Deep Learning with Homomorphic Encryption

https://youtu.be/ZFc5Be0dytI

HSU 한성대학교 HANSUNG UNIVERSITY

CryptoCraft LAB
https://crypto.modoo.at

Contents

CryptoCraft LAB

Homomorphic Encryption

$$Enc(a+b) = Enc(a) + Enc(b)$$
$$Enc(a \times b) = Enc(a) \times Enc(b)$$

- 평문을 연산하여 암호화 한 것과 평문을 암호화 하여 연산한 결과가 동일
- 비밀 데이터인 a, b를 노출하지 않으면서 암호화 한 상태로 연산 가능
- LWE(Learning with Error), RLWE, CRT(중국인의 나머지 정리) 등에 기반

Homomorphic Encryption

Partial Homomorphic Encryption

- 덧셈 or 곱셈 중 하나만 지원

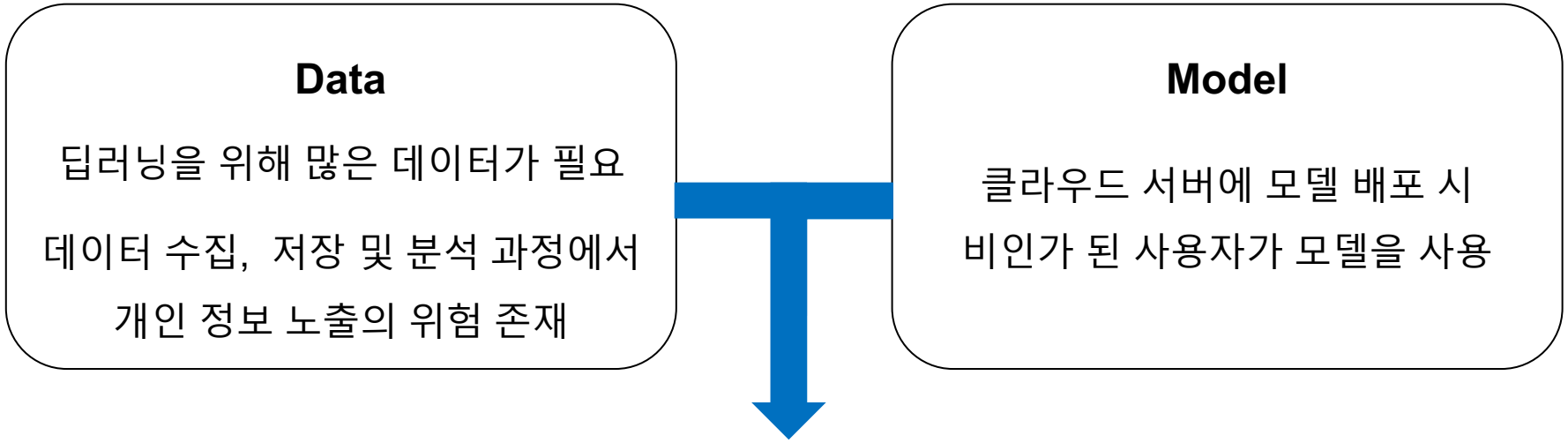
Somewhat Homomorphic Encryption

- 덧셈, 곱셈 모두 지원
- 제한된 연산 (차수 낮은 다항식)
  → 암호문에 추가되는 noise들이
  연산을 반복하여 너무 커질 경우
  메시지 훼손되므로 연산에 제한이 있음

Fully Homomorphic Encryption

- 덧셈, 곱셈 모두 지원
- boot strapping 통해 제한 없이 계속 연산 가능
  → 암호화된 비밀키를 통해 암호문을 복호화하여
  해당 암호문의 noise를 없앤 후 재암호화

DL privacy problem



Deep Learning에 Homomorphic Encryption 적용

Deep learning for data privacy

1. data를 동형 암호화

   다음 슬라이드..

2. 다자간 계산

   중앙 서버가 학습에 참여하는 여러 client에게 model을 나눠주고 각자 local에서 자신의 데이터로 계산하여 서버로 전송

3. data 암호화 + 서버가 layer 단위 계산

   data를 암호화하여 서버로 전송

   서버는 1 layer씩 연산 → 가중치와 데이터간의 연산만 수행 후 client에 반환

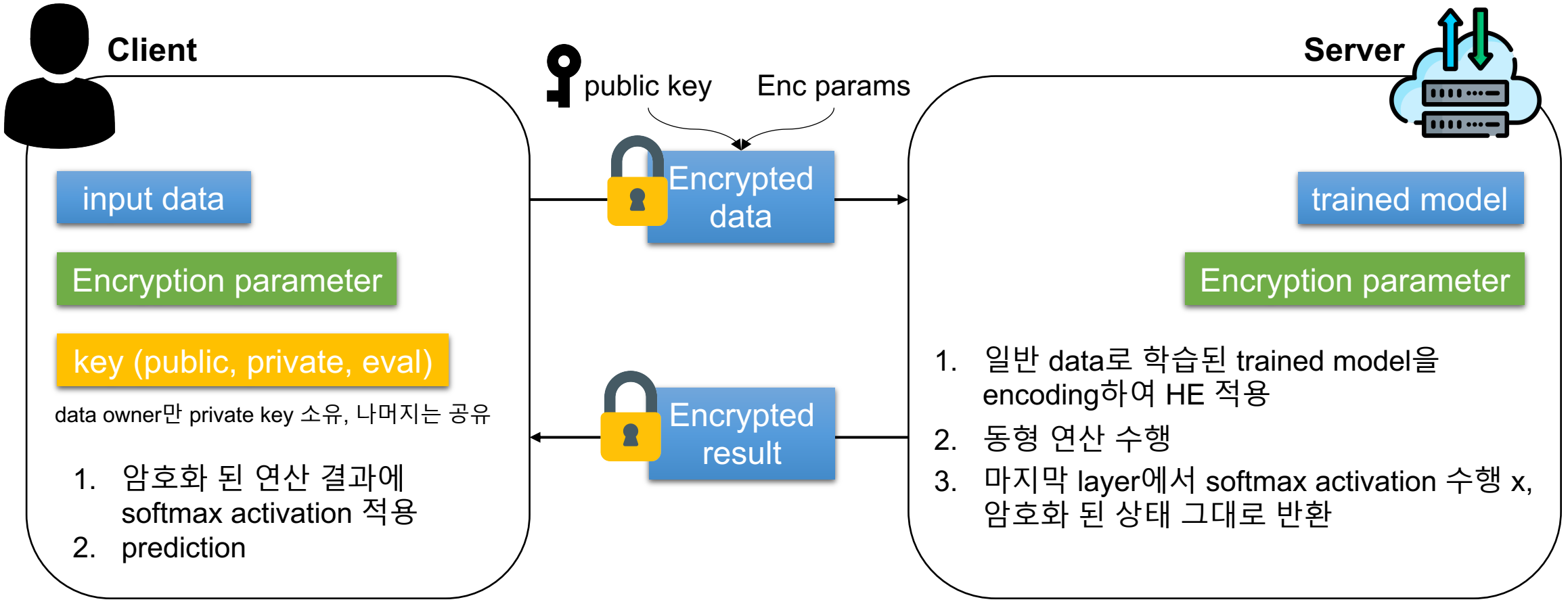
   client가 서버로부터 받은 값을 복호화하여 activation function 적용

   해당 값을 다시 서버에 다음 layer의 입력 데이터로 제공

*activation function은 암호화 된 상태의 데이터 간의 연산이 필요하여 3번의 경우에서는 서버에서 연산 불가능

→ 복호화하여 평문 상태로 연산해야 함 (1번의 경우 복호화 과정 없이 가능)

DL with HE Scenario



사용된 data와 결과는 key 소유자만이 확인 가능

Encryption Parameter setting

- noise 증가와 계산 효율 등을 고려

→ 곱셈을 얼마나 할 수 있는가 가 주요 고려 요소

- 해당 parameter들이 noise budget, message size, security level 등 결정

```
def _build_context(self, config):

    #set up encryption parameters and context
    parms = EncryptionParameters()
    parms.set_poly_modulus(config['poly_modulus'])
    parms.set_coeff_modulus(seal.coeff_modulus_128(config['coeff_modulus']))
    parms.set_plain_modulus(1 << 18)
    context = SEALContext(parms)

    return context
```

```
config = {
    'poly_modulus':'1x^4096 + 1',
    'coeff_modulus':4096,
    'int_coeff':64,
    'fract_coeff':32,
    'base':2
    }
```

- poly_modulus → 1024, 2048, 4096, …, 32768 권장

보안 레벨에 주로 영향

해당 modulus가 클수록 안전해지지만, cipher text 크기 또한 증가

→ 연산 속도 저하, 메모리 공간 증가로 인한 비효율 발생 가능

- coeff_modulus → 128 : 128-bit security level (128, 192 권장)

noise budget에 영향을 미치는 요소 (클수록 noise budget 증가)

poly_modulus에 따라 결정

- plain_modulus

plain text size 결정

coeff_modulus와 함께 re-encrypted text의 noise budget 결정

Key Generation

```
def _create_keys(self, context):

    keygen = KeyGenerator(context)
    public_key = keygen.public_key()
    secret_key = keygen.secret_key()

    ev_keys16 = EvaluationKeys()
    keygen.generate_evaluation_keys(16, ev_keys16)

    return public_key, secret_key, ev_keys16
```

- public key : for encryption
- private key : for decryption
- evaluation key : for evaluation (re-encryption, operation ; mul, add)

  → private key로부터 생성

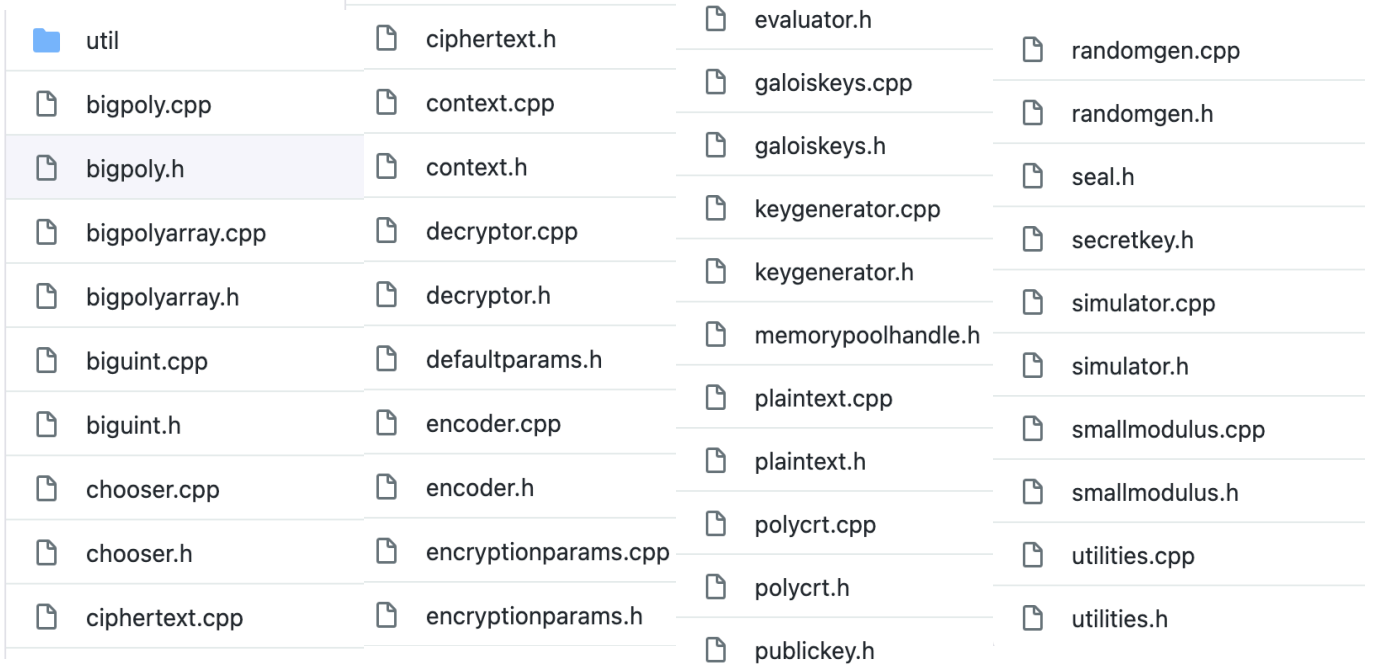
Encryption Handler

```
def _setup_members(self, context, config):

    self._encoder = FractionalEncoder(context.plain_modulus(),context.poly_modulus(),config['int_coeff'],config['fract_coeff'],config['base'])
    self._evaluator = Evaluator(context)

    pk, sk, self._ev_key = self._create_keys(context)
    self._encryptor = Encryptor(context, pk)
    self._decryptor = Decryptor(context, sk)
```

```
handler = EncryptionHandler(config)
```



Encoding

- 기존 neural network는 실수 사용, HE는 다항식 연산

  → encoding 통해 기존의 NN 수정 사용 필요

- 이 과정을 통해 plain data를 다항식으로 변환
- encryption 수행 전 encoding 수행
- bias 덧셈이나 zero padding 시 추가되는 0 등도 encoding
- Library에 구현되어 있음

```
self._encryptor.encrypt(self._encoder.encode(0),enc_0)

self._evaluator.multiply_plain(temp_x, self._encoder.encode(plain_b[column,i]))
```

encoder.cpp

encoder.h

operation

```
def _fully_connect(self,x,y):
    retval = np.zeros((y.shape[1]),dtype=np.dtype('O'))

    for i in range(y.shape[1]):
        res = self._Ciphertext()
        self._encryptor.encrypt(self._encoder.encode(0),res)
        for j in range(y.shape[0]):
            temp_x = self._Ciphertext(x[j])
            self._evaluator.multiply_plain(temp_x, self._encoder.encode(y[j,i]))
            self._evaluator.add(res,temp_x)
        retval[i] = res
    return retval
```

- dot product, sliding window 등도 인코딩되어 구현되어 있음
- 암호화된 상태로 곱셈, 덧셈 가능
- 연산은 evaluator통해 수행

Re-encrypt – boot strapping

```
def re_encrypt(self, x):

    shape = x.shape
    prod = 1
    for s in shape:
        prod = prod*s

    x = np.reshape(x,prod)

    temp_x = self.get_matrix(x)

    x = []
    for i in range(prod):
        x.append(Ciphertext())
        self.encryptor.encrypt(self.encoder.encode(temp_x[i]), x[i])
    x = np.reshape(x,shape)

    return x
```

- 연산을 계속 수행할 경우 noise가 너무 커져서, noise budget이 0이 될 경우 noise 비트가 메시지 비트 부분으로 넘어감

  → 복호화 불가, 메시지 훼손
- 연산 횟수에 제한이 생김
- 따라서 복호화하여 noise 감소 후 다시 암호화하는 과정이 필요

  → bootstrapping → Fully Homomorphic Encryption 가능
- 여기서의 복호화에는 evaluation key 사용

Re-encrypt – boot strapping

```python
def get_matrix(self, mat):

    shape = mat.shape

    assert(len(shape) == 1 or len(shape) == 4)

    if len(shape)==1:
        p_c = np.zeros((shape[0]))
        for i in range(shape[0]):
            p_c[i] = self.decrypt_ciphertext(mat[i])



    else:
        p_c = np.zeros((shape[0],shape[1],shape[2],shape[3]))
        for i in range(shape[1]):
            for j in range(shape[2]):
                for k in range(shape[3]):
                    p_c[0,i,j,k] = self.decrypt_ciphertext(mat[0,i,j,k])

    return p_c
```

```python
def decrypt_ciphertext(self, cipher):

    plain = Plaintext()
    self.decryptor.decrypt(cipher, plain)
    pl =  self.encoder.decode(plain)

    return pl
```

Prediction

- server는 마지막 softmax activation 적용하지 않은 상태로 반환
- client에서 prediction / decryption 수행

  → result 보호/ data 보호

```
predictor = Predictor(op,debug= debug, handler= handler)
logits = predictor.predict_image(encrypted_image)
```

감사합니다

새해 복 많이 받으세요..