CryptGPU

Amithabh A
112101004

May 28, 2024

System Overview

- Data and model are (arbitrarily) partitioned across three parties
- Clients first secret share their inputs to three independent cloud-service providers. They run the cryptographic protocol on secret-shared inputs.
- CryptGPU's protocols provide security against a single semi-honest corruptionl

Background

CryptGPU adapt the basic architecture of CrypTen, and make modifications to support three-party protocols based on replicated secret sharing

CrypTEN

- CrypTEN provide a secure computing back end for PyTorch while still preserving the PyTorch front end APIs
- MPCTensor in CrpyTEN, functions like a standard PyTorch tensor, except values are secret shared across multiple machines. Internally, CRYPTEN uses n-out-of-n additive secret sharing.
- For Bilinear operations(Matrix Mulitplication and convolutions) CrypTEN uses arithmetic secret sharing over a large ring (e.g., $\mathbb{Z}_{2^{64}}$)

Floating point computations

- cryptographic core of CRYPTGPU relies on (additive) replicated secret sharing over the 64-bit ring $\mathbb{Z}_{2^{64}}$.
- The goal is to take advantage of the GPU to accelerate each party's local computation on their individual shares.
- Embed the ring operations over $\mathbb{Z}_{2^{64}}$ into 64-bit floating point operations.

Integer operations using floating-point arithmetic

Exact computation for small values

- 64-bit floating pt. vals have 52 bits of precision - can exactly represent all integers in the interval $[2^{52}, 2^{52}]$.
- $\forall a, b \in \mathbb{Z}_{2^{64}}[2^{26}, 2^{26}]$, we can compute the product ab using their floating-point representations

Bilinearity

- Matrix Multiplication and convolution are bilinear. i.e, $(A1 + A2) \circ (B1 + B2) = A1 \circ B1 + A2 \circ B1 + A2 \circ B1 + A2 \circ B2$

Evaluation of Bilinear operation

- CryptGPU decomposes each inputs **A**,**B** $\in \mathbb{Z}_{2^{64}}^{n \times m}$ into smaller inputs A_1, ..., A_k and B_1, ..., B_k where A $= \Sigma_{i=1}^{k} 2^{(i-1)w} A_i$ and computes k^2 products $A_I \circ B_j$
- As long as the entries of $A_i \circ$B$_j$ do not exceed 2^{52} in magnitude, all of the above pairwise products are computed as long as $A_i \circ B_j$ do not exceed 2^{52} in magnitude.
- CryptGPU decomposes each input into k $= 4$ blocks when performing computations using floating-point kernels, where values in each block are represented by w=16-bit value.

Semi Honest Security

Let $f : (0, 1^n)^3 \rightarrow (0, 1^m)^3$ be a randomized functionality.

.

Let π be a protocol.

.

π securely computes f in presence of single semi-honest corruption if there exists an efficient simulator S such that for every corrupted party $i \in \{1, 2, 3\}$ and every input $\mathbf{x} \in (0, 1^n)^3$,
$\{output^{\pi}(\mathbf{x}), view_i^{\pi}\} \approx^c \{f(\mathbf{x}), S(i, x_i, f_i(\mathbf{x}))\}$

Computing on secret-shared values

- Two main settings : private inference and private training
- Two secret sharing : standard 3-out-of-3 additive secret sharing and 2-out-of-3 replicated secret sharing
- Modelled both types of secret sharing as pair of algorithms (Share, Reconstruct)

Computing on secret-shared values

Properties of algorithmic pair (Share, Reconstruct)

- Share$(x) = (x_1, x_2, x_3)$
- Reconstruct(S) takes set of shares. If successful, it outputs $x \in \{0,1\}^n$, and returns $\perp$ otherwise.

Private Inference

- $Eval(M, x)$:Problem of evaluating a trained model M on an input x
- Ideal functionality f maps secret shares of an input x and a model M to a secret share of the output $Eval(M, x)$.
- I/p : $((M_1, x_1), (M_2, x_2), (M_3, x_3))$
- Ideal functionality outputs $Share(Eval(M, x))$ where
  - $M \leftarrow \text{Reconstruct}(M_1, M_2, M_3)$
  - $x \leftarrow \text{Reconstruct}(x_1, x_2, x_3)$

Private Training

- Goal is to run a training algorithm Train on some dataset D
- Ideal functionality f maps secret shares of the dataset $(D1, D2, D3)$ to a secret share of the model $Share(Train(D))$ where
  - D $\leftarrow$ Reconstruct$(D1, D2, D3)$.

Secret sharing

- We work over the ring $\mathbb{Z}_n$ where $n = 2^k$, $k = 2^{64}$
- To secret share $x \in \mathbb{Z}_n$, sample shares $x_1, x_2, x_3 \xleftarrow{R} \mathbb{Z}_n$ such that $x_1 + x_2 + x_3 = x$.
- Default sharing : 2-out-of-3 replicated secret sharing where each party hold a pair of shares: P_i has (x_i, x_{i+1}). It is denoted by $[[x]]^n = (x_1, x_2, x_3)$
- 3-out-of-3 additive secret sharing scheme is used sometimes where party P_i holds x_i

Fixed point Representation

- t : no of bits of precision
- A real value $x \in \mathbb{R}$ is represented by the integer $\lfloor x \cdot 2^t \rceil$ (i.e., the nearest integer to $x \cdot 2^t$)
- Ring modulus n is chosen to ensure no overflow on integer-valued fixed-point operations (CryptGPU sets $n = 64$)

Protocol Initialization(I didn't understand this quite lot)

- Assumption : parties have many independent secret shares of 0.
- F be a pseudorandom function (PRF)
- Each party P_i samples a key k_i and sends it to party P_{i+1}.
- jth secret share of 0 is the triple (z_1, z_2, z_3) where $z_i = F(k_i, j) - F(k_{i-1}, j)$

Linear Operations

- α, β, γ are public constants
- $[[x]]^n$ and $[[y]]^n$ are secret shared values.
- $[[\alpha x + \beta y + \gamma]]^n = (\alpha x_1 + \beta y_1 + \gamma, \alpha x_2 + \beta y_2, \alpha x_3 + \beta y_3)$

Multiplication(I didn't understand replicated shares and truncation)

- Multiplication of two secret shared values $[[x]]^n = (x_1, x_2, x_3)$ and $[[y]]^n = (y_1, y_2, y_3)$
- Each party locally computes $z_i = x_i y_i + x_{i+1} y_i + x_i y_{i+1}$
- To obtain replicated shares of z, party P_i sends P_{i+1} a blinded share $z_i + \alpha_i$, where $(\alpha_1, \alpha_2, \alpha_3)$ is a fresh secret sharing of 0.

ReLU Activation Function

- $ReLU(x) := max(x, 0)$
- Computation of ReLU function corresponds to computating $msb(x)$ of x

C. Additional Building Blocks for Private Training

- Augmenting existing toolkit with several additional protocols.
- Standard backward Propagation setting with a softmax/cross entropy loss function optimized using Stochastic Gradient Descent(SGD)

Summary of Work :

- Classification tasks with d target classes
- Each iteration of SGD takes
  - input $x \in \mathbb{R}^m$
  - One-hot encoding of target vector $y \in \{0,1\}^d$, $y_i = 1$ if x belong to class i and $y_i = 0$ otherwise.
- Cross entropy computation :
  $l_{CE}(\mathbf{x}; \mathbf{y}) := -\Sigma_{i \in [d]}\ y_i \log \widetilde{z}_i$
  where $\widetilde{z} \leftarrow \text{softmax}(\mathbf{z})$, $\mathbf{z} \leftarrow \text{Eval}(M, x)$ and M is the current model

C. Additional Building Blocks for Private Training

- For a vector $x \in \mathbb{R}^d$, softmax function $\text{softmax}(\mathbf{x}_i) := \frac{e^{x_i}}{\sum_{i \in [d]} e^{x_i}}$
- Gradient of l_{CE} for output layer z is $\nabla_{\mathbf{z}} l_{CE} = \text{softmax}(\mathbf{z})$ - $\mathbf{y}$

System Implementation and Evaluation

Point-to-point communication

- default communication mode in PyTorch : **Broadcast** mode
- Broadcast channel between each pair of parties functions as a **point-to-point channel** between the parties.

System Implementation and Evaluation

Point-to-point communication

- default communication mode in PyTorch : **Broadcast** mode
- Broadcast channel between each pair of parties functions as a **point-to-point channel** between the parties.

Pseudorandom generators on the GPU

Paper uses **AES** as PRF in their protocol, by using `torchcsprng` PyTorch C++/CUDA extension.

Deep Learning Datasets

- **MNIST**
  - dataset for handwritten digit recognition.
  - benchmark in many privacy-preserving ML systems
- **CIFAR 10**
  - Dataset with 60,000 32x32 RGB images split evenly across 10 classes.
- **Tiny ImageNet**
  - Modified subset of the ImageNet dataset
  - 100,000 64 x 64 RGB training images, 10,000 testing images, split across 200 classes.
  - More challenging than CIFAR 10
- **ImageNet**
  - Large Scale Visual recognition Dataset
  - Standard benchmark for evaluating the classification performance of computer vision models
  - 1000 classes,
  - CrypTFlow is Tthe only prior system for privacy-preserving machine learning that demonstrates performance at scale of ImageNet.

Deep Learning Models

- **LeNet**
  - Handwritten digit recognition
  - Shallow network with 2 convolutional layers, 2 average pooling layers, and 2 fully connected layers
  - Uses the hyperbolic tangent (tanh) as its activation function.
- **AlexNet**
  - 5 convolutional layers, 3 max pooling layers, and 2 fully connected layers for a total of 61 million parameters
  - Uses ReLU as its activation function.
- **VCG-16**
  - Uses 16 layers consisting of convolution, ReLU, max pooling, and fully-connected layers
- **ResNet**
  - Introduces skip-connections that addresses the vanishing gradient problem when training deep neural network models.
  - Enjoyed wide adoption in the computer vision community.

Architectural Adjustments

AlexNet and VGG-16 on small datasets

AlexNet and VGG-16 are not directly compatible with smaller inputs since they were designed for ImageNet. So the architecture is modified.

- **For AlexNet**
  - Drop the final max pooling layer for CIFAR-10
  - Adjust no. of neurons to 256-256-10 for CIFAR-10
  - Adjust no. of neurons to 1024- 1024-200 for Tiny ImageNet.
- **For VCG-16**
  - Adjust no. of neurons to 256-256-10 for CIFAR-10
  - Adjust no. of neurons to 512- 512-200 for Tiny ImageNet
- **Average pooling**
  - Uses 16 layers consisting of convolution, ReLU, max pooling, and fully-connected layers

Architectural Adjustments

Activation Functions

- LeNet uses the hyperbolic tangent function tanh
- CryptGPU does not support evaluating the tanh function and modern networks primarily use ReLU as their activation function.
- They replaced tanh with ReLU in their experiments with LeNet.

Architectural Adjustments

Average Pooling

- Pooling : standard way to down-sample outputs of conv. layers in CNN.
- Pooling layer accumulates o/p of conv. layers by replacing feature map window with average of values(avg pooling) / max of values(max pooling)
- Avg Pooling is a linear operation whereas max pooling is a highly non-linear operation
- will talk more later.

Private Inference

- Falcon outperforms CryptGPU for shallow networks and small datasets
- CryptGPU is able to perform private inference over the ResNet-152 network 2.2x faster than CrypTFlow
- Compared to plaintext inference on the GPU, there still remains a significant 1000x gap in performance

Batch Private Inference

- Leveraging GPU parallelism to process a batch of images will amortize the cost of private inference.
- Add pictures and explain

Private Training

- GPUs to have a larger advantage in the setting of private training
- CryptGPU achieves a considerable speedup over Falcon, especially over larger models and datasets
- A Single iteration of (private) backpropagation completes in 11.30s with CRYPTGPU and 6.9 minutes using FALCON to train AlexNet on Tiny ImageNet set.
- Privately training AlexNet on Tiny ImageNet takes 10 days using CryptGPU while it would takes 375 days using Falcon (assuming 100 epochs over the training set).
- There still remains a large gap (roughly 2000x) between the costs of private training and plaintext training (on the GPU)

Private Training breakdown : Comparison between Falcon(CPU based Protocol) and CryptGPU(GPU based Protcol)

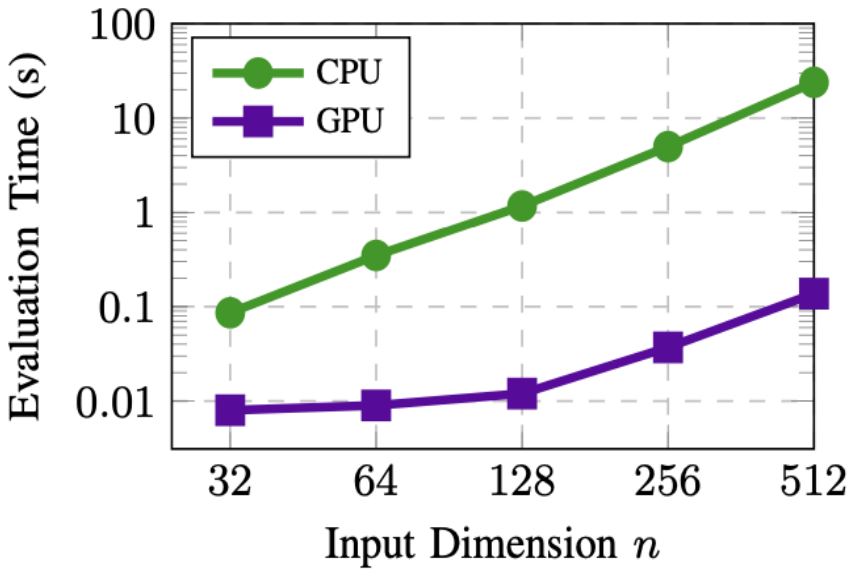
- Computation of the linear layers in CryptGPU between 25x and 70x faster than Falcon
- For Pooling layers, Falcon uses Max Pooling and CryptGPU uses Avg Pooling. CryptGPU maintains a (significant) performance edge even if we exclude the cost of the pooling layers from the running time of FALCON.
- For ReLU layers, CPU-based protocol in Falcon compares very favorably with ReLU protocol in CRYPTGPU. The Protocol performance will improve by having ReLU protocol that takes advantage of GPU parallelism.

Comparing Falcon and CryptGPU

	Linear		Pooling		ReLU		Softmax	
	FALCON	CRYPTGPU	FALCON	CRYPTGPU	FALCON	CRYPTGPU	FALCON	CRYPTGPU
LeNet (MNIST)	13.07	0.49	1.34	0.076	0.47	1.00	—	0.53
AlexNet (CIFAR)	59.23	0.86	2.65	0.077	0.41	1.33	—	0.55
VGG-16 (CIFAR)*	355.16	6.33	2.86	0.21	5.40	4.74	—	0.53
AlexNet (TI)	402.45	5.60	10.20	0.37	1.92	4.16	—	1.04
VGG-16 (TI)†	355.84	7.61	2.87	0.32	5.37	4.73	—	0.98

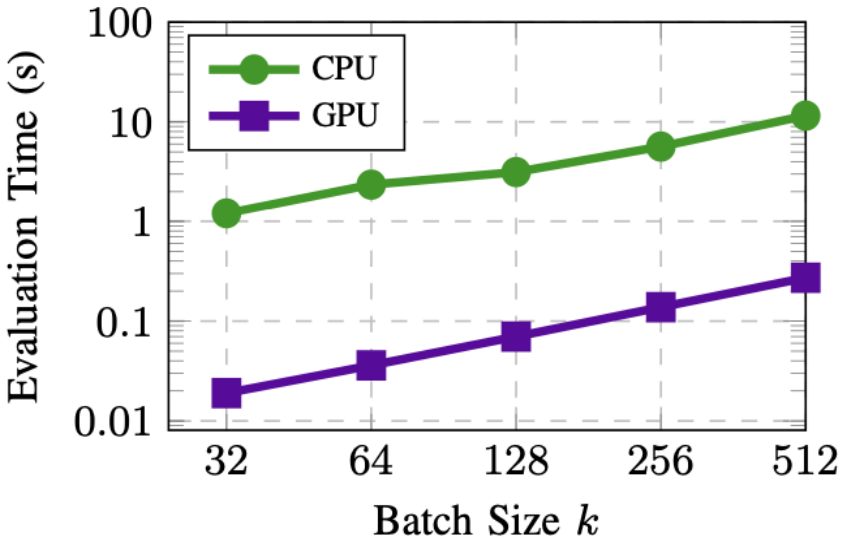
- Runtime (in seconds) of Falcon and CryptGPU for evaluating the linear, pooling, ReLU, and softmax layers for different models and datasets during private training.
- Linear layers include the convolution and the fully-connected layers.
- Pooling layer refers to max pooling in Falcon, and avg. pooling in CryptGPU.
- Implementation of Falcon donot support softmax evaluation (and correspondingly, gradient computation for the output layer).

Comparing Falcon and CryptGPU



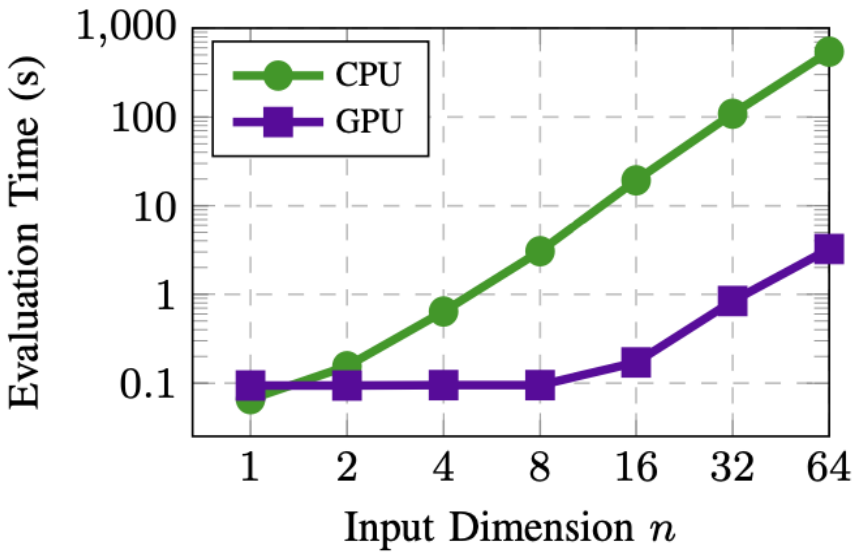
Convolution on an $n \times n \times 3$ input with a 11×11 kernel, 64 output channels, 4×4 stride, and 2×2 padding

Comparing Falcon and CryptGPU



Convolution on batch of k $32 \times 32 \times 3$ inputs with an 11×11 kernel, 64 output channels, 4 4 stride, and 2×2 padding.

Comparing Falcon and CryptGPU



Convolution on an n×n×512 input with a 3×3 kernel, 512 output channels, 1×1 stride, and 1×1 padding.

Private ReLU: GPU vs. CPU

- Delphi executed the non-linear steps (e.g., ReLU computations) on the CPU. CryptGPU takes advantage of GPU parallelism to accelerate non-linear computations with chosen set of cryptographic protocols.
- CryptGPU has 16 speedup when evaluating ReLU on a block of 256,000 secret shared inputs
- As inputs scale up to a block with 32 million inputs (250 MB of data), there is a 9 speedup on the GPU

Average Pooling vs Max pooling

Attempt to evaluate whether the choice of pooling makes a significant difference on model performance.

- Trained AlexNet and VGG-16 networks over CIFAR-10 dataset where all max pooling layers is replaced with average pooling layers
- 3% drop in accuracy (from 76% to 73%) for AlexNet and a 1% increase in accuracy with VGG-16 (from 82% to 83%)
- Average pooling in place of Max pooling donot make degradation of model performance significantly.