HEAAN library

HEAAN

Kyoohyung Han

May 23, 2018

Seoul National University

HEAAN scheme

HEAAN library is a library that supports operations between encrypted array of complex numbers. The Security of this scheme is decided by `logQ,logN` with fixed standard deviation $\sigma = 3.2$. If you use Martin's LWE parameter estimator, you can check the security of the scheme [1].

$$\texttt{encode} : (m_1, \ldots, m_\ell) \in \mathbb{C}^\ell \Rightarrow \lfloor \Delta \cdot m(x) \rceil \in \mathbb{Z}[X]/(X^N + 1).$$

The ciphertext is pair of polynomial $(a(x), b(x)) \in \mathcal{R}_Q$ such that

$$b(x) = -a(x)s(x) + \lfloor \Delta \cdot m(x) \rceil + e(x) \text{ for } m(x) \in \mathbb{R}[X]/(X^N + 1)$$

for secret key $s(x)$, $\mathcal{R} = \mathbb{Z}[X]/(X^N + 1)$ and $\mathcal{R}_Q = \mathcal{R}/Q\mathcal{R}$.

[1] https://bitbucket.org/malb/lwe-estimator

Functionality

- `encode`: input $\vec{m} \in \mathbb{C}^\ell$, output integer polynomial $m(x)$.
- `decode`: input $m(x)$, output array of complex number $\vec{m}$.
- `encrypt`: input $\vec{m}$, encode it and return a ciphertext $(a(x), b(x))$.
- `decrypt`: input $(a(x), b(x))$, decrypt and decode it and return $\vec{m}$.
- `add`: input two ciphertext, return encryption of $\vec{m}_1 + \vec{m}_2$.
- `square`: input a ciphertext, return encryption of $\vec{m} \circ \vec{m}$.
- `mult`: input two ciphertext, return encryption of $\vec{m}_1 \circ \vec{m}_2$.
- `rotate`: input a ciphertext, return encryption of rotated $\vec{m}$.

[2]

[2] $\circ$ means element-wise multiplication

How to use HEAAN library?

Pre-computations

First, we need to decide Δ and depth of target circuit L. This will decide $\mathtt{Q} = \Delta^L$. Using security parameter λ and LWE parameter estimator, we will choose N which is the dimension of polynomial [3].

```
Context context(logN, logQ);
```

$\Rightarrow$ compute matrices for encoding and decoding. This class includes `encode` and `decode` function.

```
ZZX mx = context.encode(mvec,slots,pBits);
```

$\Rightarrow$ this will return encoded result which is an integer polynomial $m(x)$. Here `pBits` is logarithm of Δ with base 2.

[3] https://github.com/kimandrik/HEAAN

Key Generation

In HEAAN scheme, we need additional public key which is a pair of polynomial in $\mathcal{R}_{PQ}^2$. Here P is called special modulus that has same bit size with Q[4].

```
SecretKey sk(logN, h);
```

$\Rightarrow$ generate a sparse secret polynomial $s(x)$ [5].

```
Scheme scheme(sk, context);
```

$\Rightarrow$ generate two public keys in below:

$$\mathsf{pk}_{enc} = (a(x), -a(x)s(x) + e(x)) \in \mathcal{R}_{PQ}^2$$

$$\mathsf{pk}_{mult} = (a(x), -a(x)s(x) + P \cdot s^2(x) + e(x)) \in \mathcal{R}_{PQ}^2$$

[4] Now, the security is based on hardness of RLWE problem with $(2\mathtt{logQ}, \mathtt{logN}, \sigma)$
[5] This polynomial has only h number of non-zero elements $\in \{-1, 1\}$

Key Generation

If users of this library need slot rotation functionality, they need to generate public keys corresponding it.

```
scheme.addConjKey(sk);
scheme.addLeftRotKey(sk,i);
scheme.addRightRotKey(sk.i);
```

$\Rightarrow$ generate public keys for left rotation and right rotations.

$$\mathsf{pk}_{leftRot,i} = (a(x), -a(x)s(x) + Ps(x^k) + e(x)) \in \mathcal{R}_{PQ}^2$$

for $k = 5^i \bmod 2N$ ($k = 2N - 1$ in case of conjugation and $k = 5^{-i}$ in case of right rotation). This key is used for left rotation of our plaintext array with index i. If you call `scheme.addLeftRotKeys` and `scheme.addRightRotKeys` is will generate all keys for power of two rotations.

Homomorphic Addition

There are various version of homomorphic addition. Notice that we can do addition between plaintext and ciphertext also.

```
cipher3 = scheme.add(cipher1, cipher2);
scheme.addAndEqual(cipher1, cipher2);
cipher2 = scheme.addConst(cipher1, const);
```

⇒ these algorithms are quite fast. addition between two ciphertexts only takes about 10ms to 20ms for commonly used parameters. Note that `addAndEqual` will update first input to the result ciphertext, and this is slightly faster because of memory allocation timing.

Homomorphic Addition

```
Homomorphic Addition time = 19.48 ms
------------------
dvec: 0.429113 (expected = 0.428984)
dvec: 0.886836 (expected = 0.886822)
dvec: 1.55806 (expected = 1.55808)
dvec: 0.683899 (expected = 0.683982)
dvec: 1.13934 (expected = 1.13933)
```

Homomorphic Multiplication

There are various version of homomorphic multiplication. Notice that we can do multiplication between plaintext and ciphertext also.

```
cipher2 = scheme.square(cipher1);
cipher2 = scheme.squareAndEqual(cipher1);
cipher3 = scheme.mult(cipher1, cipher2);
scheme.multAndEqual(cipher1, cipher2);
cipher2 = scheme.multByConst(cipher1, const);
```

$\Rightarrow$ those algorithms except `multByConst` use pk_{mult} that we generate at the first step. This takes about 100ms to 1000ms depends on the parameters like depth L and Δ.

Homomorphic Rescaling

The result of homomorphic multiplication has scaling factor Δ^2. So we need to re-scaling it. Here `pBits` is logarithm of Δ with base 2.

```
cipher2 = scheme.reScaleBy(cipher1, pBits);
scheme.reScaleByAndEqual(cipher1, pBits);
```

$\Rightarrow$ this will change to scaling factor Δ^2 to Δ. The result ciphertext has modulus Q/Δ.

Homomorphic Multiplication and Rescaling

```
Homomorphic Multiplication time = 315.924 ms
------------------
dvec: -0.584994 (expected = -0.584978)
dvec: -0.126959 (expected = -0.126956)
dvec: 0.361808 (expected = 0.361819)
dvec: 0.019111 (expected = 0.0191251)
dvec: 0.0169387 (expected = 0.0169513)
```

Homomorphic Rotation

Because the plaintext is an array of real (or complex) element, we need rotation (or shifting) operation for efficiency.

```
cipher2 = scheme.leftRotateBy(cipher1, i);
scheme.leftRotateByAndEqual(cipher1, i);
cipher2 = scheme.rightRotateBy(cipher1, i);
scheme.rightRotateByAndEqual(cipher1, i);
```

$\Rightarrow$ each rotation need corresponding public key. If there is no corresponding public key, this will combine power of two shifting automatically.

Homomorphic Rotation

```
Homomorphic Rotation time = 143.653 ms
-------------------
dvec: 0.968041 (expected = 0.96837)
dvec: 0.775928 (expected = 0.775818)
dvec: 0.582578 (expected = 0.582289)
dvec: 0.70814 (expected = 0.708524)
dvec: 0.666356 (expected = 0.666725)
```

Put All together

```
// Key Generation
Context context(logN, logQ);
SecretKey sk(logN);
Scheme scheme(sk, context);
scheme.addLeftRotKey(sk, 1);
// Encrypt
Ciphertext cipher1 = scheme.encrypt(mvec1, slots, pBits, logQ);
Ciphertext cipher2 = scheme.encrypt(mvec2, slots, pBits, logQ);
// Homomorphic Operations
Ciphertext addCipher = scheme.add(cipher1, cipher2);
Ciphertext multCipher = scheme.mult(cipher1, cipher2);
scheme.reScaleByAndEqual(multCipher, pBits);
Ciphertext rotCipher = scheme.leftRotate(cipher1, 1);
// Decrypt
complex<double>* dvecAdd = scheme.decrypt(sk, addCipher);
complex<double>* dvecMult = scheme.decrypt(sk, multCipher);
complex<double>* dvecRot = scheme.decrypt(sk, rotCipher);
```