- SIRNN: A Math Library for Secure RNN Inference
  - Part 0: Overview
    - Background
    - Goal
    - Overall System
    - Motivation by Industry practices
    - Application Senarios
    - Challenges
    - Contributions
    - Components' Hierarchy
    - Potential Improvements
  - Part 1: Notations
  - Part 2: Extension
    - 2.1 Zero extension
    - 2.2 Signed Extension
  - Part 3: Truncation
    - 3.1 Logial Right Shift:
    - 3.2 Arithmetic Right Shift: & Truncate and Reduce
    - 3.3 Division by power-of-2
  - Part 4: Multiplication with non-uniform bitwidths
    - 4.1 Unsigned Mutiplication
    - 4.2 Signed Multiplication
    - 4.3 Matrix Multiplication and Convolution
  - Part5: Digit Decomposition & MSNZB
    - 5.1 Digit Decomposition
    - 5.2 Most Significant Non-Zero Bit (MSNZB)
  - Part6: MSB to Wrap Optimization
  - Part7: Math Library
    - 7.1 Expotential
    - 7.2 Sigmoid and Tanh
    - 7.3 Reciprocal of Square Root
    - 7.4 Evaluation

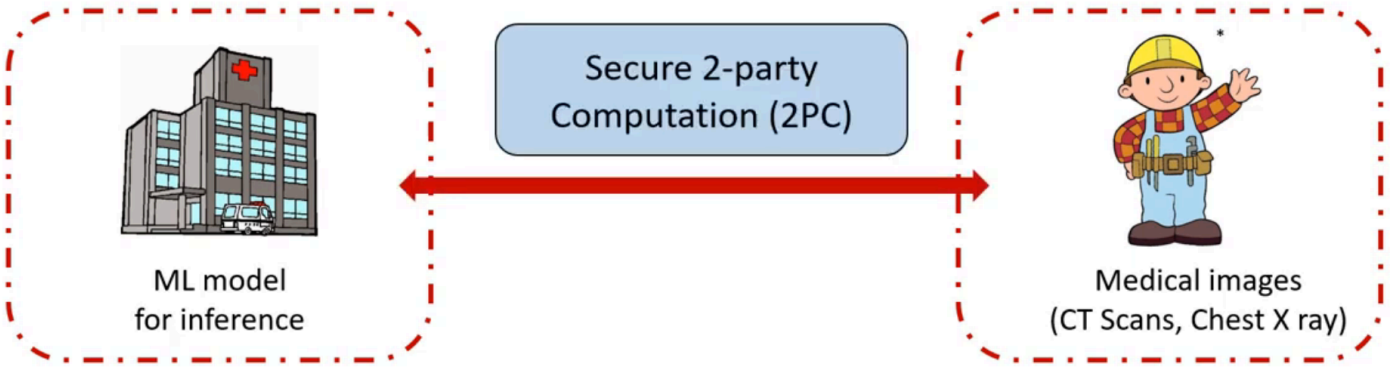
SIRNN: A Math Library for Secure RNN Inference

this is published at Oakland'21

Part 0: Overview

Background

Secure inference in PPML, which is a typical situation for Secure 2-Party Computation(2PC)



Hospital should learn no information about client's data

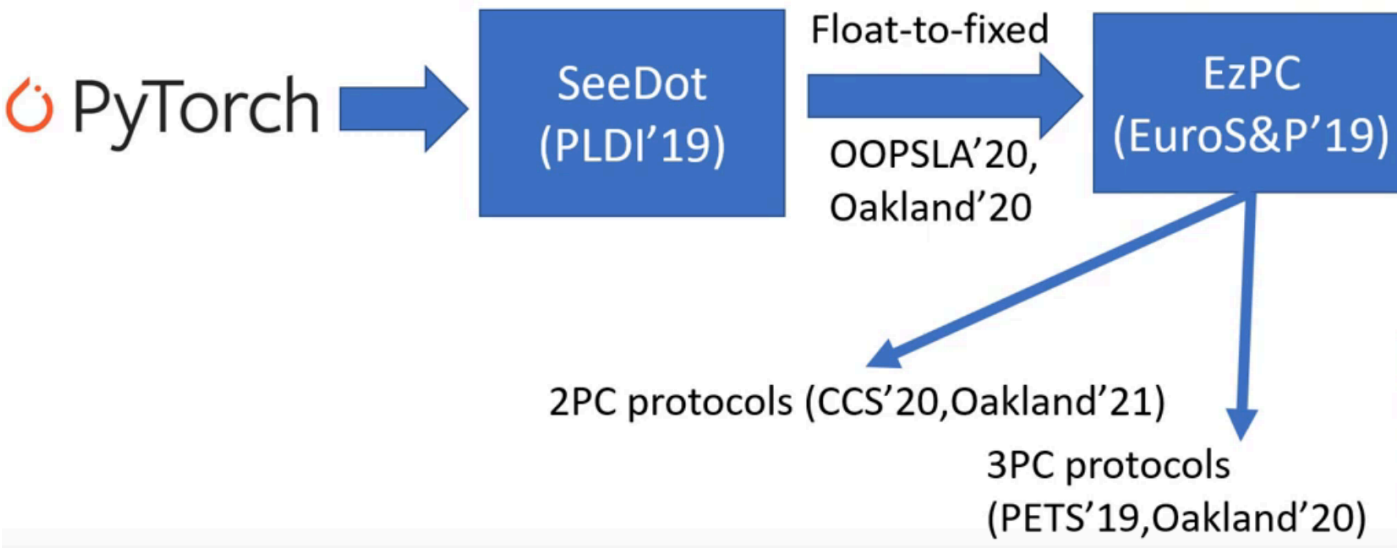
Bob: should only learn model output

Goal

Push-button solution to run secure inference of arbitrary ML models

Overall System

Poster: Multi institution AI validation without data sharing

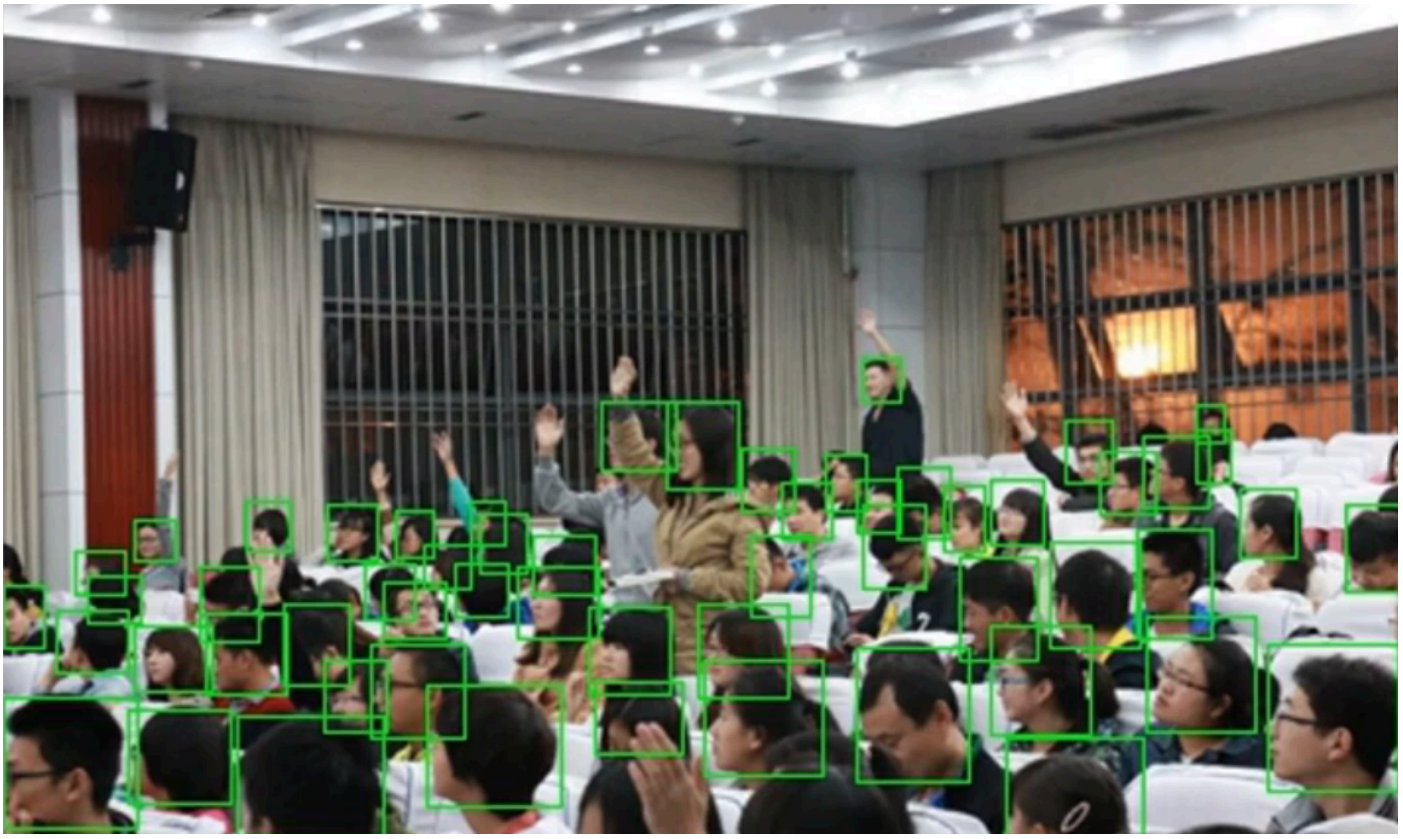


Motivation by Industry practices

1. Secure inference accuracy should match insecure inference
2. Retraining a more 2PC-friendly model is not an option
    - The model training team is disjoint from the model deployment team
    - Getting permission to onboard training dataset is a long lengthy legal process
3. Minimize total runtime per inference

Application Senarios

Head detection in images: combine CNNs and RNNs to go beyond classification



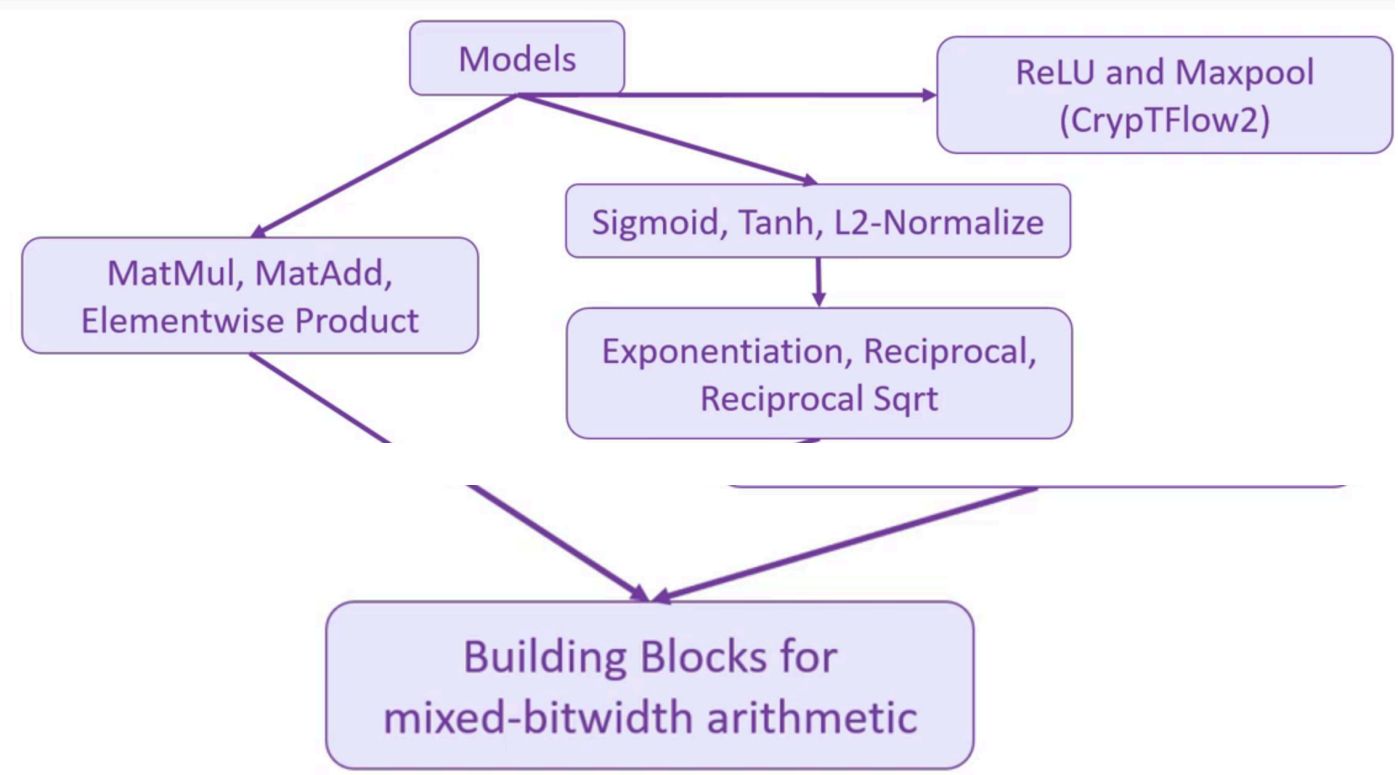
Challenges

- Uses floating-point arithmetic
  - With current techniques, secure floating-point is inefficient
  - Solution: float-to-fixed converters
- Running fixed-point code securely has 2 challenges
  1. Use of math functions
     - Accurate math functions require high bitwidths
  2. Use of mixed-bitwidths
     - All prior works use uniform bitwidth
     - Uniform bitwidth pays the cost of largest bitwidth everywhere
     - Performance depends cirtically on bitwidths
       - degrades at least linearly; quadratically for mutiplication

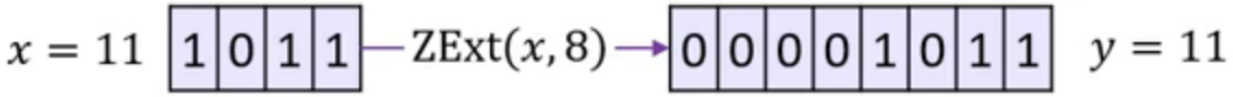
Contributions

- Create SIRNN, a library for semi-honest secure inference
- Support for mixed-bitwidth arithmetic
  - New 2PC building blocks
- Support for math functions
  - New math implementations for sigmoid/tanh, exponentiation, and reciprocal sqrt
  - Efficient to implement using those building blocks
  - Provably precise
- No loss in inference accuracy
- First to run RNNs securely on speech data and head detection in images
  - SIRNN is 2 magnitude faster than prior work

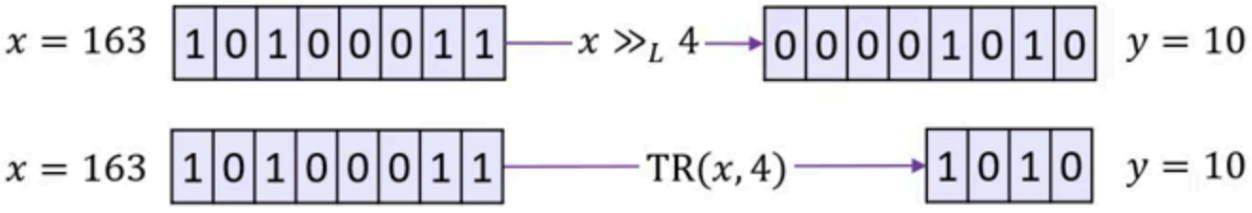
Components' Hierarchy



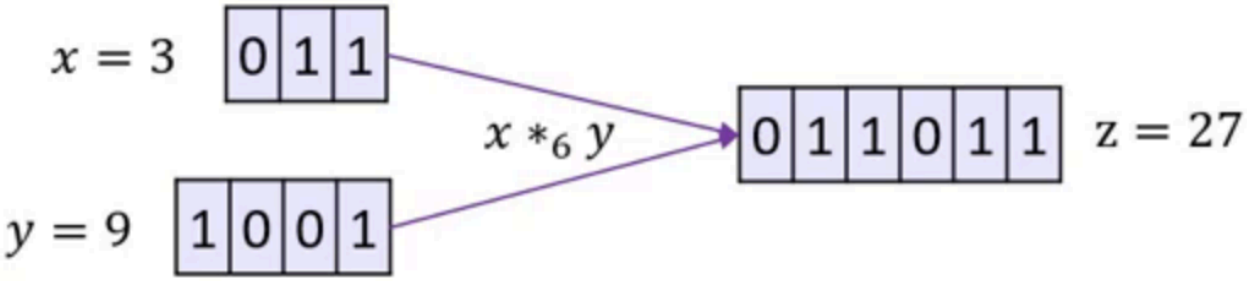
- Extension: y=x
  - Zero Extension: y = ZExt(x,n)



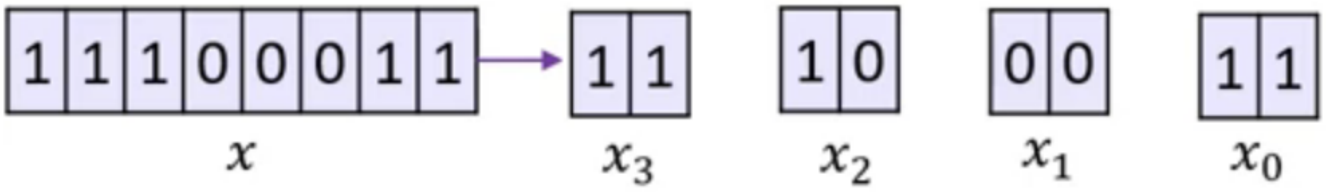
- Truncation: $y = \lfloor \frac{x}{2^s} \rfloor$
  - Logical Right-shift: $y = x >>_L s$
  - Truncate-Reduce: $y = TR(x, s)$



- Mutiplication: $z = x \cdot y \bmod 2^l$
  - Unsigned Mutilication: $z = x *_l y$



- Digit Decomposition: $x = x_{l/d-1} \mid \mid \ldots \mid \mid x_0,\ x_i \in \{0,1\}^d$



- Performance:

Operation	Max Comm. Improvement
Extension	**58** ×
Right-Shift	**8.75** ×
Truncate-Reduce	**17.5** ×
Multiplication	**5** ×
Digit-Decomposition	**5.5** ×

Fixed-point code is mixture of 8/16/32/64-bits

- Uniform bitwidth 2PC is about 3 times slower
- Half time spent in sigmoid/tanh

Potential Improvements

- Post-training quantization (float-to-fixed) is brittle
- Limited to security against semi-honest adversaries
- Large inference tasks exceed available RAM

Part 1: Notations

- 1{b}: return 1 when b is true
- $\zeta_{l,m}(x)$: for $x \in Z_L$, map x to Z_M, $m \geq l$
- int(x) and uint(x): for $x \in Z_L$, refer to the sign and unsign values in Z, $int(x) = uint(x) - MSB(x) \cdot L$: the upper half range is negative, the lower half range is positive.
- $F^l_{Mill}(x,y) : \langle z \rangle^B = 1\{x < y\}^a$;
- $F^l_{Wrap}(x,y) = F^l_{Mill}(L-1-x,y) :\ w = wrap(x,y,L)$
- $e = 1\{(x+y \mod L) = L-1\}$, check whether x+y is all 1s string on the ring
- $F^l_{Wrap\,\&All1s}(x,y) = (\langle w \rangle^B \mid \mid \langle e \rangle^B)$, this is the combo of the former 2, return a 2-bit string. *There 's no possibility for (11)*
- $L = 2^l,\ M = 2^m,\ N = 2^n$

$$x < y = (x_0 < y_0) \vee ((x_0 = y_0) \wedge (x_1 < y_1)),\ x = x_0 \mid \mid x_1,\ y = y_0 \mid \mid y_1$$

Part 2: Extension

2.1 Zero extension

$$F_{ZExt}^{m,n} : y = ZExt(x, m, n) \in Z_N \text{ with } uint(y) = uint(x),\ n > m$$

- $\langle x \rangle^m \in Z_M$, we have ***equation (1)***

  $$x = \langle x \rangle_0^N + \langle x \rangle_1^N - w \cdot M$$

  where w is in $\langle \cdot \rangle^B$;

- $F_{B2A}^{n-m}(\langle w \rangle^B) = \langle w \rangle^{n-m} \in Z_{2^{n-m}}$;
  - $w = \langle w \rangle_0^{n-m} + \langle w \rangle_1^{n-m} - 2^{n-m} \cdot\ wrap(\langle w \rangle_0^{n-m},\ \langle w \rangle_1^{n-m},\ 2^{n-m})$;
  - $M *_n w = M *_n (\langle w \rangle_0^{n-m} + \langle w \rangle_1^{n-m})$, where $x *_n y = x \cdot y \bmod N$;

- Let

  $$y = \sum_{b=0}^{1} (\langle x \rangle_b^m - M \cdot \langle w \rangle_b^{n-m}) \bmod N$$

```
we have$x\ mod\ N = y$;

The ring conversion for the share of x can be easily done locally, however, the convertion for w can be optimized by not converting to n ring, but converting to n-m ring instead, which is the same in terms of math and saves communication expense.
```

2.2 Signed Extension

$$F_{SExt}^{m,n} : y = SExt(x, m, n) \in Z_N \text{ with } int(y) = int(x),\ n > m$$

$$int(x) = uint(x') - 2^{m-1} \text{ for } x' = x + 2^{m-1} \bmod M$$

Proof

- $x < 2^{m-1}$: $x + 2^{m-1} < M$, we thus have $x' = x + 2^{m-1}$ in Z.
  - Therefore, $int(x) = x = uint(x') - 2^{m-1}$
- $x \geq 2^{m-1}$: $x + 2^{m-1} \leq M$, we thus have
  - $x' = x + 2^{m-1} - 2^m = x - 2^{m-1}$ *in* Z, since $int(x) = x - 2^m = (x - 2^{m-1}) - 2^{m-1} = x' - 2^{m-1}$ *in* Z, we have $int(x) = uint(x') - 2^{m-1}$

Thus, we implement $F_{SExt}^{m,n}$ based on $F_{ZExt}^{m,n}$ as $F_{SExt}^{m,n}(x) = F_{ZExt}^{m,n}(x') - 2^{m-1}$ without increase communication spense

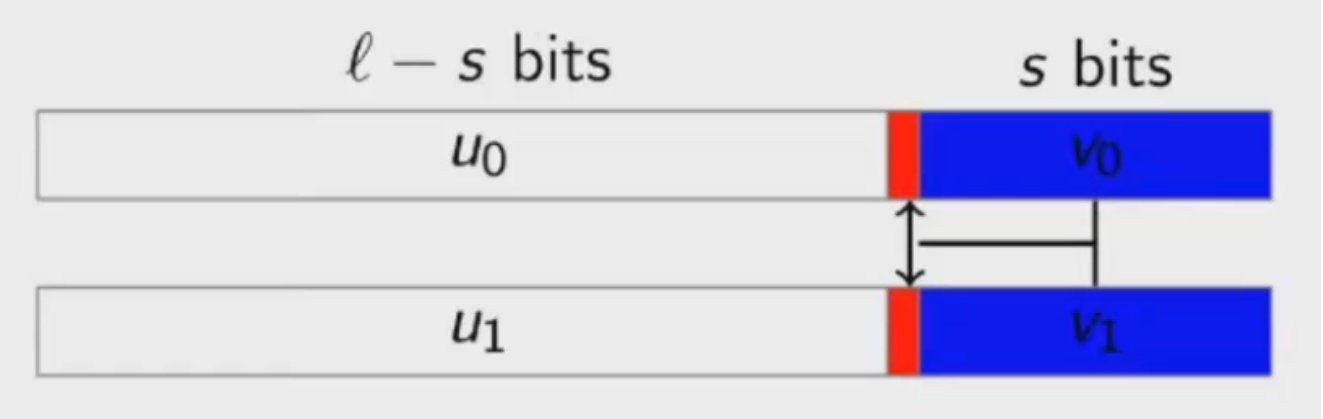
Here, x is the real number with sign, $x \in [-2^{m-1}, 2^{m-1} - 1]$, x' is the unsigned number

Part 3: Truncation

3.1 Logial Right Shift: $\rangle\rangle_L$

For $x \in Z_L$, $x = \langle x \rangle_0^l + \langle x \rangle_1^l \bmod L$, denote $\langle x \rangle_b^l = u_b \mid\mid v_b$, where $u_b \in \{0,1\}^{l-s}$ and $v_b \in \{0,1\}^s$, based on ***equation (1)***, we have ***equation (2)***

$$x\rangle\rangle_L s = u_0 + u_1 - 2^{l-s} \cdot \ wrap(\langle x\rangle_0^l,\ \langle x\rangle_1^l,\ L) + wrap(v_0,\ v_1,\ 2^s)$$



A trivial solution is compute $wrap(\langle x\rangle_0^l,\ \langle x\rangle_1^l,\ L)$ and $wrap(v_0,\ v_1,\ 2^s)$ directly.

Optimization: Use $\langle u\rangle^{l-s}$ and $\langle v\rangle^s$ compute $wrap(\langle x\rangle_0^l,\ \langle x\rangle_1^l,\ L)$ in order to save communication spense

Lemma:

Let $x \in Z_L$, where $u_b \in \{0,1\}^{l-s}$ and $v_b \in \{0,1\}^s$. Define $c = wrap(v_0,\ v_1,\ 2^s)$, $d = wrap(u_0,\ u_1,\ 2^{l-s})$, $e = 1\{(u_0 + u_1 \mod 2^{l-s}) = 2^{l-s} - 1\}$, and $w = wrap(\langle x\rangle_0^l,\ \langle x\rangle_1^l,\ L)$, then $w = d \oplus (c \wedge e)$

Proof:

Let $x_b = \langle x\rangle_b^l$, then in Z we have $x_b = u_b \cdot 2^S + v_b$, and

$$x_0 + x_1 = (v_0 + v_1) + (u_0 + u_1) \cdot 2^s$$

$$= (v_0 + v_1 - c \cdot 2^s) + (u_0 + u_1 - d \cdot 2^{l-s}) \cdot 2^s + c \cdot 2^s + d \cdot L$$

$$= v' + (u' + c) \cdot 2^s + d \cdot L$$

Let $w' = 1\{u' + c > 2^{l-s} - 1\}$, we have ***equation (3)***

$$x_0 + x_1 = v' + (u' + c - w' \cdot 2^{l-s}) \cdot 2^s + (d + w') \cdot L$$

- $d = 1 \Rightarrow u_0 + u_1 \geq 2^{l-s} (\Leftrightarrow u_0 + u_1 > 2^{l-s} - 1) \Rightarrow e = 0$ ($because\ u' = u_0 + u_1 + 1 \cdot 2^{l-s}$. $u_0, u_1 \leq 2^{l-s} - 1, u' \leq 2^{l-s} - 2$). Thus, $w' = 0$, no matter c= {1,0};
- $d = 0 \Rightarrow u_0 + u_1 \leq 2^{l-s} - 1 \Rightarrow w' = 1\ iff\ (u' = 2^{l-s} - 1) \wedge (c = 1) \Leftrightarrow (e = 1) \wedge (c = 1)$. $Otherwise$, $w' = 0$

Therefore, at most one of d and w' is 1. Thus

$$x_0 + x_1 = v' + (u' + c - w' \cdot 2^{l-s}) \cdot 2^s + (d \oplus (c \wedge e)) \cdot L$$

Since $v' < 2^s$ and $u' + c - w' \cdot 2^{l-s} < 2^{l-s}$, therefore $v' + (u' + c - w' \cdot 2^{l-s}) \cdot 2^s < 2^l$, we have

$$w = d \oplus (c \wedge e)$$

3.2 Arithmetic Right Shift: $\rangle\rangle_A$ & Truncate and Reduce $\langle TR(x, s)\rangle^{l-s}$

Recall that $int(x) = uint(x') - 2^{m-1}\ for\ x' = x + 2^{m-1} \mod M$.

$$\Rightarrow x\rangle\rangle_A = (x'\rangle\rangle_L s) - 2^{l-s-1}$$

From ***equation (2)***, we have $\langle TR(x,s)\rangle^{l-s} = u_0 + u_1 + wrap(v_0,\ v_1,\ 2^s)$, since $2^{l-s} *_l wrap(\langle x\rangle_0^l,\ \langle x\rangle_1^l,\ L)\ mod\ 2^{l-s} = 0$.

3.3 Division by power-of-2

In C++ convention, we have

$$F_{DivPow2}^{l,s}(x) = \begin{cases} \lceil \frac{int(x)}{2^s} \rceil\ mod\ L, & x < 0 \\ \lfloor \frac{int(x)}{2^s} \rfloor\ mod\ L, & x \geq 0 \end{cases}$$

Let $sign(x) = 1\{x \geq 2^{l-s}\}$, $c = 1\{x\ mod\ 2^s \equiv 0\}$, we have

$$DivPow2(x,s) = (x\rangle\rangle_A s) + sign(x) \wedge c$$

Part 4: Multiplication with non-uniform bitwidths

4.1 Unsigned Mutiplication

For $F_{UMult}^{m,n}$

- Input $\langle x\rangle^m$ and $\langle y\rangle^n$;
- Output $\langle z\rangle^l$ with $z = uint(x) *_l uint(y)$ and $l = m + n$.

$$uint(x) \cdot uint(y) = (x_0 + x_1 - 2^m w_x) \cdot (y_0 + y_1 - 2^n w_y)$$

$$= x_0 y_0 + x_1 y_1 + x_0 y_1 + x_1 y_0 - 2^m w_x y - 2^n w_y x + 2^l w_x w_y$$

For $\Pi_{UMult}^{m,n}$

- $x_b y_b$ can be computed locally;
- $2^l w_x w_y$ is modulo-reduced when $mod\ L$;
- $w_x y$ and $w_y x$ can be computed with F_{wrap} and F_{Mux}, using OT
- Focus on $x_b y_{1-b}$ ($F_{CrossTerm}^{m,n}$, use COT)

Main work is focused on $F_{CrossTerm}^{m,n}$, we choose party with less bit as receiver and the other as sender. Therefore, the receiver can offer less comparison than the other party if it is the receiver.

4.2 Signed Multiplication

For $F_{SMult}^{m,n}$

- Input $\langle x\rangle^m$ and $\langle y\rangle^n$;
- Output $\langle z\rangle^l$ with $z = int(x) *_l int(y)$ and $l = m + n$.

Firstly, compute $x' = x + 2^{m-1}\ mod\ M$ and $y' = y + 2^{n-1}\ mod\ N$ with $x' = x'_0 + x'_1\ mod\ M$ and $y' = y'_0 + y'_1\ mod\ N$, then

$$int(x) \cdot int(y) = (x' - 2^{m-1}) \cdot (y' - 2^{n-1}) = x' \cdot y' - 2^{m-1}(y'_0 + y'_1 - 2^n w_{y'}) - 2^{n-1}(x'_0 + x'_1 - 2^m w_{x'}) + 2^{m+n-}$$

- $x'y'$ can be computed using $F_{UMult}^{m,n}$;

- $2^{l-1}(w_{x'} + w_{y'})$: $2^{l-1}w_{x'} = 2^{l-1}(\langle w_{x'}\rangle_0^B + \langle w_{x'}\rangle_1^B - 2\langle w_{x'}\rangle_0^B\langle w_{x'}\rangle_1^B)$, the last term can be modulo-reduced when $mod\ L$.

4.3 Matrix Multiplication and Convolution

$C = AB$ with $A \in Z_M^{d_1 \times d_2}$, $B \in Z_N^{d_2 \times d_3}$, $C \in Z_L^{d_1 \times d_3}$, and $l = m + n$

- Addition introduces overflow;
  - $log_2 d$-bits extension.
- Immediate-values element-wise extension: $d_1 d_2 d_3 \times$extension.
- Optimization: In $F_{CrossTerm}$, 2 parties extend the larger-bit input by $log_2 d$-bits. This approach is faster than perform F_{SExt} to the data in the matrices in the first place

Multiply and Truncate: $l \leq m + n$.

By using img2col, convolution and mutiplication are the same.

Part5: Digit Decomposition & MSNZB

5.1 Digit Decomposition

For $F_{DigDec}^{l,\{d_i\}_{i\in[c]}}$

- Input $\langle x\rangle^l$;
- Output $\langle z_{c-1}\rangle^{d_{c-1}}, ..., \langle z_0\rangle^{d_0}$ with $x = z_{c-1} \mid\mid ... \mid\mid z_0$.

Considering $d \mid l$ and $c = l/d$. Denote $\langle x\rangle_b^l = y_{b,c-1} \mid\mid ... \mid\mid y_{b,0}$ and $Y_{b,i} = y_{b,i} \mid\mid ... \mid\mid y_{b,0}$.

$$z_i = y_{0,i} + y_{1,i} + carry_i \ mod\ 2^d$$

where $carry_i = Y_{0,i-1} + Y_{1,i-1} \geq 2^{id}$ ($\Leftrightarrow wrap(Y_{0,i-1},\ Y_{1,i-1},\ 2^{id})$). However we can optimize this by using recursion according to the previous lemma.

$c = wrap(Y_{0,i-2},\ Y_{1,i-2}, 2^{(i-1)d})$, $d = wrap(y_{0,i-1}, y_{1,i-1}, 2^d)$, $e = 1\{(y_{0,i-1} + y_{1,i-1\ mod\ 2^d}) = 2^d - 1\}$, and $carry_i = wrap(Y_{0,i-1}, Y_{1,i-1}, 2^{ib})$, we have $carry_i = d \oplus (c \wedge e)$.

5.2 Most Significant Non-Zero Bit (MSNZB)

For F_{MSNZB}^l

- Input $\langle x\rangle^l$;
- Output $k \in [l]$ with $x_k = 1$ and $x_j = 0$ for $j < k$.

$F^{l,d}_{DigDec}(x) = y_{i i\in[c]} \Rightarrow MSNZB(x) = MSNZB(y_i) + id$ with $y_i \not\equiv 0$ and $y_j = 0$ for $j > i$;

Wisdom of division and conquer

Solution:

- Let $\iota = log_2 l$, $F^{d,l,i}_{MSNZB-P}(\langle y_i\rangle^d) = \langle u_i\rangle^\iota$ with $2^{u_i - id} \le y_i < 2^{u_i - id+1}$;
- $F^d_{Zeros}(\langle y_i\rangle^d) : v_i = 1\{y_i = 0\}$;
- $z'_i = u_i \cdot (1 \oplus v_i) \cdot \Pi_{j>i} v_j$, such that $z'_i = u_i$, iff. $y_i \not\equiv 0$ and $y_j = 0$ for $j \not\equiv i$
- $MSNZB(x) = \sum_i (z'_i + id)$
- For $F^{d,l,i}_{MSNZB-P}(\langle y_i\rangle^d)$, we use OT and LUT

Part6: MSB to Wrap Optimization

- $w = wrap(\langle x\rangle^l_0, \langle x\rangle^l_1, L)$;
- $m_x = MSB(x)$ is known or $\langle m_x\rangle^B$ is known;
- $w = ((1 \oplus m_x) \wedge (m_0 \oplus m_1) \oplus (m_0 \wedge m_1))$;
- $\binom{4}{1} - OT_1$ or $\binom{2}{1} - OT_1$

Part7: Math Library

```
By using those building blocks mentioned ahead, we can implement math functions like exponential,
sigmoid, tan hyperbolic (tanh), and reciprocal square root. The key is to break down the problem into
small ones. And then use LUT to tackle them. Mathmatical techniques like Goldenschmidt's also help. Note
that these functions are impossible to implement exactly using finite-bit arithmetic, and hence, our
implementations realize them approximately.

For a functionality$\ f$ that takes as input the shares $\lang x\rang^m$ and outputs $\lang y\rang^n$
such that:
```

$$srt_{n,s'}(y) = f(srt_{m,s}(x))$$

7.1 Expotential

```
The protocol for this functionality can be built easily relying on the protocols described above.
```

- Step 1 can be implemented by a call to the digit decomposition functionality, F_{DicDec};
- The LUTs in Step 2 can be looked up using F_{LUT};
- These (s'+2)-bit values are multiplied using a tree-based multiplication using $F^{s'+2,s'+2,2s'+2,s'}_{SMultTR}$ to get an (s'+2)-bit number with scale s' in Step 3.
- Finally, Step 4 extends g to an n-bit value using $F^{s'+2,n}_{SExt}$.

Functionality $\mathcal{F}_{\text{rExp}}^{m,s,n,s'}(\langle x \rangle^m)$

1) Let $x = x_{k-1}||\ldots||x_0$, $x_i \in \{0,1\}^d$, $i \in [k]$, $dk = m$.
2) For $i \in [k]$, let $L_i : \{0,1\}^d \to \mathbb{Z}_{2^{s'+2}}$ s.t. $L_i(j) = \text{Fix}\left(\text{rExp}(2^{di-s}j), s'+2, s'\right)$.
3) Compute $g = L_{k-1}[x_{k-1}] * \ldots * L_0[x_0]$, g has bitwidth $s'+2$ and scale s'.
4) Return $\langle y \rangle^n$ for $y = \text{SExt}(g, s'+2, n)$.

7.2 Sigmoid and Tanh

Consider the math functionality$\ F_{sigmoid}^{m,s,n,s'}$ where $sigmoid(z) = \frac{1}{1+e^{-z}}$ can be written as

$$sigmoid(z) = \begin{cases} 0.5, & z = 0 \\ \frac{1}{1+rExp(z)}, & z > 0 \\ rExp(z) \cdot \frac{1}{1+rExp(z)}, & z < 0 \end{cases}$$

Hence, sigmoid can be built by extending the math functionality $F_h^{m,s,n,s'}$ such that $h(z) = \frac{1}{1+rExp(z)}$ described in Figure 4. This functionality calls F_{rExp} that we described above, followed by a call to a functionality to approximate the reciprocal that we describe next.

Functionality $\mathcal{F}_h^{m,s,n,s'}(\langle x \rangle^m)$

1) $\langle u \rangle^{s'+2} \leftarrow \mathcal{F}_{\text{rExp}}^{m,s,s'+2,s'}(\langle x \rangle^m)$.
2) $\langle w \rangle^{s'+2} \leftarrow \mathcal{F}_{\text{Rec}}^{s'+2,s'}(\langle 2^{s'} + u \rangle^{s'+2})$.
3) Return $\text{SExt}(w, s'+2, n)$.

Functionality $\mathcal{F}_{\text{Rec}}^{\ell,s}(\langle v \rangle^\ell)$

Computes the initial approximation w as follows [63]:

1) $v = d||e||f$, $d \in \{0,1\}^{\ell-s}$, $e \in \{0,1\}^g$, $f \in \{0,1\}^{s-g}$.
2) $c_0||c_1 = L_{\text{rec}}(e)$, $c_0 \in \{0,1\}^{g+4}$ and $c_1 \in \{0,1\}^{2g+3}$.
3) $c_2 = \text{SExt}((c_0 *_{s+4} f), s+4, s+g+4)$.
4) $w' = 2^{s-g+1} *_{s+g+4} c_1 - c_2$, $w = \text{TR}(w', g+3)$.

Goldschmidt's method for t iterations.

1) $p_1 = 2^s - \text{TR}(v *_{2s+2} w, s)$.
2) $q_1 = 2^s + p_1$, $a_1 = q_1$.
3) For $i \in \{2, \ldots, t\}$ do
   a) $a_i = \text{TR}(a_{i-1} *_{2s+2} q_{i-1}, s)$.
   b) $p_i = \text{TR}(p_{i-1} *_{2s+2} p_{i-1}, s)$.
   c) $q_i = 2^s + p_i$.
4) Return $\text{SExt}(a_t, s+2, \ell)$.

Tanh. The math functionality $F_{tanh}^{m,s,n,s'}$ where $tanh(z) = \frac{e^z - e^{-z}}{e^z + e^{-z}} = 2 \cdot sigmoid(2z) - 1$ can be realized using $F_{sigmoid}$.

7.3 Reciprocal of Square Root

```
In order to compute$\ rsqrt(x) = \frac{1}{\sqrt{x}}$, we add a small value to the denominator $\frac{1}
{\sqrt{x+\epsilon}}$
```

$$L_{rsqrt}(e \mid\mid B) = Fix(\frac{1}{(B+1)(1+urt_{(g,g)}(e))}, g+4, g+2)$$

Functionality $\mathcal{F}_{\mathsf{rsqrt}}^{\ell,s,\ell,s'}(\langle x \rangle^{\ell})$

Normalizes x to x' as follows:

1) $k = \mathsf{MSNZB}(x) \in [\ell]$.
2) $A = 2^{\ell-2-k}$, $B = (s-k) \bmod 2$.
3) $C = 2^{\lceil \frac{s-k}{2} \rceil + \lfloor \frac{\ell-s-1}{2} \rfloor}$.
4) $x' = x *_{\ell} A$.

Computes the initial approximation w as follows:

1) $x' = d||e||f, d \in \{0,1\}^2, e \in \{0,1\}^g, f \in \{0,1\}^{\ell-2-g}$.
2) $w = L_{\mathsf{rsqrt}}(e||B)$, $w \in \{0,1\}^{g+4}$.

Goldschmidt's method for t iterations:

1) $x'' = \mathsf{TR}(x', \ell - 3 - s')$, $q_0 = B \ ? \ x'' : \mathsf{TR}(x', 1)$.
2) $a_0 = 2^{s'-g-2} *_{s'+2} w$, $p_0 = a_0$.
3) For $i \in \{1, \ldots, t\}$ do
   a) $Y_i = \mathsf{TR}(p_{i-1} *_{2s'+2} p_{i-1}, s')$.
   b) $q_i = \mathsf{TR}(q_{i-1} *_{2s'+2} Y_i, s')$.
   c) $p_i = 3 \cdot 2^{s'-1} - (q_i \gg_A 1)$.
   d) $a_i = \mathsf{TR}(a_{i-1} *_{2s'+2} p_i, s')$.

Uses reciprocal square root of x' to compute the same for x:

1) Return $\mathsf{TR}(a_t *_{\ell/2+s'+3} C, \lfloor \frac{\ell-s-1}{2} \rfloor) \bmod L$.

7.4 Evaluation

```
By using exhaustive testing, they prove that our math implementations are precise for chosen parameters
and provide standard precision guarantees that are expected from math libraries.
```