**Homomorphic Encryption for Secure Evaluation of Convolutional Neural Networks on Sensitive Image Data**

Final Project Report

Arizona State University

Table of Contents

[Abstract 3](#_Toc152097384)

[Introduction/Overview 3](#_Toc152097385)

[Problem statement 5](#_Toc152097386)

[Significance of Study 5](#_Toc152097387)

[Objectives 6](#_Toc152097388)

[Design Process 7](#_Toc152097389)

[Requirements 10](#_Toc152097390)

[Scope 11](#_Toc152097391)

[Development process 12](#_Toc152097392)

[Tools 14](#_Toc152097393)

[Technical Description of Project 15](#_Toc152097394)

[Testing and Results 22](#_Toc152097395)

[Summary and Conclusions 26](#_Toc152097396)

[References 29](#_Toc152097397)

[Appendix 30](#_Toc152097398)

Abstract

Homomorphic encryption is a cryptographic approach for performing secure computation on encrypted data. Homomorphic encryption presents a viable option for privacy-preserving machine learning in the setting of convolutional neural networks (CNNs) and sensitive image data. The goal of this project is to investigate the use of homomorphic encryption for the secure assessment of CNNs on sensitive image data.

Introduction/Overview

Fully homomorphic encryption (FHE) is a cryptographic technique that allows computations to be performed on encrypted data without first decrypting it. As a result, FHE is ideal for both protecting sensitive data and enabling machine learning (ML) applications.

Google Colaboratory is a cloud-based machine learning platform that simplifies the creation, training, and deployment of ML models. Google Colaboratory also features management and monitoring functions for machine learning models.

FHE is a good way to protect sensitive data in machine learning applications. Google Colaboratory makes it easier to deploy FHE-powered machine-learning models. By combining FHE with Google Colaboratory endpoints, organizations may perform secure, real-time inference on encrypted data.

Organizations can use FHE with Google Colaboratory endpoints to produce predictions on encrypted data using machine learning models. This allows businesses to protect sensitive data while still using it for machine learning.

There are other FHE libraries available, such as HElib and SEAL. You must choose an FHE library that is compatible with your Google Colaboratory endpoint.

* Deploy a trained machine learning model to a Google Colaboratory endpoint: Google Colaboratory may be used to deploy a variety of ML models, such as classification, regression, and natural language processing models.
* Decrypt the inference results: You must use the same FHE library that you used to encrypt your data to decrypt the inference findings.

**Benefits of using FHE with Google Colaboratory endpoints for secure, real-time inferencing**

* **Protect sensitive data:** FHE allows businesses to protect sensitive data from unauthorized access even when it is used for machine learning.
* **•Real-time inference:** Google Collaboratory endpoints support real-time inference, allowing businesses to obtain predictions on encrypted data in real time.
* **Simple to use:** Google Collaboratory simplifies the deployment and management of machine learning models.

**Real-time applications on how FHE can be used to protect sensitive data in machine learning applications.**

* **Financial institution:** A financial institution might employ FHE to encrypt customer data before running a machine learning model to predict fraud. This would allow the financial institution to protect its customers' data while also detecting fraud.
* **Healthcare provider:** A healthcare provider may use FHE to encrypt patient data before deploying a machine-learning model to diagnose diseases. This would allow the healthcare provider to keep its patients' data while still providing the best service possible.
* **Government agency:** A government agency may use FHE to encrypt classified information before analyzing it using a machine-learning model. This would enable the government agency to protect sensitive data while still using it to make vital decisions.

Problem statement

Organizations must be able to protect sensitive data while also using it for machine learning applications. Traditional machine learning approaches, on the other hand, demand that the data be encrypted before it can be used. This means that while the data is being processed, it is vulnerable to unwanted access. There is a need to activate privacy-preserving ML predictions for the most highly regulated environments, where predictions (inference) use encrypted data and the results are only decrypted by the end consumer (client side). Homomorphic encryption is a cryptographic technique that enables secure computation over encrypted data. In the context of convolutional neural networks (CNNs) and sensitive image data, homomorphic encryption offers a promising solution for privacy-preserving machine learning. This project aims to explore the application of homomorphic encryption for the secure evaluation of CNNs on sensitive image data.

Significance of Study

This research shows how to use homomorphic encryption for real-time confidential machine-learning inference. Predictions on sensitive data can be kept secret even in highly regulated situations by incorporating homomorphic encryption into the Google Colaboratory platform. This enables businesses to gain insights from data while maintaining end-to-end encryption.

This project represents significant progress in making homomorphic encryption feasible for production deployments. However, more work needs to be done to make confidential inferences more common. In order to determine ideal use cases, it will be necessary to evaluate performance tradeoffs and optimizations. Extending the approach to model training could have far-reaching consequences.

This project illustrates the possibility of confidential computing services on cloud platforms by demonstrating interaction with Google Colab. Continued research into cryptographic approaches, algorithms, and infrastructure, on the other hand, will be vital to driving adoption. The results of this approach could be used to inform the creation of other privacy-preserving analytics services.

Objectives

The goal of this project is to create a scalable Fully Homomorphic Encryption (FHE) pipeline on Google Colaboratory that allows for model inference on encrypted data without the need for decryption (Google, n.d.).The goal is to show FHE by performing encrypted evaluation on MNIST instances using a convolutional neural network. As illustrated in the picture below, the stages will involve using Google Colaboratory to explore and prepare the data, train the model, and deploy it to an endpoint. Before submitting the data for processing, the client program will encrypt it. The model will then perform classification on the encrypted data within an isolated T4 GPU kernel, ensuring that it remains encrypted.

The model will generate an encrypted forecast, which the client application can optionally decrypt. The data handled by the model for inference will be encrypted throughout its lifecycle, including during runtime within the processor in the T4 GPU kernel (Google, n.d.). This architecture will demonstrate how FHE allows model inference without decryption while simultaneously employing Google services for scalability by keeping the data protected end-to-end. This project's overall goal is to use FHE and Colab to enable privacy-preserving machine learning.

A diagram of a computer security system

Description automatically generated

Design Process

The fundamental requirements for this project are to enable secure computation on encrypted data using homomorphic encryption, evaluate convolutional neural networks on sensitive image data confidentially, and showcase practical applications for real-time private machine learning inference. A key objective is integrating homomorphic encryption into Google Colaboratory to allow predictions on encrypted data, even in highly regulated settings like healthcare. Keeping data encrypted end-to-end including during model execution is essential, necessitating encryption at the client side before sending to Google Colab, evaluation within isolated environments, and encrypted results that are only decrypted by the client. Performance considerations around factors such as latency, throughput, computational overhead, model complexity, batch size, and hardware implications due to the encryption also need evaluation. Optimizing these vectors is necessary for viable deployments while extending confidential computing to model training could provide efficiency benefits but requires engineering efforts.

The model will generate an encrypted forecast that the client application can optionally decrypt. The data handled by the model for inference will be encrypted throughout its lifecycle, even during execution within the processor in the T4 GPU kernel (Google, n.d.). This design will demonstrate how FHE allows model inference without decryption while simultaneously employing Google services for scalability by keeping data secured end-to-end. Overall, the goal of this research is to use FHE and Colab to enable privacy-preserving machine learning.

The design necessitates the coordination of complicated interactions between encryption, machine learning, and cloud components such as Google Colab and cryptography libraries. Google Colaboratory automates data, modeling, training, and deployment workflows while simultaneously providing isolated settings for confidential inference. The Microsoft SEAL library supports homomorphic encryption, allowing computations on encrypted data. End-to-end encrypted machine learning pipelines can be supported by modular design across these domains, which include client-side data encryption/result decryption, transmitting input/output data with public keys, loading encrypted data into enclaves, inference by SEAL-integrated PyTorch models within the enclaves, and returning encrypted outputs to clients.

SEAL and its Python interface power confidentiality with performant homomorphic encryption techniques that allow math on encrypted data. Its tiered method groups parameters and keys based on model complexity requirements. SEAL provides direct processing of encoded data for neural network assessments by portraying plaintext as polynomial structures encrypted into polynomial ciphertexts. Client-side encryption keys are generated and supplied to Colab in order to encrypt outputs. The results contain inferences encoded as polynomials that can only be decrypted with the client's private keys, ensuring confidentiality. The pipeline includes key creation, data encryption, parameter transmission, enclave result encryption, and final client-side decryption.

For real-time inference, the machine learning technique employs Colab for data management, PyTorch estimators injected with SEAL for homomorphic model training, and encrypted model deployment to a Google Colaboratory endpoint within an isolated environment. Client apps send encrypted inputs, and SEAL supports encoded computation on activations within the isolated environment, allowing inferences to be generated without decryption. Colab, Google Drive, and the isolated environments are used for data preparation, with datasets on Google Drive, a bespoke PyTorch estimator linked with SEAL for encrypted training, and hosting the trained model on a Colabr endpoint within an isolated inference. This ensures privacy across the machine-learning pipeline.

System development begins with creating cloud environments based on the architecture. Pandas/NumPy are used for data handling in Jupyter notebooks. Pyfhel makes FHE integration easier. Customizing PyTorch estimators using HELib/TenSEAL offers homomorphic training orchestration via private VPC endpoints. Following that, SEAL integrations inject encryption, optimizing performance by fine-tuning parameters such as modulus selection, lattices, and multi-key mechanisms to balance security, accuracy, and efficiency (GitHub - OpenMined/TenSEAL: A Library for Doing Homomorphic Encryption Operations on Tensors, n.d.). POLYSAFE tensors and encoders in Pyfhel convert data to encrypted polynomial structures. Model management Encryption is built into libraries. Integration is completed by connecting the pipelines of the client, framework, and Colab.

Development process

In Google Colaboratory, the team built a machine learning workflow environment complete with data preparation, model training, and deployment tools. The MNIST image dataset was loaded during training. Colab notebooks were used to preprocess the photographs, standardizing pixel values and augmenting with extra channels to increase model performance.

Using PyTorch, a convolutional neural network with two convolution layers and two fully linked layers achieved 98% test accuracy on plaintext MNIST data. To provide privacy-preserving inference, Microsoft's SEAL homomorphic encryption module was implemented via the TenSEAL Python wrapper. TenSEAL use lattice cryptography to compute on encrypted data without first decrypting it.

The trained neural network weights and parameters were encoded into polynomial representations compatible with TenSEAL for homomorphic evaluation.

Before passing the model, test photos are encrypted locally with TenSEAL public keys. The CNN calculations then perform direct operations on the ciphertext data, producing encrypted predictions. On the encrypted test photographs, the integrated model retained 99% accuracy in comparison to the original plaintext version. Clients use their secret keys to decrypt results locally rather than on the server (GitHub - OpenMined/TenSEAL: A Library for Doing Homomorphic Encryption Operations on Tensors, n.d.).

TenSEAL was natively coupled with Colab and PyTorch to provide a secret inferencing pipeline with end-to-end encryption to preserve sensitive data confidentiality. The next objective was to show the model's real-time outputs. Matplotlib was used for display and analysis of predictions. Benchmarking was also performed for performance optimizations such as batching, computational precision tuning, model quantization, and complexity reduction.

The computational costs incurred by homomorphic encryption caused challenges for the project. The initial objective was to assess the final encrypted model throughout the whole 10,000-image MNIST test segment. On a compute optimized Colab system, with encryption enabled, each batch took more than 2 minutes. For complete testing, extrapolating required more over 50 hours, exceeding the free tier restrictions. As a result, experiments were reduced to a 100-image subset that took 3 minutes to complete. This emphasized the prohibitive overheads that homomorphic encryption currently imposes, prompting demonstrable improvements.

The underlying polynomial mathematical representations employed during encrypted computations also caused numerical instability. Controlling error growth required accurate calibration of encryption parameters. There were also framework problems in using tools like TenSEAL, which necessitated knowledge of specialist concepts like modulus switching and Galois keys. Integrating them with Colab and PyTorch introduced new hurdles. While the privacy guarantees were unprecedented, the overheads, stability, and convenience of use were not. Model and pipeline optimization were required to address issues.

Finally, this trailblazing research effort yielded an encrypted ML prototype that coupled Colab's process automation capabilities with the security of homomorphic encryption on sensitive data. The steps that followed required hardening this into a high-performance, widely tested system that was on the cusp of widespread adoption. Long-term advancement in this domain has the potential to enable previously impossible investigations on exceedingly sensitive datasets, hence enabling societally beneficial research.

Tools

**Google Colaboratory:** For building and training ML models as well as deploying trained models to endpoints for inference.

**SEAL:** Microsoft's homomorphic encryption library that will provide the underlying FHE cryptography

**HELib:** Python libraries that provide interfaces to SEAL's functionality.

**Google Drive:** For storing datasets, model artifacts, etc.

**Other requirements:**

**Numpy:** For numerical and mathematical computing

**Pandas:** For data preprocessing and feature engineering

**PyTorch:** For building deep learning models.

**OS:** For file system operations like loading data

**Pickle:** For serializing Python objects like encrypted data

So, for the ML components, crucial Python libraries would be numpy, pandas, and PyTorch, Pickle and JSON for serialization, requests for API calls, and OS, math, base64, and so on. For the FHE and Colab interaction, the Pyfhel/HELib, Pickle, JSON, Requests, and Base64 modules are extremely important.

Technical Description of Project

The purpose of this project is to examine the use of homomorphic encryption for the secure evaluation of CNNs on sensitive image data by setting up a machine learning pipeline on Google Colab for tasks such as data preparation, model training, and hosting the trained model (Google, n.d.). This project makes use of the Modified National Institute of Standards and Technology (MNIST) database, which comprises 70,000 small 28x28 pixel grayscale photos of handwritten single digits spanning from 0 to 9. Before submitting the training data to the model for training, it is prepared in a Colaboratory notebook. This project will then call our FHE-enabled training script using the SEAL FHE cryptography tools. We utilized the PyTorch framework to create this estimator. Once the estimator is in place, the model will be trained on the training data, yielding an encrypted model.

Convolutional filters extract spatial properties first, followed by highly coupled layers that identify patterns in the machine learning model. Using four filters with 7x7 kernel widths and 3x3 strides, the convolution layer transferred input bitmaps into a filtered representation summarizing visual attributes. The square activation function created sparsity by boosting distinguishing features. This compressed output was flattened into a 256-dimension vector that summarized important visual cues per image and served as the input for subsequent processing.

By incorporating derived convolution filters into a 64-node interpretation mapping, the first dense layer returned features into an encoded feature space. Repeated squaring activation emphasizes separation between encodings to different classes. Softmax normalization was utilized in the final logits layer to perform 10-way classification over digit categories. The general model architecture was designed to strike a balance between accuracy demands and the complexity constraints imposed by homomorphic encryption approaches.

A diagram of a layer

Description automatically generated with medium confidence

In this study, Microsoft's Simple Encrypted Arithmetic Library (SEAL) was used to establish privacy-preserving secret computation based on homomorphic cryptographic protocols. SEAL offers performant lattice-based leveled homomorphic encryption algorithms that allow computations on encrypted material without prior decryption. The Cheon-Kim-Kim-Song (CKKS) technique, in example, organizes encryption parameters and keys into tiers corresponding to a supported number of successive operations, allowing capacity to be configured for a wide variety of model complexities.

SEAL turns plaintext numeric data into encrypted polynomial structures generated with public keys, producing ciphertext outputs that are also polynomial forms that may be manipulated mathematically. SEAL performs encrypted neural network evaluations directly on encrypted polynomial representations. It uses polynomial approximations to represent information like neural weights, with nonlinearity approximations allowing for activation operators. SEAL oversees internal ciphertext arithmetic processes. FHE private keys are only used at endpoints to ensure end-to-end confidentiality.

A screenshot of a computer

Description automatically generated

**Image to column convolution with CKKS - step 1**

A diagram of a computer program

Description automatically generated with medium confidence

**Image to column convolution with CKKS - step 2**

This project's encryption workflow begins with the usage of SEAL and integration libraries such as Pyfhel to encrypt client input data into CKKS polynomial ciphertexts on trusted settings. The model's encoded parameters were sent to Google Colaboratory, which hosted it in a secure environment. Additional public keys were generated to provide result encryption for client communication. Within the confined environment, SEAL-powered encoded neural network computations on encrypted data were accomplished by homomorphically changing underlying polynomial structures.

Model inferences were encoded in polynomial ciphers that could only be decrypted using client-side private keys in the encrypted outputs returned. By integrating SEAL with Colab endpoints and progression frameworks that extend native homomorphic support, such as PyTorch, an efficient, seamless secret deep learning pipeline develops. The system preserved encrypted data across all pipeline steps, including transitory runtime conditions, maintaining integrity while not influencing predictions.

This secret computing technology offers exceptional privacy safeguards by retaining end-to-end encryption. Overheads, on the other hand, caused difficulties in terms of calculation times, expected resource requirements for complete evaluations on full datasets, numerical stability concerns, and framework complexities when including specialist libraries such as SEAL. However, these difficulties of efficiency and precision highlight optimization potential rather than intrinsic limitations of homomorphic encryption capability.

By addressing these areas through research advances and continued innovation, increasing efficiency alongside trustworthy security, this pioneering effort could lay the groundwork for a new generation of confidential AI systems upholding stringent regulatory mandates around data privacy while empowering indispensable analytics - a milestone for Privacy Enhancing Technologies materializing societal benefit - by addressing these areas through research advances and continued innovation, increasing efficiency alongside trustworthy security.

Following training, the model will be deployed to a Colab real-time inference endpoint, which will receive encrypted data from client applications (Google, n.d.). On the client side, the SEAL library will be used to encrypt local data for inference. We created encryption parameters such as public and private keys using SEAL. The client then sends the encrypted data to the endpoint, along with the public keys. This allows the model to encrypt the results it sends to the client while leaving the data unencrypted. When the encrypted data is received at the endpoint, the model employs SEAL for homomorphic computations to do inference on it within a secure isolated environment. As a cryptography toolkit, Microsoft SEAL is used for the fully homomorphic encryption (FHE) library. SEAL implements the BFV and CKKS homomorphic encryption methods efficiently, allowing computations on encrypted data to be performed without first decrypting it (Bourse et al., 2018).

SEAL encodes plaintext data as polynomials with a public key to generate usable ciphertext polynomials (NumPy, n.d.). SEAL is responsible for carrying out mathematical calculations on the encrypted polynomials. SEAL encodes information such as neural network weights as polynomials and gives polynomial approximations of non-linear functions to aid in the evaluation of deep learning models (Microsoft, 2022).

SEAL keeps ciphertexts utilizing its tiered FHE approach, which splits parameters like the encryption modulus, polynomials, and keys into stages that match to the supported circuit depth. SEAL can then be adjusted to handle a specific model complexity.

By performing polynomial operations directly on the encrypted data, SEAL permits homomorphic assessment of neural networks using encrypted data during model inference. The result of not decrypting the input is an encrypted polynomial containing the model's inference output. This project attempts to include end-to-end encryption into the Colab pipeline while leveraging SEAL's speedy FHE implementation. This ensures that when ML insights are retrieved, private data is kept private. Importantly, no decryption takes place on the server. The client decrypts the inference findings locally using the FHE private key.

TenSEAL, a library for performing homomorphic encryption operations on tensors, was then used. TenSEAL, which is built on top of Microsoft SEAL, provides ease of use through a Python API while keeping efficiency by implementing the majority of its activities in C++. Finally, the client can use the predictions from a Colab model at the end (GitHub - OpenMined/TenSEAL: A Library for Doing Homomorphic Encryption Operations on Tensors, n.d.).

A diagram of a file with a key and a lock

Description automatically generated

Testing and Results

To confirm performance across units, integrations, and end-to-end pipeline capabilities under real-world settings, rigorous testing is required. Unit tests employing simulated inputs independently tested the correctness of fundamental building components such as encryption and neural network evaluation algorithms for this solution. Subsystem integration testing revealed that essential components such as Google Colaboratory, Google Drive, and runtime environments supporting homomorphically encrypted models were orchestrated seamlessly.

End-to-end tests demonstrated completeness across the whole pipeline, including encrypted data upload, confidential training, private inferencing, and final client-side result decryption. Due to the technical restrictions of using complete homomorphic encryption, testing used a sampled subset of the standard MNIST digit picture collection. Metrics collected included encryption and decryption latency computation benchmarks, accuracy parity comparisons to plaintext model equivalents, model evaluation overheads, and so forth.

On the encrypted test set, the encrypted model matched the original plaintext version with 99% accuracy. End-to-end confidentiality was maintained by avoiding server-side decryption. However, homomorphic encryption added significant overheads; with encryption enabled, each test batch took more than 2 minutes on a Colab T4 GPU instance. Using this per-batch time as a guide, a full 10,000-image test run would take more than 50 hours, surpassing free resource constraints.

As a result, experiments were scaled down to a 100-image subset, which took 3 minutes in total. Another challenge was the loss of numerical precision caused by the polynomial approximations used in the ciphertext computations. To control error growth, encryption parameters have to be carefully calibrated, trading off additional multipliers for less instability. Framework complexity issues arise when using specific libraries like as SEAL and integrating them into Colab and PyTorch. While these performance and precision problems highlight areas that require further investigation, the prototype successfully showed an encrypted end-to-end machine learning pipeline that provides unprecedented data confidentiality protections. The next steps will be to test on larger datasets, optimize for throughput and latency, improve security through cryptographic agility, and harden production through redundancy and recovery mechanisms. Longer-term possibilities include expanding to fully encrypted model development, which could enable hitherto unthinkable analyses on highly sensitive data - enabling science to benefit society.

|  |  |
| --- | --- |
| Input image | Visual representation of the output layer |
| A black and white image of a number  Description automatically generated | A close-up of a screen  Description automatically generated |
| A black and white pixelated number  Description automatically generated | A close-up of a screen  Description automatically generated |
| A black and white image of a curved object  Description automatically generated | A black and white rectangular object with text  Description automatically generated with medium confidence |
| A black and white image of a string  Description automatically generated | A grey and black rectangular object with text  Description automatically generated with medium confidence |
| A black bird with a white background  Description automatically generated | A close-up of a sensor  Description automatically generated |
| A black and white pixelated number  Description automatically generated | A close-up of a color bar  Description automatically generated |
| A black and white image of a number  Description automatically generated | A black and white image of a sensor  Description automatically generated |
| A black and white image of a number  Description automatically generated | A grey and white rectangular object with black text  Description automatically generated with medium confidence |
| Input image | Visual representation of the output layer |
| A black and white image of a number  Description automatically generated | A close-up of a screen  Description automatically generated |
| A black and white pixelated number  Description automatically generated | A close-up of a screen  Description automatically generated |
| A black and white image of a curved object  Description automatically generated | A black and white rectangular object with text  Description automatically generated with medium confidence |
| A black and white image of a string  Description automatically generated | A grey and black rectangular object with text  Description automatically generated with medium confidence |
| A black bird with a white background  Description automatically generated | A close-up of a sensor  Description automatically generated |
| A black and white pixelated number  Description automatically generated | A close-up of a color bar  Description automatically generated |
| A black and white image of a number  Description automatically generated | A black and white image of a sensor  Description automatically generated |
| A black and white image of a number  Description automatically generated | A grey and white rectangular object with black text  Description automatically generated with medium confidence |

Summary and conclusions

This research project pioneered the incorporation of homomorphic encryption into a cloud-based machine learning pipeline, resulting in a prototype that provides unparalleled security protections for sensitive data while enabling useful insights. The method proved capability for practical confidential inferencing suited to real-time clinical predictions by leveraging the workflow automation strengths of Google Colaboratory and the security of cryptographic libraries such as Microsoft SEAL.

The fundamental architecture entailed orchestrating complex interactions between Colab for scalable data preparation, modeling, training, and deployment, isolated runtime environments for secure execution via isolated environments, and homomorphic encryption powering encoded computations on encrypted data. Encrypting pipeline inputs at the client side, transmitting parameters to provision operations on protected data, evaluating within hardware-isolated enclaves, and returning encrypted outputs to clients for final decryption were all possible because to the modular design.

The project aims to allow regulatory-compliant patient data analytics based on healthcare motives. However, there are opportunities for cross-industry applicability as computational optimizations unlock efficiency, and by investigating fresh crypto methods and improving hardware-assisted confidential computing, this technology has the potential to revolutionize data security across industries. A sampled dataset was used to construct an encrypted MNIST image classification method in the prototype.

On encrypted test data, the combined model outperformed the original plaintext version by 99%.

Overheads, on the other hand, caused challenges: with full encryption, each batch took more than 2 minutes, implying 50+ hours for complete 10,000 sample evaluations, revealing unreasonable inefficiencies that are now restricting practicality. Intricacies of numerical stability, such as managing error explosions through precise parameter setups, also emerged.

However, rather than technology constraints, these performance and precision issues highlight areas for continued optimization study. The solution effectively manifested an end-to-end encrypted machine learning pipeline, giving exceptional protection for sensitive data while overcoming complicated cross-disciplinary orchestration. Commercial-grade hardening for regulatory approval before fielding, greatly improved computational efficiency, enhanced security via cryptographic agility introducing defense-in-depth with algorithm diversity and anti-quantum preparations, demand-focused custom optimizations targeting throughput/latency thresholds, and safety assurance through adversarial simulations and penetration testing are among the next frontiers.

Longer-term opportunities abound by extending confidential computing across the entire machine learning lifecycle - securing dataset accumulation by enabling integration of disparate encrypted sources via protected aggregation, encrypted collaborative model development without information leakage among participants, and encrypted model retraining on live private data feeds - collectively pioneering an environment enabling fully private machine learning.

The project represents significant research advances in solving engineering problems involving cryptography, machine learning, and cloud computing. Transitioning academic discoveries like homomorphic encryption into actual deployed solutions necessitates crossing difficult performance, precision, hardness, and assurance thresholds. . However, the prototype serves as a blueprint: by addressing optimization areas, this pioneering effort could lay the groundwork for a new generation of trustworthy AI systems that maintain regulatory standards for data confidentiality while powering critical analytics - a watershed moment for Privacy Enhancing Technologies that materialize societal benefit.

Finally, this project produced a working design for homomorphically encrypting a cloud machine learning pipeline while adhering to deployability limitations - integrating advanced componentry from various specializations into an integrated architecture. Significant research is required to adapt this discovery into large-scale production systems. Model limits, on the other hand, present opportunities for improvement rather than fundamental impediments. Solving these issues could result in exceptional techno-social progress, enabling secret insights for social welfare, making the promise of this prototype prescient rather than premature.””””

References

Google Colaboratory. <https://research.google.com/colaboratory/>

Bourse, F., Minelli, M., Minihold, M., & Paillier, P. (2018). Fast homomorphic evaluation of deep discretized neural networks. Cryptology ePrint Archive. https://eprint.iacr.org/2018/434.pdf

HomomorphicEncryption.org. (n.d.). Homomorphic encryption. https://homomorphicencryption.org/

Microsoft. (n.d.). Simple Encrypted Arithmetic Library. GitHub. <https://github.com/Microsoft/SEAL>

Microsoft. (2022). SEAL Developer Guide. GitHub. https://github.com/Microsoft/SEAL/raw/master/docs/SEAL\_manual.pdf

NumPy. (n.d.). NumPy documentation. https://numpy.org/doc/stable/user/index.html

*GitHub - OpenMined/TenSEAL: A Library for Doing Homomorphic Encryption Operations on Tensors*. (n.d.). GitHub. Retrieved November 24, 2023, from <https://github.com/OpenMined/TenSEAL.git>

Appendix

# -\*- coding: utf-8 -\*-  
*"""Another copy of 520 project.ipynb  
“  
Automatically generated by Colaboratory.  
  
Original file is located at  
 https://colab.research.google.com/drive/1BfM6WwubCBWVIDJWtgVvTjjeYq93k9Cp  
"""*# “# Commented out IPython magic to ensure Python compatibility.  
import torch  
from torchvision import datasets  
import torchvision.transforms as transforms  
import numpy as np  
import matplotlib.pyplot as plt  
import torchvision  
from torchvision import models, utils  
from torch.autograd import Variable  
  
import scipy.misc  
from PIL import Image  
import json  
# %matplotlib inline  
  
torch.manual\_seed(73)  
  
train\_data = datasets.MNIST('data', train=True, download=True, transform=transforms.ToTensor())  
test\_data = datasets.MNIST('data', train=False, download=True, transform=transforms.ToTensor())  
  
batch\_size = 64  
  
train\_loader = torch.utils.data.DataLoader(train\_data, batch\_size=batch\_size, shuffle=True)  
test\_loader = torch.utils.data.DataLoader(test\_data, batch\_size=batch\_size, shuffle=True)  
  
class ConvNet(torch.nn.Module):  
 def \_\_init\_\_(self, hidden=64, output=10):  
 super(ConvNet, self).\_\_init\_\_()  
 self.conv1 = torch.nn.Conv2d(1, 4, kernel\_size=7, padding=0, stride=3)  
 self.fc1 = torch.nn.Linear(256, hidden)  
 self.fc2 = torch.nn.Linear(hidden, output)  
  
 def forward(self, x):  
 x = self.conv1(x)  
 # the model uses the square activation function  
 x = x \* x  
 # flattening while keeping the batch axis  
 x = x.view(-1, 256)  
 x = self.fc1(x)  
 x = x \* x  
 x = self.fc2(x)  
 return x  
  
def train(model, train\_loader, criterion, optimizer, n\_epochs=10):  
 # model in training mode  
 model.train()  
 for epoch in range(1, n\_epochs+1):  
  
 train\_loss = 0.0  
 for data, target in train\_loader:  
 optimizer.zero\_grad()  
 output = model(data)  
 loss = criterion(output, target)  
 loss.backward()  
 optimizer.step()  
 train\_loss += loss.item()  
  
 # calculate average losses  
 train\_loss = train\_loss / len(train\_loader)  
  
 print('Epoch: {} \tTraining Loss: {:.6f}'.format(epoch, train\_loss))  
  
 # model in evaluation mode  
 model.eval()  
 return model  
  
model = ConvNet()  
criterion = torch.nn.CrossEntropyLoss()  
optimizer = torch.optim.Adam(model.parameters(), lr=0.001)  
model = train(model, train\_loader, criterion, optimizer, 10)

#”  
  
"""# Test the model  
  
"""  
# “  
#printing the lables  
print(test\_data.targets)  
  
def test(model, test\_loader, criterion):  
 # initialize lists to monitor test loss and accuracy  
 test\_loss = 0.0  
 class\_correct = list(0. for i in range(10))  
 class\_total = list(0. for i in range(10))  
  
 # model in evaluation mode  
 model.eval()  
  
 for data, target in test\_loader:  
 output = model(data)  
 loss = criterion(output, target)  
 test\_loss += loss.item()  
 # convert output probabilities to predicted class  
 \_, pred = torch.max(output, 1)  
 # compare predictions to true label  
 correct = np.squeeze(pred.eq(target.data.view\_as(pred)))  
 # calculate test accuracy for each object class  
 for i in range(len(target)):  
 label = target.data[i]  
 class\_correct[label] += correct[i].item()  
 class\_total[label] += 1  
  
 # calculate and print avg test loss  
 test\_loss = test\_loss/len(test\_loader)  
 print(f'Test Loss: {test\_loss:.6f}\n')  
  
 for label in range(10):  
 print(  
 f'Test Accuracy of {label}: {int(100 \* class\_correct[label] / class\_total[label])}% '  
 f'({int(np.sum(class\_correct[label]))}/{int(np.sum(class\_total[label]))})'  
 )  
  
 print(  
 f'\nTest Accuracy (Overall): {int(100 \* np.sum(class\_correct) / np.sum(class\_total))}% '  
 f'({int(np.sum(class\_correct))}/{int(np.sum(class\_total))})'  
 )  
  
test(model, test\_loader, criterion)  
  
!pip install tenseal  
import tenseal as ts  
  
#”  
“ “ “  
It's a PyTorch-like model using operations implemented in TenSEAL.  
 - .mm() method is doing the vector-matrix multiplication explained above.  
 - you can use + operator to add a plain vector as a bias.  
 - .conv2d\_im2col() method is doing a single convolution operation.  
 - .square\_() just square the encrypted vector inplace.  
” ” ”  
 # “  
class EncConvNet:  
 def \_\_init\_\_(self, torch\_nn):  
 self.conv1\_weight = torch\_nn.conv1.weight.data.view(  
 torch\_nn.conv1.out\_channels, torch\_nn.conv1.kernel\_size[0],  
 torch\_nn.conv1.kernel\_size[1]  
 ).tolist()  
 self.conv1\_bias = torch\_nn.conv1.bias.data.tolist()  
  
 self.fc1\_weight = torch\_nn.fc1.weight.T.data.tolist()  
 self.fc1\_bias = torch\_nn.fc1.bias.data.tolist()  
  
 self.fc2\_weight = torch\_nn.fc2.weight.T.data.tolist()  
 self.fc2\_bias = torch\_nn.fc2.bias.data.tolist()  
  
  
  
  
  
 def forward(self, enc\_x, windows\_nb):  
 # conv layer  
 enc\_channels = []  
 for kernel, bias in zip(self.conv1\_weight, self.conv1\_bias):  
 y = enc\_x.conv2d\_im2col(kernel, windows\_nb) + bias  
 enc\_channels.append(y)  
#”

# “  
  
  
 # pack all channels into a single flattened vector  
 enc\_x = ts.CKKSVector.pack\_vectors(enc\_channels)  
  
 # square activation  
 enc\_x.square\_()  
 # fc1 layer  
 enc\_x = enc\_x.mm(self.fc1\_weight) + self.fc1\_bias  
 # square activation  
 enc\_x.square\_()  
 # fc2 layer  
 enc\_x = enc\_x.mm(self.fc2\_weight) + self.fc2\_bias  
 return enc\_x  
  
 def \_\_call\_\_(self, \*args, \*\*kwargs):  
 return self.forward(\*args, \*\*kwargs)

#”

# “  
  
def enc\_test(context, model, test\_loader, criterion, kernel\_shape, stride):  
 # initialize lists to monitor test loss and accuracy  
 test\_loss = 0.0  
 class\_correct = list(0. for i in range(10))  
 class\_total = list(0. for i in range(10))  
  
 for data, target in test\_loader:  
 # Encoding and encryption  
 x\_enc, windows\_nb = ts.im2col\_encoding(  
 context, data.view(28, 28).tolist(), kernel\_shape[0],  
 kernel\_shape[1], stride  
 )  
  
 fig=plt.figure(figsize=(15,15))  
 fig.subplots\_adjust(left=0,right=1,bottom=0,top=1,hspace=0.05,wspace=0.05)  
 images = np.reshape(data[0], [28,28])  
 ax=fig.add\_subplot(6,5,1,xticks=[],yticks=[])  
 ax.imshow(images,cmap=plt.cm.binary,interpolation='nearest')  
 ax.text(0,7,"Input image: "+str(target))  
 # Encrypted evaluation  
 enc\_output = enc\_model(x\_enc, windows\_nb)  
 # Decryption of result  
 output = enc\_output.decrypt()  
 output = torch.tensor(output).view(1, -1)  
 fig=plt.figure(figsize=(15,7))  
fig.subplots\_adjust(left=0,right=1,bottom=0,top=1,hspace=0.05,wspace=0.05)  
 images = np.reshape(output[0], [1,-1])  
 ax=fig.add\_subplot(27,6,1,xticks=[],yticks=[])  
 ax.imshow(images,cmap=plt.cm.binary,interpolation='nearest')  
 ax.text(0,7,"Output image: "+str(target))  
  
 # compute loss  
 loss = criterion(output, target)  
 test\_loss += loss.item()  
  
 # convert output probabilities to predicted class  
 \_, pred = torch.max(output, 1)  
  
 # compare predictions to true label  
 correct = np.squeeze(pred.eq(target.data.view\_as(pred)))  
 # calculate test accuracy for each object class  
 label = target.data[0]  
 class\_correct[label] += correct.item()  
 class\_total[label] += 1  
#”  
# “  
  
 # calculate and print avg test loss  
 test\_loss = test\_loss / sum(class\_total)  
 print(f'Test Loss: {test\_loss:.6f}\n')  
 for label in range(10):  
 if class\_total[label] !=0:  
 print(  
 f'Test Accuracy of {label}: {int(100 \* class\_correct[label] / class\_total[label])}% '  
 f'({int(np.sum(class\_correct[label]))}/{int(np.sum(class\_total[label]))})'  
 )  
 print(class\_correct,class\_total)  
 if class\_total !=0:  
 print(  
 f'\nTest Accuracy (Overall): {int(100 \* np.sum(class\_correct) / np.sum(class\_total))}% '  
 f'({int(np.sum(class\_correct))}/{int(np.sum(class\_total))})'  
 )

#”  
# “  
  
## Encryption Parameters  
  
# controls precision of the fractional part  
bits\_scale = 26  
  
# Create TenSEAL context  
context = ts.context(  
 ts.SCHEME\_TYPE.CKKS,  
 poly\_modulus\_degree=8192,  
 coeff\_mod\_bit\_sizes=[31, bits\_scale, bits\_scale, bits\_scale, bits\_scale, bits\_scale, bits\_scale, 31]  
)  
  
# set the scale  
context.global\_scale = pow(2, bits\_scale)  
  
# galois keys are required to do ciphertext rotations  
context.generate\_galois\_keys()  
  
from torch.utils.data import Subset  
from torch.utils.data import DataLoader

#”  
# “

fraction = 0.01 # Specify the fraction you want to use (e.g., 0.5 for 50% of the dataset)  
num\_samples = int(len(test\_data) \* fraction)  
test\_data\_subset = Subset(test\_data, range(num\_samples))  
  
test\_loader = DataLoader(test\_data\_subset, batch\_size=1,shuffle=True)  
kernel\_shape = model.conv1.kernel\_size  
stride = model.conv1.stride[0]  
  
  
enc\_model = EncConvNet(model)  
enc\_test(context, enc\_model, test\_loader, criterion, kernel\_shape, stride)

#”