Improving Inference Efficiency and Accuracy in Encrypted Neural Networks

Ketan Jog
Department of Computer Science
Columbia University
New York City, NY 10027
kj2473@columbia.edu

Brandon Zhang
Department of Computer Science
Columbia University
New York City, NY 10027
bwz2104@columbia.edu

Abstract

In this paper, we propose EffHE[1], a privacy preserving protocol for addressing accuracy degradation and slow inference times in leveled homomorphic encrypted neural networks. In particular, EffHE is a client-server model in which the client is responsible for executing non-polynomial activation functions, while the server performs all the linear calculations. Using the MNIST dataset as our benchmark, we find that EffHE almost exactly retains the accuracy of the original neural network. This is an improvement over methods where the same neural network that approximates non-polynomial activation functions observes a drop in accuracy. Additionally, we find that under EffHE, time efficiency for inference improves compared to approximated encrypted neural networks as the number of layers scale. We also provide a proof-of-concept implementation of EffHE built upon TenSEAL, a nascent framework integrating homomorphic encryption with PyTorch.

1 Introduction

Being able to conduct inference on private data is of great interest in the machine learning community. More and more prediction and inference services are being offered at scale. Some well known examples include Google Cloud Compute and Amazon Web Services. However, these third parties get direct access to any and all data sent to them. This might not always be preferable. For instance, consider the case where a third-party provider offers machine learning inference on patient scans as a service. if a hospital would like to submit a patient scan to this service, they cannot send the scan in its raw form since it contains sensitive and private data. Rather, it is much more preferable to first encrypt the data, and then send it to the service provider for inference. However, the machine learning model is generally trained on raw data, and thus cannot directly process the data in its encrypted form.

Homomorphic encryption (HE) offers a way to do this. The first working scheme of Homomorphic Encryption was put forth by Gentry [10], although the idea was postulated much earlier [14]. If an input is encrypted using HE, additions, multiplications, and subtractions are all still supported in this homomorphically encrypted space. For instance, if $H(x)$ homomorphically encrypts x, then $H(x + y) = H(x) + H(y)$. This means applying $H^{-1}(H(x) + H(y))$ returns the output $x + y$. So, in the case of a neural network, the client could first encrypt the data using HE, perform homomorphic additions and multiplications (e.g. for a convolution or linear layer) throughout the network on the server-side, and then decrypt the output to receive a prediction on the client-side.

[1] Our code is available at https://github.com/ketanjog/effhe

One particular kind of HE is called leveled homomorphic encryption (LHE) [5]. In LHE, each cryptographic key has a polynomial modulus size (henceforth referred to as key size), which essentially dictates the number of bits allocated to the key. This in turn determines the maximum number of multiplications that can be successively chained together for an encrypted input, which is also referred to as multiplicative depth. Every homomorphic operation adds a bit of noise to the least significant bits of the output. This noise compounds across multiple operations. The multiplicative depth indicates a noise threshold - the amount of noise an output can tolerate before its significant bits get affected. Thus, in order to increase the multiplicative depth without losing any security guarantees, the key size must also increase. Note that an increase in key size also means an increase in the computational complexity of homomorphic operations. This means that on the same input, working with an encryption with a longer key will make it much slower. Currently, the most popular method for doing this is the CKKS scheme [6] due to its fixed-point arithmetic scheme that makes it particularly suited for neural networks since it can do operations on non-integer representations. To do so, it represents numbers with integer and fractional components, with the precision of the latter set by the user (in terms of the number of allocated bits).

From a privacy standpoint, HE is an attractive option due to the strong security guarantees that it offers compared to other existing methods in the privacy-preserving machine learning (PPML) space, such as multi-party computation (MPC). However, it is infeasible for widespread use due to two major flaws:

- Because HE only supports linear and polynomial operations, it cannot support non-polynomial activation functions commonly seen in deep learning, such as ReLU and sigmoid. Homomorphically encrypted neural networks must then approximate such activation functions, which invariably leads to accuracy degradation since the original neural network structure is no longer preserved.
- Inference with LHE is also extremely slow, especially as the number of layers grow. This is because computational expense scales unfavorably as the multiplicative depth (and thus key size) increases. Using polynomial approximations for non-linear functions only exacerbate this issue, as a second-degree polynomial approximation already uses two additional multiplications (performing the square of input, and then multiplying the result by its coefficient), which is the same amount that two fully connected layers would use.

Because of these limitations, we implement the protocol described in Barni et. al (2006) [4] to alleviate these issues. In particular, we demonstrate a scheme in which the client first encrypts the data and sends the data to a third-party service (i.e. the server). The server then handles the linear operations, while passing the non-polynomial operations to be performed on the client's machine. Since the data can be exposed to the client in its raw form, the client can perform the non-polynomial operations as-is without any approximations. This means that the structure of the original neural network is preserved, and thus we see little-to-no decrease in accuracy. Furthermore, we find that this approach allows the multiplicative depth to be held constant regardless of the number of layers, which improves inference time, even when factoring in the additional latency introduced by server-client communication. The server-client scheme is described in more detail in Section 3.4.2.

2 Related Work

Work on machine learning that aims to preserve privacy of data can be divided into two broad subcategories: secure training and secure inference. Methods like differential privacy [3] have been used to create training and data mining procedures that ensure confidentiality across different data providing groups. Frameworks like federated learning [13]] [12] are being built to establish norms that prepare data in a way that no additional information about the data owner's or holders should be disclosed through the model during its lifetime. These ideas, however, do not carry to the other branch of secure inference. The broad goal of encrypted inference as we are studying it, is to allow two parties, one with query data and the other with a prediction function, to interact in such a way that the prediction verifiably correctly processes the query data, conveying only the result to the enquirer without leaking information about the function, and not gleaning any information about the query data point.

One of the breakthrough approaches to solve encrypted inference was seen in CryptoNets [9], where the authors combined Homomorphic Encryption with Neural Networks to create the first instance of encrypted neural networks that were capable of producing accurate inference on a fully encrypted query. In particular, they utilized LHE, which lets one add and multiply encrypted messages as long as one knows the number of operations that are to be applied to the data. Later, Li et al [8] used a different LHE scheme called FV-RNS to implement Faster-CryptoNets. This paper successfully leveraged transfer learning as a way to create sparser representations in the fine tuned layers, thus reducing the time needed for encrypted operations. More importantly, the pre-trained base layers are moved to the "client" side, thus allowing feature extraction to take place in plaintext. This further improves the time efficiency. However, they still use approximations for non-polynomial activation functions, and furthermore the multiplicative depth of the fine tuned layers is still a bottleneck. Additionally, allowing the base layers to sit on the client side divulges significant information about the model to the client, which violates our definition of encryption inference.

Privacy Preserving Machine Learning (PPML) emerged as a field as Machine Learning as a Service (MLaS) became a more popular offering. An evaluative study [11] on performing HE on neural network computations suggests that the protection of query data is a trade-off between the level of security, accuracy and model complexity. Our work shows that this does not have to be the case. In fact, evidence to support our new formulation comes from Faster CryptoNets, where the idea of moving computations to the client side is successfully implemented. Barni et. al [4] outlined the idea of a secure neural network server client model in 2006, in three levels of security. While homomorphic encryption was still a theoretical concept then, the protocol of neural network security applies directly to our framework, and we leverage security levels 1 and 3 from their work to our benefit.

Many libraries implement a variety of homomorphic encryption mechanisms. We use TenSEAL, an encrypted neural network library built using the Simple Encrypted Arithmetic Library (SEAL)[15]. Other works like Palisade [1], Lattigo [2] also exist. Our choice of TenSEAL is based on user-friendliness.

3 Methods

3.1 Dataset

We primarily consider the MNIST dataset, a popular optical character recognition dataset that is commonly used as a benchmark for encrypted neural network performance. It consists of $60,000$ 28×28 grayscale images of digits ranging from 0 to 9, with $50,000$ images used for training and $10,000$ images used for testing.

3.2 Models

In this paper, we consider two models:

- 1c1f: a network with one convolutional layer followed by one fully-connected layer, with ReLU applied between the two layers.
- 1c2f: a network with one convolutional layer followed by two fully-connected layers, with ReLU applied between each layer.

We necessarily consider shallower models in this work due to the computational cost of inference relative to their unencrypted counterparts. This is partially due to the fact that the framework (described in Section 3.5) we use for homomorphic operations is only supported on the CPU due to the lack of general specialized hardware for secure neural network inference.

3.3 Training Details

We trained each model on the raw MNIST training data for 10 epochs and a batch size of 64. We would like to reiterate that training takes place on the plain unencrypted data.

3.4 Inference details

Since the base forms of 1c1f and 1c2f are trained on the raw MNIST data, they cannot do secure inference in their unmodified forms. Therefore, we introduce two main variants of each model during the inference process: an "eff" version, which implements our main approach, and an "approx" version, which approximates the ReLU functions during the network's forward passes and serves as our main point of comparison for performance benchmarks.

So, the full list of models we consider are 1c1f-eff, 1c1f-approx, 1c2f-eff, 1c2f-approx.

3.4.1 1c1f-approx and 1c2f-approx

For the "approx" models, the forward pass takes in a CKKS encrypted input. It performs the convolutions and fully-connected layers using the homomorphic addition, subtraction, and multiplication operators, and then sends the output back to the client for decryption at the end. However, in between each layer, it uses an approximated ReLU function introduced by Chou et. al. This approximated ReLU function is given by $0.125x^2 + 0.5x + 0.25$, where x is the encrypted input into the function. Note that the additions and multiplications involved in the approximation would also be their homomorphic versions.

For 1c1f-approx, we use a key size of 8192 bits, fractional precision of 26 bits, and a multiplicative depth of 5.

For 1c2f-approx, we use a key size of 16384 bits, fractional precision of 26 bits, and a multiplicative depth of 8.

3.4.2 1c1f-eff and 1c2f-eff (EffHE)

For the "eff" models, we implement a client-server protocol which we call EffHE for the forward pass. In a practical scenario, the client can be interpreted as the customer who wants to send data to a third-party ML service, while the server can be interpreted as the actual service provider. As with the "approx" models, inference receives a CKKS encrypted input and performs the convolutions and fully-connected layers in homomorphic space. The only difference is that the ReLU functions are computed in full (i.e. not approximated) client-side on intermediate plaintext outputs, before being encrypted and sent back to the server for the next stage of the forward pass.

This is perhaps best illustrated by a concrete example, so we will describe the procedure of EffHE for the specific case of 1c2f-eff:

1. The client encrypts the data they want classified and sends it to the server.
2. On the server-side, 1c2f-eff receives the encrypted input from the, performs a convolution in homomorphic space, and sends the result to the client.
3. The client decrypts the result, applies ReLU, encrypts the result, and sends it back to the server.
4. The server receives the encrypted result, applies a fully-connected layer, and sends it to the client again.
5. The client decrypts the result, applies ReLU, encrypts the result, and sends it back to the server.
6. The server receives the encrypted result, applies a fully-connected layer, and sends the final output to the client.
7. Finally, the client decrypts the output, which is the prediction of the network.

Notably, throughout the entire process the server never sees any data in its raw form. Only the client sees sensitive information, which does not present any additional security risk.

For both 1c1f-eff and 1c2f-eff, we use a key size of 8192 bits, fractional precision of 26 bits, and a multiplicative depth of 3.

3.5 Implementation

To implement our 1c1f and 1c2f models (along with their variants for inference), we used TenSEAL, an open-source Python framework for performing CKKS operations built upon the Microsoft SEAL library. It is a particularly attractive option for our use-case since it provides integration with PyTorch models.

For both 1c1f-eff and 1c2f-eff, we provide a full demo of the described client-server protocol as a proof-of-concept. To do so, we used the native sockets library that Python provides to build out the client and server programs.

All tests were run on one 2.3 GHz Quad-Core Intel Core i5 CPU.

4 Results

For our results, we consider both accuracy and timing benchmarks. In particular, we are concerned with whether or not 1c1f-eff and 1c2f-eff retain the original accuracies of their raw counterparts, and whether or not they are faster compared to their approximated equivalents for inference. Note that 1c1f-raw and 1c2f-raw refer to the models where inference is done normally, i.e. in an unencrypted fashion.

4.1 Accuracy Results

Model Inference Accuracies (Percent)	
1c1f-raw	96.72
1c1f-eff	**96.72**
1c1f-approx	79.46
1c2f-raw	97.73
1c2f-eff	**97.73**
1c2f-approx	76.84

Table 1: Model inference accuracies over MNIST test data.

We ran each model over the entire MNIST test dataset, with our approach emphasized in bold. Of particular note is the fact that our encrypted approach performs exactly the same as the raw models that are trained upon and do inference with the plaintext data. By contrast, the approximated versions show noticeable accuracy degradation.

4.2 Timing Results

Layer Type	**1c1f-approx**		**1c2f-approx**	
	Mean	**Std Dev**	**Mean**	**Std Dev**
Convolution	0.17302	0.00485	0.84962	0.08204
First Relu	0.01749	0.0006	0.07612	0.00822
Fully Connected 1	0.59803	0.03578	4.25181	0.4782
Second Relu	-	-	0.05264	0.00362
Fully Connected 2	-	-	0.31279	0.03521
Full Forward Pass	0.78854	0.03522	5.54299	0.56446

Table 2: Layer Wise Analysis: Approximation Network (seconds)

Layer Type	1c1f-eff		1c2f-eff	
	Mean	**Std Dev**	**Mean**	**Std Dev**
Convolution	0.10615	0.0101	0.1148	0.01506
First Relu	8e-05	2e-05	8e-05	3e-05
Fully Connected 1	1.33268	0.1074	1.43814	0.19702
Second Relu	-	-	8e-05	2e-05
Fully Connected 2	-	-	0.29407	0.04539
Full Forward Pass	1.78475	0.12358	2.24049	0.25774

Table 3: Layer Wise Analysis: EFF network (seconds)

We did our timing analysis on separate runs from our accuracy benchmark. Therefore, due to time and computational constraints, we performed our timing analysis on a subset (100 samples large) of MNIST test data, which we believe is sufficient for providing adequate comparisons of average inference times across each model type.

Based on our results for a full forward pass, the 1c1f-eff network does not perform as well as 1c1f-approx in terms of inference speed. However, we observe that adding another fully-connected layer causes the "approx" network's inference time to sharply increase. While 1c2f-eff also takes longer to do inference relative to 1c1f-eff, it is much more efficient than 1c2f-approx, more than doubling its speed to do one forward pass.

In our layer-by-layer analysis, we see that performing the actual ReLU function client-side is much faster than doing the approximated ReLU calculation in encrypted space on the server-side by roughly three orders of magnitude.

We also measured the latencies of the "eff" models during the forward pass, i.e. the total time of sending the intermediate layer output back-and-forth between the server and client. For 1c1f-eff, this was 0.34592 seconds, and for 1c2f-eff, this was slightly longer at 0.39348 seconds. Note that this does not include the time it takes to send the prediction from the server to the client, since this is a cost that would apply irrespective of the approach used.

5 Discussion

5.1 Accuracy Results Discussion

As noted in Section 4.1, our approach shows no change in accuracy compared to the unencrypted raw models. This confirms our thinking that the eff model performs inference in the same fashion. Intuitively, this also makes sense – we are essentially doing the same operations as the original network, so it stands that our predictions should be the same as well. On the other hand, we see significant drop-offs in accuracies of the approximated networks, which reinforces the validity of our approach in maintaining the integrity of the original network.

5.2 Timing Results Discussion

Our main focus is on comparing the inference times using the EffHE protocol (the "eff" models) to the forward pass times in the "approx" models which are run on a single server. While we did measure the handshake time (the time it takes to establish a connection to the server and send the initial input data) to be ≈ 3 seconds, we can disregard it since this one-time cost would apply for any scenario, regardless of the type of model. In other words, the client must always connect and send the data to the service, so this cost is universal rather than model-specific.

Originally, our hypothesis was that the ReLU activation functions would be the bottlenecks in performing inference. As mentioned in the results section, we do observe a thousand-fold speed up in the non-linearities from moving to the "eff" models, but it is insignificant when compared to the

overall inference time per image. Rather, we find that the performance improvement comes from the fact that we can hold the multiplicative depth and key size constant regardless of the number of layers in the network. This is because we essentially "refresh" the noise tolerance every time we decrypt and re-encrypt the data, which means that our multiplicative depth only needs to match the maximum number of consecutive multiplications within a particular layer. This is in contrast to the approximated networks that must have a sufficiently large multiplicative depth to match the number of consecutive multiplications totaling across *all* layers.

This makes itself apparent as the number of layers grow. 1c1f-eff fails to outperform 1c1f-approx in terms of average inference time, likely due to the fact that both use the same key size of 8192. However, 1c2f-approx is forced to use a larger key size of 16384 to accommodate the greater multiplicative depth, which leads to a dramatic increase in inference time. On the other hand, due to the resetting of the noise tolerance after each layer for 1c2f-eff, we can keep the multiplicative depth and key size the same, and thus becomes much more performant time-wise when compared to 1c2f-approx. One minor surprise to us was the fact that we would have expected the increase in time per-layer to be more uniform. Therefore, one area of future research would be to look at TenSEAL's source-code to better understand how their implementation impacts layer-wise calculations on a deeper, more theoretical level. Nevertheless, it does seem our approach is more scalable when compared to the approximated network.

Our EffHE implementation is also done on a local machine, and thus the server-client latencies are unoptimized. We believe that in a practical setting, data transfer can be done in the order of miliseconds, which would improve inference times for the "eff" models. This would also improve their scalability with respect to the number of layers, especially when compared to their "approx" versions. While the inference times are still slow compared to unencrypted neural networks, we believe that our work brings LHE and its security guarantees a step closer to becoming feasible for more widespread adoption.

5.3 Future Work

Encrypted neural network inference is still a growing field, and thus we believe there are many interesting directions to pursue. For instance, in our work, we evaluate our server-client protocol on the ReLU activation function. However, there are many other choices of non-polynomial activation functions that popular neural network architectures use, such as sigmoid and tanh. An extensive study of the effects of choice of such activation functions might be beneficial to the overarching PPML community, as well as serving to solidify our results. For instance, while the improvement in evaluation time might not differ across different activation functions, we might see differences in the improvement of prediction accuracies between ours and the approximated implementations (though we would still expect our approach to retain the accuracy of its unencrypted version). We have also limited our evaluation to the MNIST dataset to establish proof of concept. A more thorough exploration might involve other further testing on other benchmark datasets like CIFAR-10. Unfortunately, it still stands that encrypted inference is quite slow compared to the industry standard plaintext evaluation of prediction queries, so evaluation on even larger datasets might be a more distant possibility.

Barni et al's discussion on neural network security protocols introduces the concept of fake neurons. In our current implementation, during inferene the client can see the number of neurons per layer before a non-polynomial activation, as well as the number these activations. This can easily be avoided using Levels 2 and 3 of Security as described in Barni et. al [4], where additional neurons are padded to the output of each layer to hide the number of them. A further extension of this would be fake layers, where we could send fake output to the client in order to conceal the number of layers. Implementing these obfuscation measures in essence fully complete our definition of encrypted inference: no information about the prediction function will be leaked to the querying entity. It should be noted that these additional features might affect inference time slightly due to the increased size of data being sent in the case of fake neurons, as well as the number client-server interactions needed in the case of fake layers. However, given the relatively small proportion of time the server-client

latencies occupy in the inference process, we do not expect this to become an immediate bottleneck.

Our work uses the client-server communication in between the neural network computations to essentially reset the noise accumulation in the encryption. This allows us to use a fixed key size, independent of the multiplicative depth of the computation circuit. There are other ways of achieving this noise reset. One method is programmable bootstrapping, implemented on TFHE, that allows for efficient computation of homomorphic operations [7]. While we have not yet explored this avenue, such an approach would perhaps enable our benefits of accuracy and time speedup to be achieved on a single device (avoiding the latency of client-server back and forth).

Finally, we think it would be useful to develop a mathematical model of how our protocol scales better than frameworks like Faster CryptoNets and other boilerplate implementations. This would entail quantifying the effect of key size on the computation time of homomorphic operations. In general, as the key size increases, these operations take longer even on the same inputs. Since our method keeps key size constant, we would want to analytically show how our protocol is better.

6 Conclusion

Overall, EffHE's server-client encrypted neural network approach successfully recovers the original accuracies of the unencrypted neural network trained on plaintext, which is not the case in their approximated counterparts. This is because the non-polynomial activations are performed in full on the client-side, bypassing LHE's inability to calculate them. Furthermore, we also find that our approach allows the key size and multiplicative depth to remain constant regardless of the number of layers in the neural network. This addresses one of the main limitations of LHE's applicability to deep learning, where the multiplicative depth – and thus computational cost – would increase with the number of layers in the network. To demonstrate this, we provide a working client-service program which in the 1c2f case, even with unoptimized latencies, manages to significantly outstrip an approximated network of the same structure.

7 Bibliography

[1] https://palisade-crypto.org.

[2] Lattigo v3. Online: `https://github.com/tuneinsight/lattigo`, April 2022. EPFL-LDS, Tune Insight SA.

[3] Martin Abadi, Andy Chu, Ian Goodfellow, H. Brendan McMahan, Ilya Mironov, Kunal Talwar, and Li Zhang. Deep learning with differential privacy. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*. ACM, oct 2016.

[4] M. Barni, C. Orlandi, and A. Piva. A privacy-preserving protocol for neural-network-based computation. In *Proceedings of the 8th Workshop on Multimedia and Security*, MMamp;Sec '06, page 146–151, New York, NY, USA, 2006. Association for Computing Machinery.

[5] Zvika Brakerski, Craig Gentry, and Vinod Vaikuntanathan. (leveled) fully homomorphic encryption without bootstrapping. *ACM Trans. Comput. Theory*, 6(3):13:1–13:36, 2014.

[6] Jung Hee Cheon, Andrey Kim, Miran Kim, and Yongsoo Song. Homomorphic encryption for arithmetic of approximate numbers. In Tsuyoshi Takagi and Thomas Peyrin, editors, *Advances in Cryptology – ASIACRYPT 2017*, pages 409–437, Cham, 2017. Springer International Publishing.

[7] Ilaria Chillotti, Marc Joye, Damien Ligier, Jean-Baptiste Orfila, and Samuel Tap. Concrete: Concrete operates on ciphertexts rapidly by extending tfhe. In *WAHC 2020–8th Workshop on Encrypted Computing & Applied Homomorphic Cryptography*, volume 15, 2020.

[8] Edward Chou, Josh Beal, Daniel Levy, Serena Yeung, Albert Haque, and Li Fei-Fei. Faster cryptonets: Leveraging sparsity for real-world encrypted inference, 2018.

[9] Nathan Dowlin, Ran Gilad-Bachrach, Kim Laine, Kristin Lauter, Michael Naehrig, and John Wernsing. Cryptonets: Applying neural networks to encrypted data with high throughput and accuracy. Technical Report MSR-TR-2016-3, February 2016.

[10] Craig Gentry. Fully homomorphic encryption using ideal lattices. In Michael Mitzenmacher, editor, *STOC*, pages 169–178. ACM, 2009.

[11] Nestor J. Hernandez Marcano, Mads Moller, Soren Hansen, and Rune Hylsberg Jacobsen. On fully homomorphic encryption for privacy-preserving deep learning. In *2019 IEEE Globecom Workshops (GC Wkshps)*, pages 1–6, 2019.

[12] Tian Li, Anit Kumar Sahu, Ameet Talwalkar, and Virginia Smith. Federated learning: Challenges, methods, and future directions. *IEEE Signal Processing Magazine*, 37(3):50–60, may 2020.

[13] H. Brendan McMahan, Eider Moore, Daniel Ramage, Seth Hampson, and Blaise Agüera y Arcas. Communication-efficient learning of deep networks from decentralized data. 2016.

[14] R L Rivest, L Adleman, and M L Dertouzos. On data banks and privacy homomorphisms. *Foundations of Secure Computation, Academia Press*, pages 169–179, 1978.

[15] Microsoft SEAL (release 4.0). `https://github.com/Microsoft/SEAL`, March 2022. Microsoft Research, Redmond, WA.

8 Appendix

As linked in the abstract, our code is available at `https://github.com/ketanjog/effhe`. For testing and training baseline models, most of our code was taken from and then further adapted from `https://github.com/OpenMined/TenSEAL/blob/main/tutorials/Tutorial%204%20-%20Encrypted%20Convolution%20on%20MNIST.ipynb`.

However, the main focus of our paper – the client-server model – is our original code, which we describe in more detail below:

The server files for the "eff" models are `effhe/server_client/server_1c1f.py` and `effhe/server_client/server_1c2f.py`. This includes the client-server communication utility file, `effhe/server_client/message_protocols.py`.

Their respective client files are located at `effhe/scripts/client_vs_1c2f.py` and `effhe/scripts/client_vs_1c1f.py`, which also include timing details of the run.

Finally, we also had to modify the corresponding model files to make it work with the server-client model. These files are `effhe/models/baseline_relu_1c2f_enc.py` and `effhe/models/baseline_relu_1c1f_enc.py`. While originally based on the tutorial code, we made significant modifications to the forward inference pass to the point where almost all of it is our original work.

The repository's README contains more detail about how to run the demo of our code.