30.5	5.2	3977
DenseNet-121 Client	Analytical		12.2	2.4	30		8.6	1.6	31
	NN	504.0	6.7	1.3	2274	546.7	12.1	2.2	1804
	RNN		12.0	2.4	3427		6.5	1.2	2999
DenseNet-121 Server	Analytical		6.1	1.2	30		3.5	0.64	30
	NN	504.0	18.0	3.6	2217	546.7	30.7	5.6	2051
	RNN		20.4	4.0	2985		21.6	3.9	3881

and actual execution times. In most cases, the analytical model demonstrates comparable performance to the NN approach, also yielding good results.

For CONV layer, the analytical model yields much worse results than the NN-based approach. Beside Cheetah, we also experimented with the SNNI library Crypten [17]. With Crypten, the analytical model achieved good results also for CONV layers, with R2 values consistently above 0.99 (not shown here due to page limitation). This shows the potential of the analytical model for specific protocols while also indicating variations in performance across different protocols.

On the other hand, the NN-based method incurs higher prediction times than the analytical model. This trade-off between prediction accuracy and computational cost should be taken into consideration when selecting a prediction method. It is also possible to combine the advantages of the two approaches by using the NN-based approach for layers where the analytical model performs less effectively, while relying on the analytical model for others.

Even with the NN-based approach, prediction times remain consistently under 0.2 seconds. The prediction time is always lower – in most cases significantly lower – than the average SNNI execution time. This quick response time allows clients to efficiently select SNNI services based on predicted inference durations.

Whole-NN comparison. Table 4 shows the results on entire NNs using all three prediction methods. The analytical model exhibits less than 6% error on two of the NNs, and an error of 28-29% on the third one. The NN- and RNN-based approaches both demonstrate high predictive accuracy with consistently less than 5% error.

The relatively poor performance of the analytical model on the ResNet-50 NN may be due to the high proportion of CONV layers in this NN and the already mentioned weakness of the analytical model on CONV layers.

The analytical model excels in speed, leading to a trade-off between accuracy and prediction time. The NN-based approach takes approximately 70 times more time than the analytical model. For RNN, this figure rises to about 100 times. Nevertheless, even with the slowest RNN model, the prediction time only amounts to roughly 3 seconds, which is still significantly less than the SNNI time, making it practical and feasible for use.

6 Related Work

To the best of our knowledge, this is the first work to address the problem of predicting SNNI execution time. Related work can be identified in two areas: (i) in SNNI and (ii) in predicting normal (i.e., non-secure) NN inference time.

Work on SNNI. In recent years, the SNNI problem was the subject of intensive research. We refer the reader to the recent survey [20] about SNNI in general. SNNI approaches use several cryptographic techniques, such as homomorphic encryption [5], additive secret sharing [19], garbled circuits [23], and oblivious transfer [18]. State-of-the-art SNNI approaches, like Cheetah [14], typically use a combination of these techniques, so that for different types of layers of the NN, the most efficient protocol can be applied.

Due to the overhead of the used cryptographic techniques, the efficiency of SNNI remains a concern. This is especially critical in resource-constrained environments [22, 27]. Some papers used neural architecture search (NAS) to tune the architecture of the NN so that SNNI becomes faster. For example, NASS searches possible NN architectures while at the same time also optimizing the parameters of homomorphic encryption for SNNI [4]. Delphi uses NAS to decide which ReLU layers to replace with a more SNNI-friendly square activation function [21]. Also CryptoNAS uses NAS to reduce the number of ReLUs in a NN, since the secure evaluation of ReLU is quite time-consuming [9].

Such NAS-based SNNI optimization approaches could significantly benefit from our approach. Without a way to predict SNNI execution time, they resort to either performing the time-consuming SNNI execution or using some far-fetched proxy, like the number of ReLUs, instead of SNNI execution time.

Work on NN inference time prediction. Some previous papers used simple models to approximate the inference time of NNs, with the aim of finding good resource allocations for the NNs or parts of them. For example, Shafi et al. [24] profile the execution time of NN inference using different hardware configurations and develop an analytical model of how inference time depends on different hardware parameters. Collage [16] uses profiling to determine the execution cost of ML workloads, and then employs dynamic programming to find the optimal allocation of the workload to available ML backends.

Predicting inference time only makes sense if the prediction is significantly faster than performing the actual inference. Since normal (i.e., non-secure) inference is much faster than secure inference, only very simple models make sense for predicting normal inference time. However, for predicting secure inference time, also more sophisticated models could be reasonably applied. As our work shows, such more sophisticated models indeed lead to better prediction accuracy.

7 Conclusions

This paper addressed the problem of predicting the execution time of secure neural network inference, which has various important applications in the deployment of secure machine learning services. We proposed three different approaches to solve this problem: using an analytical model, using a NN per layer of the original NN, and using a RNN. Our experiments showed that the three approaches lead to different trade-offs between prediction time and prediction accuracy, with the analytical model being the fastest and the other two approaches being more accurate. Important paths for future research include the extension of our models with parameters of the system configuration, and the prediction of other cost metrics, such as energy consumption.

References

1. Alzubaidi, L., Zhang, J., Humaidi, A.J., et al.: Review of deep learning: concepts, CNN architectures, challenges, applications, future directions. Journal of Big Data **8**, art. 53 (2021)
2. Asperti, A., Evangelista, D., Marzolla, M.: Dissecting FLOPs along input dimensions for GreenAI cost estimations. In: International Conference on Machine Learning, Optimization, and Data Science. pp. 86–100 (2021)
3. Basha, S.S., Farazuddin, M., Pulabaigari, V., Dubey, S.R., Mukherjee, S.: Deep model compression based on the training history. Neurocomputing **573**, 127257 (2024)
4. Bian, S., Jiang, W., Lu, Q., Shi, Y., Sato, T.: NASS: optimizing secure inference via neural architecture search. In: ECAI. Frontiers in Artificial Intelligence and Applications, vol. 325, pp. 1746–1753. IOS Press (2020)
5. Brutzkus, A., Gilad-Bachrach, R., Elisha, O.: Low latency privacy preserving inference. In: International Conference on Machine Learning. pp. 812–821 (2019)
6. Cai, E., Juan, D.C., Stamoulis, D., Marculescu, D.: Neuralpower: Predict and deploy energy-efficient convolutional neural networks. In: Asian Conference on Machine Learning. pp. 622–637. PMLR (2017)
7. Chabal, D., Sapra, D., Mann, Z.Á.: On achieving privacy-preserving state-of-the-art edge intelligence. 4th AAAI Workshop on Privacy-Preserving Artificial Intelligence (PPAI-23) (2023)
8. Feurer, M., Hutter, F.: Hyperparameter optimization. In: Automated machine learning: Methods, systems, challenges, pp. 3–33. Springer (2019)

9. Ghodsi, Z., Veldanda, A.K., Reagen, B., Garg, S.: CryptoNAS: Private inference on a ReLU budget. Advances in Neural Information Processing Systems **33**, 16961–16971 (2020)
10. Greff, K., Srivastava, R.K., Koutník, J., Steunebrink, B.R., Schmidhuber, J.: LSTM: A search space odyssey. IEEE Transactions on Neural Networks and Learning Systems **28**(10), 2222–2232 (2016)
11. He, K., Zhang, X., Ren, S., Sun, J.: Deep residual learning for image recognition. In: IEEE Conf. on Computer Vision and Pattern Recognition. pp. 770–778 (2016)
12. Hochreiter, S., Schmidhuber, J.: Long short-term memory. Neural Computation **9**(8), 1735–1780 (1997)
13. Huang, G., Liu, Z., Van Der Maaten, L., Weinberger, K.Q.: Densely connected convolutional networks. In: Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition. pp. 4700–4708 (2017)
14. Huang, Z., Lu, W.j., Hong, C., Ding, J.: Cheetah: Lean and fast secure two-party deep neural network inference. In: 31st USENIX Security. pp. 809–826 (2022)
15. Iandola, F.N., Han, S., Moskewicz, M.W., Ashraf, K., Dally, W.J., Keutzer, K.: SqueezeNet: AlexNet-level accuracy with 50x fewer parameters and <0.5 MB model size. arXiv preprint arXiv:1602.07360 (2016)
16. Jeon, B., Park, S., Liao, P., Xu, S., Chen, T., Jia, Z.: Collage: Seamless integration of deep learning backends with automatic placement. In: International Conference on Parallel Architectures and Compilation Techniques. pp. 517–529 (2022)
17. Knott, B., Venkataraman, S., Hannun, A., Sengupta, S., Ibrahim, M., van der Maaten, L.: CrypTen: Secure multi-party computation meets machine learning. Advances in Neural Information Processing Systems **34**, 4961–4973 (2021)
18. Kumar, N., Rathee, M., Chandran, N., Gupta, D., Rastogi, A., Sharma, R.: CrypTFlow: Secure TensorFlow inference. In: IEEE Symposium on Security and Privacy. pp. 336–353. IEEE (2020)
19. Liu, J., Juuti, M., Lu, Y., Asokan, N.: Oblivious neural network predictions via MiniONN transformations. In: ACM SIGSAC Conference on Computer and Communications Security. pp. 619–631 (2017)
20. Mann, Z.Á., Weinert, C., Chabal, D., Bos, J.W.: Towards practical secure neural network inference: The journey so far and the road ahead. ACM Computing Surveys **56**(5) (2023), article 117
21. Mishra, P., Lehmkuhl, R., Srinivasan, A., Zheng, W., Popa, R.A.: Delphi: A cryptographic inference service for neural networks. In: 29th USENIX Security Symposium. pp. 2505–2522. USENIX Association (2020)
22. Prins, J., Mann, Z.Á.: Secure neural network inference for edge intelligence: Implications of bandwidth and energy constraints. In: IoT Edge Intelligence. Springer (2024)
23. Rouhani, B.D., Riazi, M.S., Koushanfar, F.: DeepSecure: Scalable provably-secure deep learning. In: 55th Design Automation Conference (2018)
24. Shafi, O., Rai, C., Sen, R., Ananthanarayanan, G.: Demystifying TensorRT: Characterizing neural network inference engine on Nvidia edge devices. In: IEEE International Symposium on Workload Characterization (IISWC). pp. 226–237 (2021)
25. Sherstinsky, A.: Fundamentals of recurrent neural network (RNN) and long short-term memory (LSTM) network. Physica D: Nonlinear Phenomena **404**, 132306 (2020)
26. Timan, T., Mann, Z.: Data protection in the era of artificial intelligence: trends, existing solutions and recommendations for privacy-preserving technologies. In: The elements of big data value: Foundations of the research and innovation ecosystem, pp. 153–175. Springer (2021)

27. de Vries, R., Mann, Z.Á.: Secure neural network inference as a service with resource-constrained clients. In: Proceedings of the IEEE/ACM 16th International Conference on Utility and Cloud Computing. p. art. 8 (2023)