SecureDNN: Efficient Analytics over Encrypted Data

$[r]_L^{S_1,3}$ n-out-of-n secret share of Party3 of the secret r in $\mathbb{Z}_L$ where $n = |S_1|$

$S_1 = \{P_1, P_2, P_3, P_4\}, S_2 = \{P_1, P_2\}, S_3 = \{P_3, P_4\}$.

TODO: Change all the code notation to the above format.

We start with a 4-out-of-4 arithmetic sharing of a private number z in $\mathbb{Z}_L$, where $L = 2^{64}$. $[z]_L$ denotes a 2-out-of-2 arithmetic shares of z over $\mathbb{Z}_L$ and $[\![z]\!]_L$ denotes a 4-out-of-4 arithmetic shares of z over $\mathbb{Z}_L$. $\langle z_i \rangle_L$ denotes shares of a bit z_i in the field $\mathbb{Z}_L$ and $\langle\!\langle z \rangle\!\rangle_L$ denotes the set of shares $\{ \langle z_i \rangle_L \}_{i=1}^{l}$ where z_i are the bits of z. Finally, $(x >_B 0)$ denotes the bit representing the comparison *i.e.,* $(x >_B 0) = 1$ if $x > 0$ and 0 otherwise.

Algorithm 3 is an optimized version of Nishide T. *et al.* [6] and Algorithm 8 is an optimized version of DGK [2]. For correctness of Algorithm 1, we note that Algorithm 1 computes shares of $z \cdot (\widetilde{z} >_B 0)$ and it is easy to verify that this is exactly the ReLU function on z if $\widetilde{z} = z$ or $z + 1$. Correctness of Algorithm 3 follows from the fact that $\text{MSB}(a) = 1 - \text{LSB}(2a)$ when N is odd.

Details of Algorithm 1:

- Step 1 is conversion of 4-out-of-4 shares to two random 2-out-of-2 shares.
- Step 2 is converting the secret shared input from ring $\mathbb{Z}_L$ where $L = 2^l$ to a ring with odd modulus $(\mathbb{Z}_{L-1})$ (Step 2 can replaced by a more precise ShareConvert algorithm describes as Algorithm 12). It is easy to see that $\widetilde{z} = z$ or $z + 1$.
- Steps 3-4 compute shares of $(\widetilde{z} >_B 0)$ with P_1, P_2. (Note that Steps 3-12 can be optimized but are separated for clarity)
- Steps 5-13 send a XOR'ed version of $(\widetilde{z} >_B 0)$ to P_3, P_4 and the latter reconstruct this bit.
- Steps 14-27 send shares of 0 and z appropriately to P_1 and P_2 so that using their initial shares, they can compute new shares of $\text{ReLU}(z)$.

Details of Algorithm 3:

- Steps 1-2 is pre-processing to transfer the problem challenge to that over a random number with boolean shares.
- Steps 3-14 computes the LSB of $(2a)$.
- Finally, Step 15 connects the LSB of $2a$ in $\mathbb{Z}_N$ to the $\text{MSB}(a)$.

Details of Algorithm 8:

- Steps 1-6 is pre-processing for security against P_3, P_4 learning the comparison bit.
- Steps 7-14 performs local computation for the compare.
- Finally, Steps 15-21 complete the computation for the bit β'.

Algorithm 1 CompleteProtocol $[\![z]\!]_L$:

Input: P_1, P_2, P_3, P_4 hold $[\![z]\!]_L$.
Output: P_1, P_2 hold $[\text{ReLU(z)}]_L$ without anyone learning the bit $(z >_B 0)$.

1: $[\![z]\!]_L \xrightarrow{R} [z]_L$ between P_1, P_2
2: $[\![z]\!]_L \xrightarrow{R} [-z]_L$ between P_3, P_4.
3: $[0]_L$ between P_3, P_4 (random shares of 0 in $\mathbb{Z}_L$).

4: Will have to use the ShareConvert protocol
5: shares of z to shares of $2z$. Then we convert shares of $2z$ into the field $L - 1$. Then we compute the MSB of $2z$. We know that the MSB of $2z$ is the same as MSB of z and then use this to select the shares of the original z or 0.
6: P_1, P_2 just assume their share $[z]_L$ is a share of some $\widetilde{z}$ in $\mathbb{Z}_{L-1}$ *i.e.,* $[\widetilde{z}]_{L-1}$.

7: $[\alpha]_2 \leftarrow \text{EfficientMSB}([\widetilde{z}]_{L-1})$.
8: This α is ReLU'$(\widetilde{z})$, which the same as ReLU of z except at 0 it is 1 instead of 0.

9: $\text{ComputeReLU}([\alpha]_2)$.
10: **return** P_1 and P_2 have shares of $[\max(z, 0)]_L$.

For EfficientMSB algorithm, the correctness follows from the following observation:

$$x_0 = \begin{cases} c_0 \oplus r_0 & \text{If no wrap occurs} \\ 1 - \{c_0 \oplus r_0\} & \text{If wrap occurs} \end{cases}$$

Also,

$$(c <_B r) = \begin{cases} 0 & \text{If no wrap occurs} \\ 1 & \text{If wrap occurs} \end{cases}$$

Since c is publicly known, $\mathbb{Z}_2$ shares of $c_0 \oplus r_0$ can be computed from $\mathbb{Z}_2$ shares of bits of r *i.e.,* from $\langle\!\langle r \rangle\!\rangle_2$. Hence we can find $\mathbb{Z}_2$ shares of x_0 using $\langle\!\langle r \rangle\!\rangle_2$ and $\langle (c <_B r) \rangle_2$. More precisely, let $(c <_B r) = \gamma$ and $c_0 \oplus r_0 = \delta$. Then,

Algorithm 2 ComputeReLU (α):

Input: P_1, P_2 hold shares $[\alpha]_2$ where α is the selector bit. P_1, P_2 also hold shares $[z]_2$ and P_3, P_4 hold shares $[-z]_2$.
Output: P_1, P_2 hold new shares of $[z]_2$ or $[0]_2$ depending on whether $\alpha = 0$ or 1.

P3, P4 need to randomize their shares of $[-z]$ since right now they are just shares of z with P1, P2 with a random number added. Hence step where P3, P4 send share tuples . . . leaks information.
P_1, P_2 agree on a bit β.
if $\beta = 1$ **then**
    P_1 sends share $[\alpha]_2$ to P_3, P_4
    P_2 sends share $[\alpha]_2$ to P_3, P_4
else
    P_1 sends share 1 - $[\alpha]_2$ to P_3, P_4
    P_2 sends share - $[\alpha]_2$ to P_3, P_4
end if
P_3, P_4 reconstruct bit β' from the received shares.

if $\beta' = 0$ **then**
    P_3, P_4 set $[u]_L = [-z]_L$ and $[v]_L = [0]_L$
else
    P_3, P_4 set $[u]_L = [0]_L$ and $[v]_L = [-z]_L$
end if
P_3 sends share tuple $([u]_L, [v]_L)$ to P_1
P_4 sends share tuple $([u]_L, [v]_L)$ to P_2
if $\beta = 0$ **then**
    P_1, P_2 compute $[z]_L + [u]_L$.
else
    P_1, P_2 compute $[z]_L + [v]_L$.
end if

P_1, P_2 hold new shares of $[z]_2$ or $[0]_2$ depending on whether $\alpha = 0$ or 1.

$$\begin{aligned} x_0 &= \gamma * \delta + (1-\gamma) * (1-\delta) \\ &= 2\gamma\delta - \gamma - \delta + 1 \\ &\equiv 1 - \gamma - \delta \pmod 2 \end{aligned}$$

Finally, the MSB of a can be found as $1 - (2a)_0 \equiv \gamma + \delta \pmod 2$.

Algorithm 3 EfficientMSB ($[a]_N$):

Input: N is odd. P_1, P_2 have $[a]_N$.
Output: P_1, P_2 hold shares $[\alpha]_2$ where $\alpha = \text{MSB}(a)$

P_3 generates $r \xleftarrow{R} \mathbb{Z}_N$ and $[r]_N$, $\langle\langle r \rangle\rangle_p$ and $\langle\langle r_0 \rangle\rangle_2$.
P_1, $P_2 \leftarrow [r]_N$, $\langle\langle r \rangle\rangle_p$ and $\langle\langle r_0 \rangle\rangle_2$.

P_1, P_2 reconstruct $c \equiv 2a + r \pmod N$
P_1, P_2 agree on a bit β.
$\beta' \leftarrow \text{PrivateCompare}(\langle\langle r \rangle\rangle_N, c, \beta)$
P_4 has the bit β'.
P_4 generates θ_1, θ_2 as shares $[\beta']_2$.
P_4 sends θ_i to P_i for $i = \{1, 2\}$.

P_1 sets $[\gamma]_2 = \beta \oplus \theta_1$
P_2 sets $[\gamma]_2 = \theta_2$

if $c_0 = 0$ **then**
    P_1, P_2 set $[\delta]_2 = [r_0]_2$
else
    P_1 sets $[\delta]_2 = 1 \oplus [r_0]_2$
    P_2 sets $[\delta]_2 = [r_0]_2$
end if
P_1, P_2 compute $[\alpha]_2 = [\gamma]_2 \oplus [\delta]_2$

return P_1, P_2 have $\alpha = \text{MSB}(a)$.

Algorithm 6 Division protocol:

Input: P_1 and P_2 have shares of x, y $(x \le y)$ x/y
Output: P_1 and P_2 have shares of x/y

ReLU'(something) = 1 if something > 0 and 0 otherwise
selectShares(b, s_0, s_1) will select shares of s_b.
$x/y = 0.b_1b_2...b_f$ where f = FLOAT_PRECISION.

$P = 0$, $X = 0$.
Q, D.

for $i = \{1, 2, \ldots, f\}$ **do**
    ReLU'$(x - P - y/2^i) = B$
    selectShares$(B, 0, y/2^i) = D$
    $P += D$
    selectShares$(B, 0, 1/2^i) = Q$
    $X += Q$
end for
return P_1 and P_2 have shares of the quotient in X.

Algorithm 4 PrivateCompare ($\langle m \rangle_N$, x, β): where x is public

Change this for correctness. From $\mathbb{Z}_N$ to $\mathbb{Z}_p$.

Input: P_1, P_2 have a bit β and $\mathbb{Z}_N$ shares of private message m. $N \geq l + 3$

Output: P_3 receives a bit β' where $\beta' = \beta \oplus \gamma$ where $\gamma = (m >_B x)$

P_1, P_2 agree on a permutation Π
if $\beta = 0$ **then**
    P_1, P_2 compute $(m > x)$
else
    P_1, P_2 compute $(m < x)$ (swap x, m)
end if

for $i = \{1, 2, \ldots, l\}$ **do**
    $[w_i]_N^2 = [m_i + x_i - 2m_i x_i]_N = [m_i \oplus x_i]_N^2$
    $[c_i]_N^2 = [x_i - m_i + 1 + \sum_{j=i+1}^{l} w_j]_N^2$
    P_1, P_2 agree on $s_i \xleftarrow{R} \mathbb{Z}_N^*$.
    P_1, P_2 shuffle $c_i s_i$ according to Π.
    P_1, P_2 send permuted shares to P_3.
end for

P_3 reconstructs $c_i s_i$ for $i = \{1, 2, \ldots, l\}$.
if $\exists i$ such that $c_i s_i = 0$ **then**
    Set bit $\beta' =$1
else
    Set bit $\beta' =$0
end if
return P_3 has the desired bit β'.

Algorithm 5 Divide shares by power of 2

Input: Shares a_0, a_1 of a. Everything is in the ring uint64_t

Output: Shares of $a/2^\alpha$.

Truncate(a_i, α) := Arithmetic shift of a_i by α.
if $i = 0$ (i.e., depends on party) **then**
    Truncate(a_i, α)
else
    -Truncate($-a_i, \alpha$)
end if
return Each party has required shares.

We have four parties holding shares of a secret value $a \in \mathbb{F}_p$ for some prime p (shares in this Field by $[a]_p$) and wish to compute shares of $\max(a, 0)$ (in the same Field). Specifically, coming back from step 3 of Table 1, we need to compute

$$\max(\sum_{k=0}^{3} z_k, 0)$$

given shares of $\left(\sum_{k=0}^{3} z_k\right)$.

Solution:

1) Figure out the best way to go to shares of Boolean values of z.
2) Convert shares $\{z_k\}_{k=0}^3$ into two sets of random shares of $z = \sum_{k=0}^{3} z_k$ (this is required for an efficient implementation of step 3, is not strictly necessary).
3) Compute compare$(z > 0)$ and depending on it, split new shares (in $\mathbb{F}_p$) of either z or 0.

One protocol to complete step 2 and 3 together is to follow DGK [2] to get the shares of the variables c_i and then instead of the heavy cryptographic operations, leverage the power of the 4 parties in the HbC model. Such a protocol is described in Algorithm 7. The correctness follows directly from DGK [2] where the encryption has been replaced by the blinding using s_i in the field. The security follows from DGK, the HbC model for adversary and some minor bookkeeping.

Algorithm 7 NewShares $([\mathtt{z}]_{\mathtt{p}}^4, [\mathtt{z}]_{\mathtt{B}}^2, [\mathtt{x}]_{\mathtt{B}}^2)$: where $\mathtt{x} = \mathtt{0}$

$[z]_p^4 \xrightarrow{R} [z]_p^2, [z]_p^2$

PartyA, PartyB agree on a bit β.
PrivateCompare$([z]_B^2, [0]_B^2, \beta)$
if $\beta' = 0$ **then**
    PartyD sends his share tuple $([-z]_p^2, [0]_p^2)$ to PartyB
    PartyC sends his share tuple $([-z]_p^2, [0]_p^2)$ to PartyA
else
    PartyD sends his share tuple $([0]_p^2, [-z]_p^2)$ to PartyB
    PartyC sends his share tuple $([0]_p^2, [-z]_p^2)$ to PartyA
end if
if $\beta = 0$ **then**
    PartyA, PartyB add their shares of $[z]_p^2$ to the first inputs received from PartyC and PartyD respectively.
else
    PartyA, PartyB add their shares of $[z]_p^2$ to the second inputs received from PartyC and PartyD respectively.
end if
return PartyA and PartyB together have $[\max(z, 0)]_p^2$.

One possible way to complete step 1 is as below:

$$\begin{aligned}
&\mathbb{F}_p \text{ shares of } z \\
&\xrightarrow{\text{ABY}} \mathbb{F}_2 \text{ shares of Bool } z_i \\
&\xrightarrow{l\text{-MULT}} \mathbb{F}_p \text{ shares of Bool } z_i \\
&\xrightarrow{\text{NewShares}} \mathbb{F}_p \text{ shares of ReLU(z)}
\end{aligned}$$

Computational complexity of this approach is:

$$\begin{aligned}
\text{ABY: A2B cost} &: 12l \\
l\text{-MULT} &: O(l) \\
\text{NewShares} &: O(1)
\end{aligned}$$

Communication complexity of this approach is:

$$\begin{aligned}
\text{ABY-A2B cost} &: 6lk \\
2l - \mathbb{Z}_p \text{ shares} &: 2l^2\text{-bits} \\
\text{NewShares} &: (4l + 4)l + 1\text{-bits} + 4\text{-2 cost}
\end{aligned}$$

Algorithm 8 PrivateCompare $([\mathtt{m}]^2_{\mathtt{B}}, [\mathtt{x}]^2_{\mathtt{B}}, \beta)$: where $\mathtt{x}$ is public

Input: PartyA, PartyB have a bit β and $\mathbb{F}_p$ shares of private message m.

Output: PartyC, PartyD receive a bit β' where $\beta' = \beta \oplus \gamma$ where $\gamma = (m >_B x)$

if $\beta = 0$ **then**
    PartyA, PartyB agree to compute: $m > x$
else
    PartyA, PartyB agree to compute: $m < x$ (swap x, m and proceed)
end if
for $i = 1 : l$ **do**
    $[w_i]^2_p = [m_i + x_i - 2m_i x_i]_p = [m_i \oplus x_i]^2_p$
    $[c_i]^2_p = [x_i - m_i + 1 + \sum_{j=i+1}^{l} w_j]^2_p$
    PartyC generates $s_i \xleftarrow{R} \mathbb{F}_p$.
    PartyC sends $s_i \rightarrow$ PartyA, PartyB.
    PartyA, PartyB send their shares of $c_i s_i$ to PartyD
end for
PartyD reconstructs shares $c_i s_i$ for $i = 1 : l$.
if $\exists i$ such that $c_i s_i = 0$ **then**
    Set bit $\beta' = 0$
else
    Set bit $\beta' = 1$
end if
Send β' to PartyC.
return PartyC and PartyD both have the desired bit β'.

One possible way of implementing the PrivateCompare function is shown in Algorithm 8. All these required the following flowchart:

$$\mathbb{F}_p \text{ shares of } z \xrightarrow{\text{ABY}} \mathbb{F}_2 \text{ shares of Bool } z_i \xrightarrow{\text{MULT}} \mathbb{F}_p \text{ shares of Bool } z_i \xrightarrow{\text{NewShares}} \mathbb{F}_p \text{ shares of ReLU(z)}$$

Next we show how to avoid the overhead of the ABY conversion i.e., directly from $\mathbb{F}_p$ shares of z to $\mathbb{F}_p$ shares ReLU$(z) = \max(z, 0)$. This is done using CompareDirect presented as Algorithm 9.

We consider the specialized protocol for finding the interval test as given in Sec. 6.2 of Nishide T.*et al.* [6]. The protocol takes as inputs shares of a private integer a in $\mathbb{Z}_p$ (works even if p is not a prime) and outputs shares of $[a \leq_B (p-1)/2]_p$. We demonstrate a more efficient version of this protocol using a 4 party HbC adversary model (we also show that this protocol works for a generic modulus N as long as N is odd). This protocol is called EfficientMSB.

Algorithm 9 CompareDirect $([\mathtt{m}]^2_{\mathtt{p}})$:

Input: PartyA, PartyB have $\mathbb{F}_p$ shares of private message m.

Output: PartyA, PartyB need $\mathbb{F}_p$ shares of ReLU$(z) = \max(z, 0)$ without learning the value of $(z >_B 0)$.

PartyA, PartyB run FindLSB$(2[m]^2_p)$.
PartA, PartB find shares of MSB(m) as $[1 -$ FindLSB$(2[m]^2_p)]$.
What is happening here?
PartyC and PartyD both have the desired bit β'.

Algorithm 10 FindLSB $([\mathtt{m}]^2_{\mathtt{p}})$:

Input: PartyA, PartyB have $\mathbb{F}_p$ shares of private message m.

Output: PartyA, PartyB need $[m_0]^2_p$ i.e., shares of the LSB of m.

PartyD generates $r \xleftarrow{R} \mathbb{F}_p$ and sends $[r]^2_p$ and $[r]^2_B$ to PartyA, PartyB.
PartyA, PartyB reconstruct $c \equiv m + r \pmod p$
PartyA, PartyB agree on a bit β.
$\beta' \leftarrow$ PrivateCompare$([r]^2_B, [c]^2_B, \beta)$ ▷ Note that c is public
PartyA, PartyB compute $[c <_B r]^2_p = [\beta \oplus \beta']^2_p$ securely.
if $c_0 = 0$ **then**
    PartyA, PartyB set $[\gamma]^2_p = [r_0]^2_p$
else
    PartyA sets $[\gamma]^2_p = 1 - [r_0]^2_p$
    PartyB sets $[\gamma]^2_p = -[r_0]^2_p$
end if
$[m_0]^2_p = [c <_B r]^2_p + [\gamma]^2_p - 2[c <_B r]^2_p [\gamma]^2_p$
return PartyA, PartyB have $[m_0]^2_p$.

For efficiency, we perform all the protocols in $\mathbb{Z}_{2^l}$ for some l. But Algorithm 3 works only for odd modulus N. Hence we need a technique to convert shares of a private number a from $\mathbb{Z}_{2^l}$ to $\mathbb{Z}_N$ for some odd N. Here we present two ideas:

- Assume that the shares $[a]_{2^l}$ are shares of $\widetilde{a}$ in $\mathbb{Z}_N$ where $N = 2^l - 1$. It is easy to see that these shares are shares of either a or $a+1$ in $\mathbb{Z}_N$. Finally, we complete the protocol by showing how to get shares of ReLU(a) starting from shares of $[a]_{2^l}$ and using $[\widetilde{a}]_N$.
- ~~Use an efficient algorithm to convert shares of the private integer a in $\mathbb{Z}_{2^l}$ to shares of the same integer in $\mathbb{Z}_N$ where $N = 2^l - 1$. Such an algorithm is presented in Algorithm 12. This protocol might be of independent interest to the community.~~

A 4 party, HbC model protocol to convert $\mathbb{Z}_L$ shares of a secret a to $\mathbb{Z}_{L-1}$ shares of a. The algorithm is called ShareConvert (ShareConvert). This might be of independent

interest.

The key idea of this protocol is to find whether there is a wrap around when the shares of a are added together i.e., if $a_1 + a_2 > L$. And depending on that, they generate new shares of a.

Algorithm 11 ShareConvert $([a]_L^2)$:

Input: PartyA, PartyB have $\mathbb{Z}_L$ shares of private message a.

Output: PartyA, PartyB need $[a]_{L-1}^2$ i.e., shares of the a in $\mathbb{Z}_{L-1}$

1: change this to $\mathbb{Z}_p$ for security.
2: PartyD generates $r_1 + r_2 \equiv r \pmod{L}$ such that $r \xleftarrow{R} \mathbb{Z}_L$ as well as shares of bits of r i.e., $[r]_B^2$. It also computes a bit α which indicates if there was a wrap around in $r_1 + r_2 \equiv r \pmod{L}$ ($\alpha = 1$ if there is a wrap around).
3: PartyD sends $[r_1]_L^2$ and $[r_2]_L^2$ to PartyA, PartyB.
4: PartyA computes $\widetilde{a_1} \equiv a_1 + r_1 \pmod{L}$. Set $\beta_1 = 1$ if there was a wrap around and 0 otherwise.
5: PartyB computes $\widetilde{a_2} \equiv a_2 + r_2 \pmod{L}$. Set $\beta_2 = 1$ if there was a wrap around and 0 otherwise.
6: PartyA, PartyB send $\widetilde{a_1}$ and $\widetilde{a_2}$ to PartyC.
7: PartyC computes $x \equiv \widetilde{a_1} + \widetilde{a_2} \pmod{L}$ and δ denotes a bit if there was a wrap around.
8: PartyC knows x and provides PartyA and PartyB with Boolean shares of x. ▷ $x \equiv a + r \pmod{L}$
9: PartyC sends x to PartyA and PartyB.
10: PartyA and PartyB using the Boolean shares of r and $a + r$ compute shares of $\eta = (r <_B x)$ using an 4 party optimized version of DGK. (Algorithm 8:)
11: Let θ denote a bit if there is a wrap around in $a \equiv a_1 + a_2 \pmod{L}$.
12: $\theta = \eta + \beta_1 + \beta_2 - \alpha - \delta$.
13: PartyC sends boolean shares of δ to PartyA, PartyB.
14: PartyA, PartyB generate boolean shares of 0 called γ_1, γ_2.
15: PartyA, PartyB locally compute the boolean bit $\eta_i + \beta_i - \delta_i + \gamma_i$ for $i = 1, 2$.
16: PartyA, PartyB send this to PartyD.
17: PartyD reconstructs this bit and subtracts α.
18: PartyD generates shares of this (essentially θ) in $\mathbb{Z}_{L-1}$.
19: **return** Now PartyA and PartyB can add this share of θ to their shares of a as everything is in $\mathbb{Z}_{L-1}$.

Correctness can be verified easily by considering the following: We know that $x = a + r - \eta \cdot L$ where $x \equiv \widetilde{a_1} + \widetilde{a_1} \pmod{L}$. Also, we have the following equations:

$$\begin{aligned} a &= a_1 + a_2 - \theta \cdot L \\ r &= r_1 + r_2 - \alpha \cdot L \\ \widetilde{a_1} &= a_1 + r_1 - \beta_1 \cdot L \\ \widetilde{a_2} &= a_2 + r_2 - \beta_2 \cdot L \\ a + r - \eta \cdot L = x &= \widetilde{a_1} + \widetilde{a_2} - \delta \cdot L \end{aligned}$$

Combining these, we get that, $\theta + \alpha + \delta = \beta_1 + \beta_2 + \eta$.

a_1 and a_2 assume their shares are essentially (just reinterpret them) x_1, x_2 in $\mathbb{Z}_{L-1}$. Then,

$$x_1 + x_2 = \begin{cases} a, & \text{if } \theta = 0 \\ a + 1, & \text{if } \theta = 1 \end{cases}$$

Hence, $x_1 - \theta_1$ and $x_2 - \theta_2$ when interpret as numbers in $\mathbb{Z}_{L-1}$ are shares of a in $\mathbb{Z}_{L-1}$.

Big picture, for the above shareConvert to work, we need that $a \neq L - 1$. Since we know that a is small positive or negative, we run everything on $2a$ get new shares of $2a$ and then finally, the efficient MSB protocol will give us a bit which is the ReLU'. From here on, to compute the ReLU, we use the selection protocol where we go back to using the shares of a.

The next page contains the new shareConvert protocol

Algorithm 12 ShareConvert ($[a]_L^2$):

Input: PartyA, PartyB have $\mathbb{Z}_L$ shares of private message a, where $a \neq L-1$.

Output: PartyA, PartyB need $[a]_{L-1}^2$ i.e., shares of the a in $\mathbb{Z}_{L-1}$

1: A,B generate random number r and its shares $[r_1]_L^2$ and $[r_2]_L^2$ and compute α as the overflow bit.
2: PartyA computes $\widetilde{a_1} \equiv a_1 + r_1 \pmod{L}$. Set $\beta_1 = 1$ if there was a wrap around and 0 otherwise.
3: PartyB computes $\widetilde{a_2} \equiv a_2 + r_2 \pmod{L}$. Set $\beta_2 = 1$ if there was a wrap around and 0 otherwise.
4: PartyA, PartyB send $\widetilde{a_1}$ and $\widetilde{a_2}$ to PartyC.
5: PartyC computes $x \equiv \widetilde{a_1} + \widetilde{a_2} \pmod{L}$ and δ denotes a bit if there was a wrap around.
6: PartyC knows x and provides PartyA and PartyB with Boolean shares of x in $\mathbb{Z}_p$.
7: PartyA and PartyB run 4 party optimized version of DGK. (Algorithm 8:) on the Boolean shares of r and x and give PartyD a bit $\eta' = (r <_B x) \oplus \eta''$ where A,B generated the bit η''
8: PartyA, PartyB generate $\eta = \eta' \oplus \eta''$.
9: Since $x < r \Rightarrow \eta = 1$, we set $\eta = 1 - \eta$ and then proceed.
10: Let θ denote a bit if there is a wrap around in $a \equiv a_1 + a_2 \pmod{L}$.
11: $\theta = \eta + \beta_1 + \beta_2 - \alpha - \delta$.
12: PartyC sends $\mathbb{Z}_{L-1}$ shares of δ to PartyA, PartyB.
13: PartyA, PartyB locally compute $\eta_i + \beta_i + \gamma_i$ as everything is in $\mathbb{Z}_{L-1}$ for $i = 1, 2$ and one of them subtracts α.
14: What they hold now is shares of θ.
15: **return** Now PartyA and PartyB can add this share of θ to their shares of a as everything is in $\mathbb{Z}_{L-1}$. This gives them $[a_i]_{L-1}$ as long as the $2a$ preprocessing is done.

1. Notation

To be consistent with notation, I follow a convention from [5]. Let $l \in \{1, 2, \cdots L\}$ denotes a general layer and L is the total number of layers (layer 1 is the input and leftmost). w^l_{jk} is the weight for the connection from k^{th} neuron in layer $l-1$ to the j^{th} neuron in layer l.
The output of the j^{th} neuron in layer l is a^l_j. For secret shares of a secret s we use $\langle s \rangle_i$ where $i \in \{0, 1\}$. We use δ^l_j to denote the error at the j^{th} neuron in layer l for the back prop. B denotes a mini-batch, $|B|$ denotes the size of the mini-batch. I have deferred the discussion about floating point to later in the project. As of now, the cost function has not come into picture.

2. Our protocol

Protocol is split into forward pass and back prop. Details are briefly summarized below.

2.1. Forward Pass

The equation below describes how the outputs propagate across the layers. The second equation is merely a vectorized representation of the first one. The notation for the matrices and vectors can be guessed from the first equation.

$$a^l_j = \sigma\left(\sum_k w^l_{jk} a^l_k\right) \qquad \text{i.e.,} \qquad a^l = \sigma\left(w^l a^l\right) \qquad (1)$$

where σ is the non-linear function and is applied component wise in the vector notation.
For the ReLU compute, we can (1) First get the shares back into a two party compute mode and then perform ReLU on these shares (2) Keep them in the four party mode (i.e., split across 4 parties) and then compute the function. Not sure which one is better yet. I'm going to assume the former is better and complete Table 1 in Table 2.
So we have two parties with shares $\langle \gamma^l \rangle_0, \langle \gamma^l \rangle_1$ and they wish to compute shares of $\max(0, \langle \gamma^l \rangle_0 + \langle \gamma^l \rangle_1)$. Basic idea is to convert the arithmetic shares into Yao (or Boolean) shares and perform the computation on the Yao (or Boolean) shares. I am not sure which one will be more efficient, Yao or Boolean; from the ABY paper, in theory the conversion cost is the same for both i.e., the complexity of (A2B, A2Y) is the same and so is the complexity of the pair (Y2A, B2A). At a deeper level, this could be a consequence of the free Y2B conversion. Hence, all we need to know is which one is more efficient, Yao or Boolean and use that in our protocol. This process is then repeated for each level l. At the end of the forward pass, parties secret share a^L. (Possibly need functionality to evaluate cost function.)
So the detail is in finding the most efficient way to compute the ReLU. Again, two approaches here (1) Compute only the sign bit of the shares somehow (rather than evaluating the complete circuit for ReLU) and then based on that do something lightweight (because depending on the outcome, we either need to share 0 or fresh shares of an already existing vector) (2) Do usual garbled circuit approach or convert to boolean shares and compute it. I feel there might be an efficient solution in the former framework, though I have not been able to come up with any concrete ideas.

Details of conversion schemes across related work:

- **ABY [3]:** I assume we need to convert A2Y and Y2A (not sure if Boolean circuits are faster). Among their contributions, ABY [3] recognize that among the techniques to implement Y2A, implementing Y2A as a composition of Y2B ("essentially free") and B2A ("cheaper in terms of communication and computation") turns out to be most effective. B2A is also one of their contribution.
  Conventional B2A is performed using subtraction circuits. ABY improve on the performance by using C-OT where the correlation is the power of 2 corresponding to the bit-value in consideration. For l-bit boolean shares, they improve performance of B2A to $6l$ symmetric key computations and $l \cdot k + (l^2 + l)/2$-bits of communication where k is the symmetric key security parameter (compared to either $O(l)$-gates and $O(l)$-depth or $O(l \log_2 l)$-gates and $O(\log_2 l)$-depth.).
- **Payman [4]:** Slightly terse in terms of the details of the NN training. Apart form the ReLU, all operations are additions (simple) and multiplications (using multiplicative triplets). As for the ReLU, they use Yao's garbled circuit which adds the two shares and outputs the most significant bit. For the conversion, there is insufficient detail in the paper itself but they refer to ABY for the schemes, so I presume they use the same conversion protocols as ABY (there is no mention of using anything different from them). Will confirm this when I look at the code.
- **NDSS'14 [1]:** They look at prediction only. The other property of their scheme is that usually each input is completely with one party (rather than having shares) and together they need to compute something. Hence their techniques are somewhat different, for ex: to compute max element of an array, they have one party permute the data and then they compute the max in that permuted data and then the first party inverts the permutation to get back the index of the maximum element. Only in Type 1 conversions of their paper they use OT along with garbled circuits to compute which party's input is larger. They use some older constructions of OT and garbled circuits and I believe ABY is much more recent work and have more efficient protocols. I have looked at this a while ago but will give it a deeper look to see if any techniques here can help us solve our sub-problems. Side note: to handle floats, they simply support finite precision by multiplying all floats by a constant like 2^{52}.

3. Practical Considerations

Typical Neural Networks over MNIST have:

TABLE 1: Diagram describing the flowchart of the forward pass.

	P_1	P_2	P_3	P_4	
0					For $l : 1 \rightarrow L$
1	$\langle a^l \rangle_0, \langle w^l \rangle_0$	$\langle a^l \rangle_0, \langle w^l \rangle_1$	$\langle a^l \rangle_1, \langle w^l \rangle_0$	$\langle a^l \rangle_1, \langle w^l \rangle_1$	Shared inputs
2	$\langle z^l \rangle_0 = \langle w^l \rangle_0 \cdot \langle a^l \rangle_0$	$\langle z^l \rangle_1 = \langle w^l \rangle_0 \cdot \langle a^l \rangle_1$	$\langle z^l \rangle_2 = \langle w^l \rangle_1 \cdot \langle a^l \rangle_0$	$\langle z^l \rangle_3 = \langle w^l \rangle_1 \cdot \langle a^l \rangle_1$	Compute $W \cdot X$
3					ReLU $\sigma(\sum_{k=0}^{3} z_k)$

TABLE 2: Specifics of ReLU operation. This has a security issue depending on how the 4to2 party sharing is done.

	P_1, P_2	P_3, P_4	
1	$\langle \gamma^l \rangle_0 = \langle z^l \rangle_0 + \langle z^l \rangle_1$	$\langle \gamma^l \rangle_1 = \langle z^l \rangle_2 + \langle z^l \rangle_3$	$4 \rightarrow 2$ conversion
2	$A2Y(\langle \gamma^l \rangle_0)$	$A2Y(\langle \gamma^l \rangle_1)$	A2Y conversion
3	$\langle \max(0, \langle \gamma^l \rangle_0 + \langle \gamma^l \rangle_1) \rangle_0$	$\langle \max(0, \langle \gamma^l \rangle_0 + \langle \gamma^l \rangle_1) \rangle_1$	ReLU computation
4	$Y2A(\langle \max(0, \langle \gamma^l \rangle_0 + \langle \gamma^l \rangle_1) \rangle_0)$	$Y2A(\langle \max(0, \langle \gamma^l \rangle_0 + \langle \gamma^l \rangle_1) \rangle_1)$	Y2A conversion
∞			Cost function?

- 2 Layers (i.e., just one hidden layer)
- Generally 300 or 1000 dimensional hidden layer.
- Achieve test error rates $\in (1, 5)$
- Common cost function: Mean Squared Error (MSE), Cross Entropy Function.

3.1. Back Prop

For the back prop, we begin by computing the error in the last layer and then propagate it backwards. We need to compute (1) The error in the last layer δ^L (2) An equation to see how error propagates backwards $\delta^{l+1} \rightarrow \delta^l$ (3) Compute the partial gradients of cost w.r.t weights to update weights in stochastic gradient descent.
Might need derivative of ReLU: (1+sgn(x))/2. As in Payman, we can compute ReLU and its derivative in the same circuit.
extend to CNN?

References

[1] R. Bost, R. A. Popa, S. Tu, and S. Goldwasser. Machine learning classification over encrypted data. In *Symposium on Network and Distributed System Security (NDSS)*, 2015.

[2] I. Damgård, M. Geisler, and M. Krøigaard. Homomorphic encryption and secure comparison. 2008.

[3] D. Demmler, T. Schneider, and M. Zohner. Aby-a framework for efficient mixed-protocol secure two-party computation. In *Symposium on Network and Distributed System Security (NDSS)*, 2015.

[4] P. Mohassel and Y. Zhang. Secureml: A system for scalable privacy-preserving machine learning. In *IEEE Symposium on Security and Privacy (S&P)*, 2017.

[5] M. Nielsen. Neural networks and deep learning. 2017.

[6] T. Nishide and K. Ohta. Multiparty computation for interval, equality, and comparison without bit-decomposition protocol. In *Public Key Cryptography*, 2007.