실험 7 CKKS Homomorphic Encryption 이해

작성자: 위두랑가
담당 조교 : 황현하,hien

1. 실험 목적

Homomorphic encryption 의 정의는 무엇인지 이해하고 TenSEAL library 를 통해 homomorphic encryption/decryption 실습을 수행

- Anaconda program 다운로드 및 virtual environment 생성
- TenSEAL library 설치
- CKKS encryption을 이용한 vector 및 metric 계산
- Context serialization and Security Key
- Ciphertext serialization
- Activation and Pooling in Homomorphic Encryption

2. 실험 전에 준비해야할 내용

a) Deep Learning as a Service Introduction

딥러닝 기술이 발달하며 클라우드 기반 정보 어플리케이션은 혁신을 거듭하고 있고 이에 따라 Deep Learning as a service(DLaas)를 사용할 수 있다. DLaas 시스템을 어떤 곳에 사용해야 하는지 많은 문제가 있다. 하지만 사용자의 보안이 DLaaS 시스템의 가장 큰 문제이다.

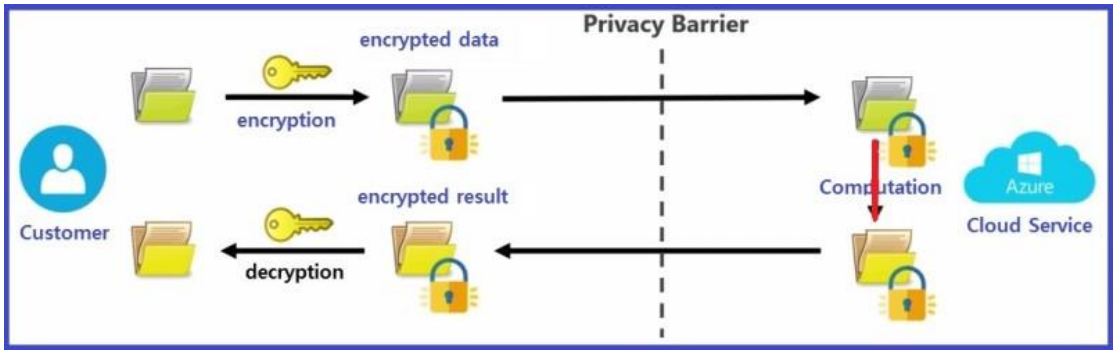


Figure. DLaaS System.

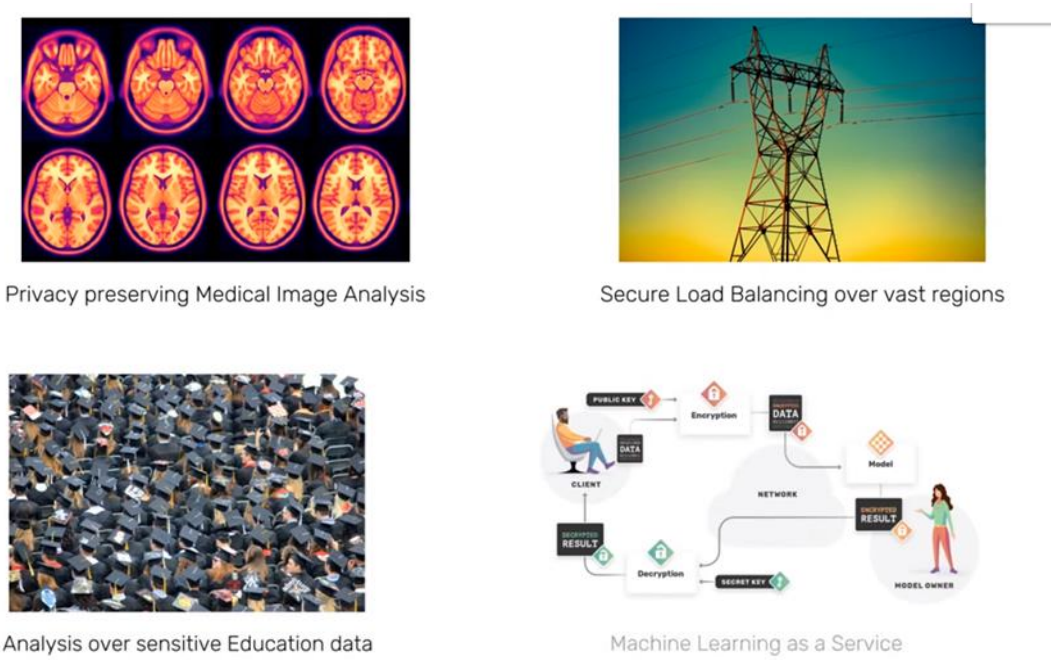


Figure. Examples of DLaaS System.

b) Homomorphic Encryption

Homomorphic Encryption 은 암호화된 데이터를 계산할 수 있게하는 encryption 이다. 이것은 암호화된 데이터에 대한 추론을 할 수 있으며, 서비스 제공자가 고객의 개인 데이터를 볼 수 없음을 의미한 homomorphic encryption 은 데이터와 서비스 제공자의 상호작용을 필요로하지 않는다.

회사에서 클라우드에 데이터를 업로드하고 싶지만 서비스 제공자를 신뢰할 수 없을 때 homomorphic encryption 을 사용할 수 있다. Homomorphic encryption 을 이용하면 회사가 데이터를 암호화하고 public key 와 함께 서버로 전송한다. 서버는 데이터를 받은 후 해독하지 않고 적절한 계산을 한 후 바로 암호화된 결과를 회사로 보낸다. 데이터의 주인이 해독된 자료를 볼 수 있는 유일한 사람이다.

그러나 homomorphic encryption 은 비싸다. 따라서 효율성 문제로 사용이 제한되어 있다.

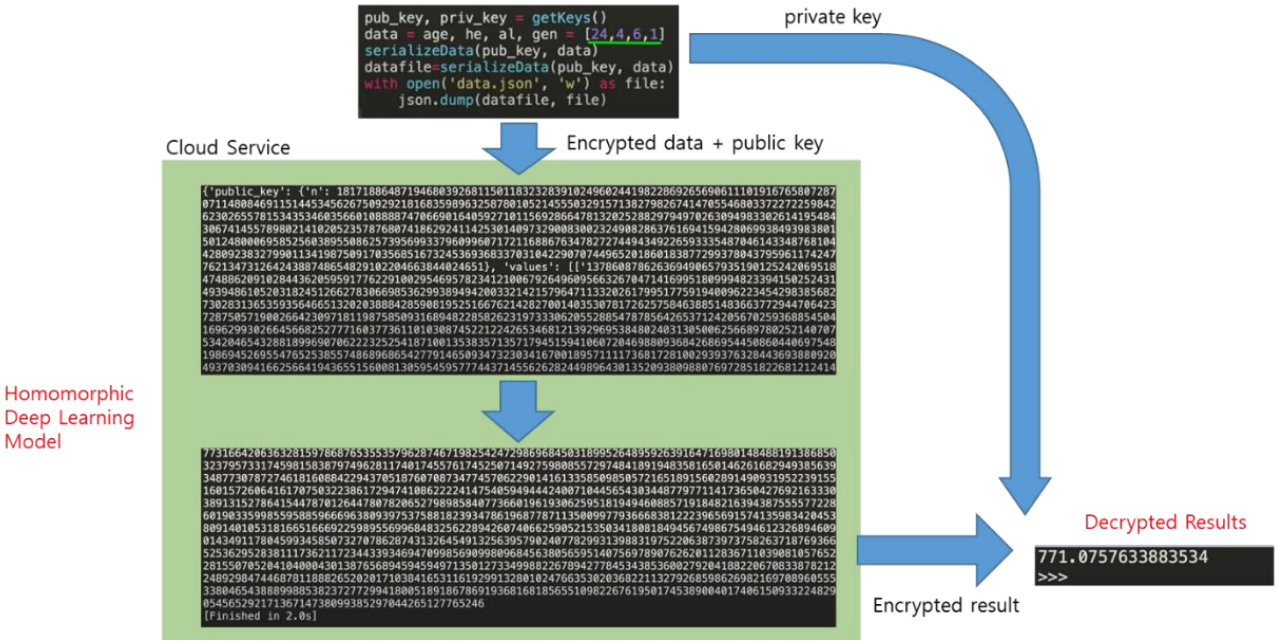


Figure. Examples about using homomorphic deep learning model.

c) TenSEAL library and CKKS Encryption

TenSEAL 은 Microsoft SEAL 기반으로 homologous encryption 을 수행할 수 있는 library 이다. TenSEAL 은 python API 를 통해 편의성을 제공하는 동시에 효율성을 유지하기 위해 대부분의 작업에서 C++를 사용한다.

TENSEAL: A LIBRARY FOR ENCRYPTED TENSOR OPERATIONS USING HOMOMORPHIC ENCRYPTION

https://dp-ml.github.io/2021-workshop-ICLR/files/18.pdf

TenSEAL

A library for doing homomorphic encryption operations on tensors

Figure. TenSEAL library

Encryption/Decryption of vectors of real numbers using CKKS

i. Cheon-Kim-Kim-Song (CKKS) scheme

Cheon-Kim-Kim-Song(CKKS)는 실수와 더불어 복소수의 대략적 계산까지 지원하는 Leveled Homomorphic Encryption 의 scheme 이다.

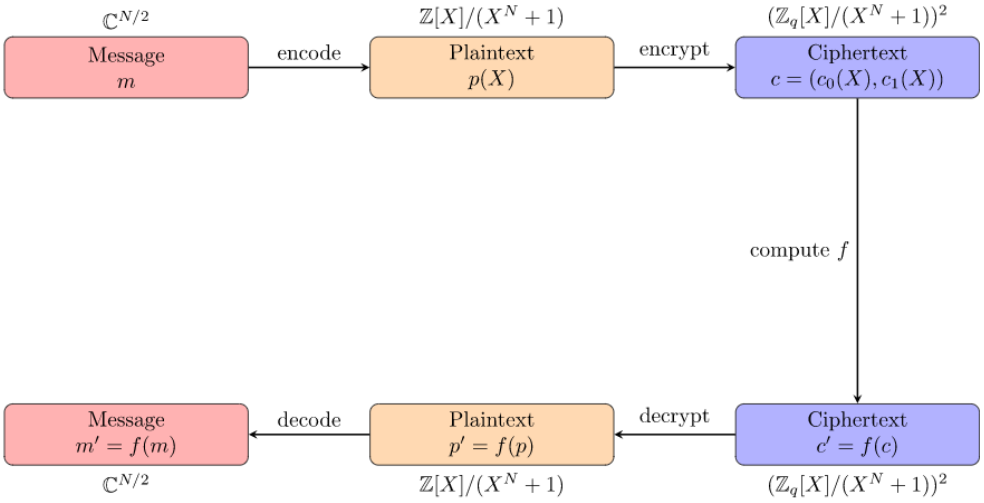


Figure. The Cheon-Kim-Kim-Song(CKKS) scheme diagram

이 연산은 CKKS 체계를 사용하여 암호화되고 계산될 복합 또는 실수 벡터를 plaintext polynomial 로 인코딩한다.

CKKS scheme 의 encryption 과 decryption 은 plaintext polynomial 을 ciphertext 로 변환시킨다.

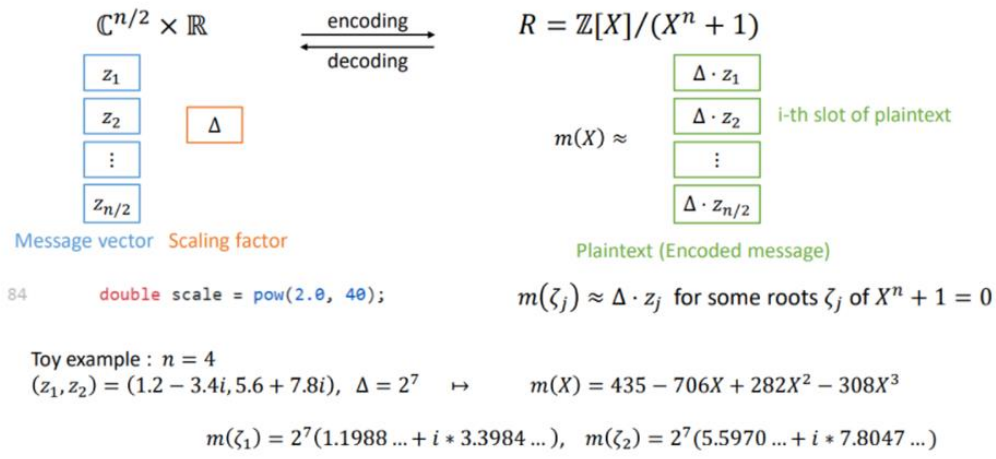


Figure. The diagram shows the detailed encoding-decoding flow.

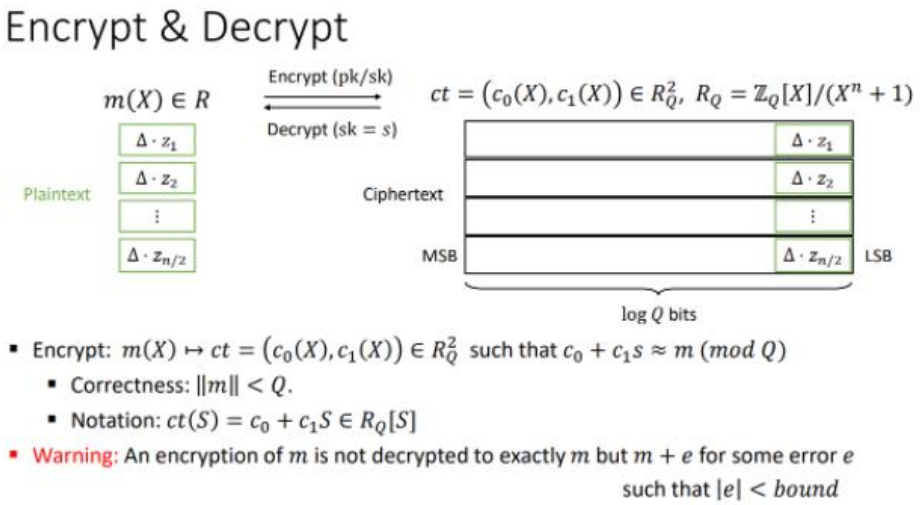
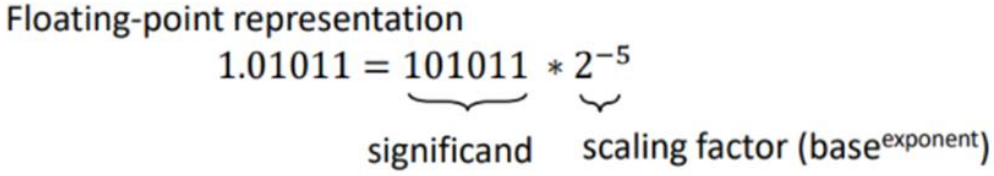


Figure. The diagram shows the detailed encryption-decryption flow.

Scaling factor(polynomial modulus degree 와 coefficient modulus sizes)인 parameter 를 CKKS scheme 은 필요로 한다. Scaling factor 는 숫자의 binary 표기를 위한 encoding precision 을 정의한다.



The polynomial modulus(N in the diagram) directly affects:

- Plaintext polynomial 에서 coefficients 의 갯수
- ciphertext elements 의 크기.
- The computational performance of the scheme (클수록 안좋다).
- 보안레벨 (클수록 좋다).

TenSEAL 에서 polynomial modulus 의 degree 는 2 의 제곱이어야한다. (e.g 1024, 2048, 4096...)

SEAL, using a list of binary sizes, will generate a list of primes of those binary sizes, called the coefficient modulus. The coefficient modulus directly affects:

- ciphertext elements 의 사이즈
- The length of the list indicates the level of the scheme (or the number of encrypted multiplications supported).
- The security level (bigger is worse).

CKKS scheme has 4 types of keys: the secret key, the public encryption key, the relinearization keys and the Galois keys (optional).

secret key 는 decryption 에 사용되고 public key 는 encryption 에 사용된다. Re-linearization key 는 ciphertexts 의 사이즈를 2 로 줄이는 re-linearization 에 필요한 public key 중 하나이다. The re-linearization keys created by the secret key owner. Galois keys 는 batched ciphertext 에서 encrypted vector rotation operations 을 하는데 필요한 public keys 의 종류 중 하나이다.

Theory: CKKS Keys

The secret key

The secret key is used for decryption. DO NOT SHARE IT.

The public encryption key

The key is used for encryption in the public key encryption setup.

The re-linearization keys

Every new ciphertext has a size of 2, and multiplying ciphertexts of sizes K and L results in a ciphertext of size $K+L-1$. Unfortunately, this growth in size slows down further multiplications and increases noise growth.

Re-linearization is the operation that reduces the size of ciphertexts back to 2. This operation requires another type of public keys, the re-linearization keys created by the secret key owner.

The operation is needed for encrypted multiplications. The plain multiplication is fundamentally different from normal multiplication and does not result in ciphertext size growth.

The Galois Keys(optional)

Galois keys are another type of public keys needed to perform encrypted vector rotation operations on batched ciphertexts.

One use case for vector rotations is summing the batched vector that is encrypted.

Theory: CKKS internal operations

These operations are automatically executed by TenSEAL, unless the user opts-out.

Re-linearization

The operation is executed automatically by TenSEAL after each encrypted multiplication.

The operations re-linearize a ciphertext, reducing its size down to 2. If the size of encrypted ciphertext is $K+1$, the given re-linearization keys need to have a size of at least $K-1$.

Rescaling

The operation is executed automatically by TenSEAL after each encrypted or plain multiplication.

The approximation error exponentially grows with the number of homomorphic multiplications. To overcome this problem, most HE schemes usually use a modulus-switching technique. In the case of CKKS, the modulus-switching procedure is called rescaling. Applying the rescaling algorithm after a homomomorphic multiplication, the approximation error grows linearly, not exponentially.

Given a ciphertext encrypted modulo $q_1...q_k$, this function switches the modulus down to $q_1...q_{k-1}$ and scales the message down accordingly.

This step consumes one prime from the coefficient modulus. And when you consume all of them, you won't be able to perform more multiplications.

3. 실습 실험을 진행하기 위한 환경 세팅

실험 절차:

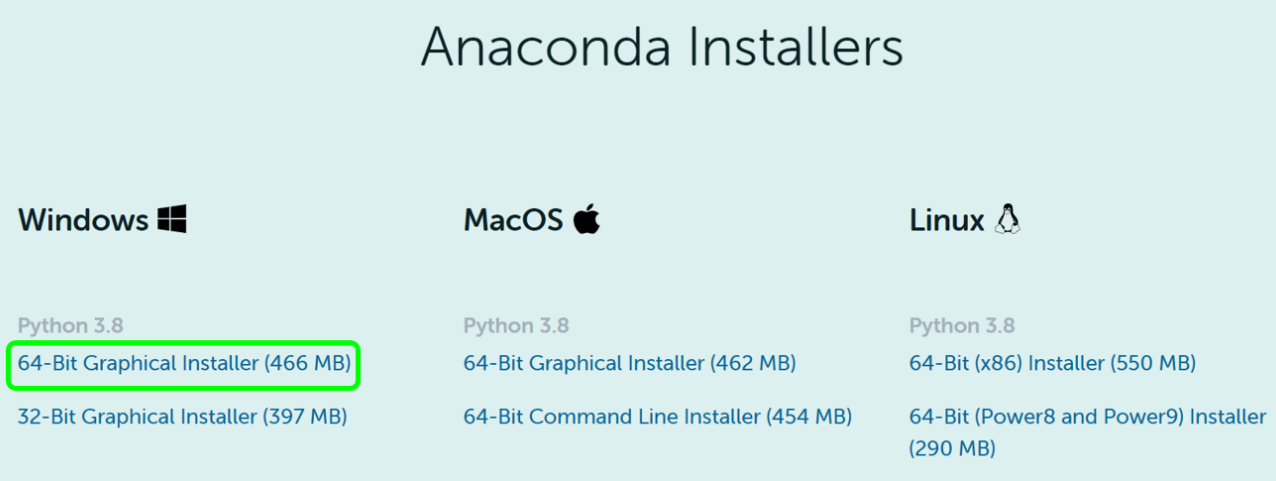
- Anaconda program 다운로드 및 virtual environment 생성
- TenSEAL library 설치
- CKKS encryption을 이용한 vector 및 metric 계산
- Context serialization and Security Key
- Ciphertext serialization
- Activation and Pooling in Homomorphic Encryption

a) How to install Anaconda program, TenSEAL library and create a virtual environment

Anaconda 설치

Go to this website link: https://www.anaconda.com/products/individual

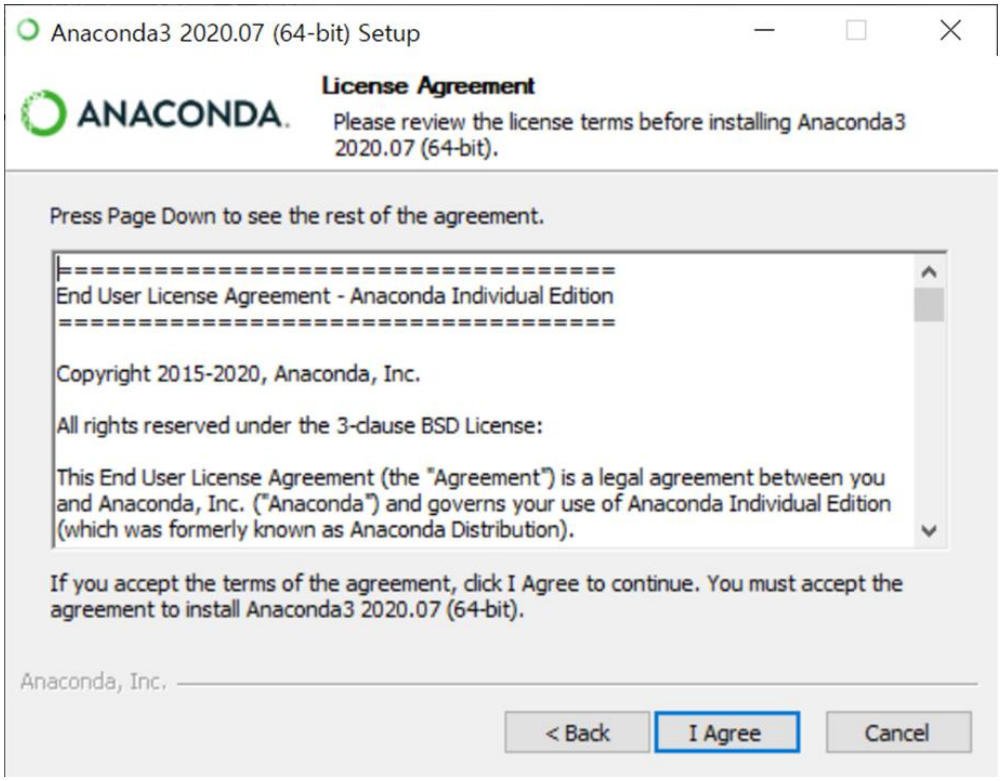
PC 에 맞는 버전 다운로드하기



다운로드 후 start setup 클릭

Next 클릭

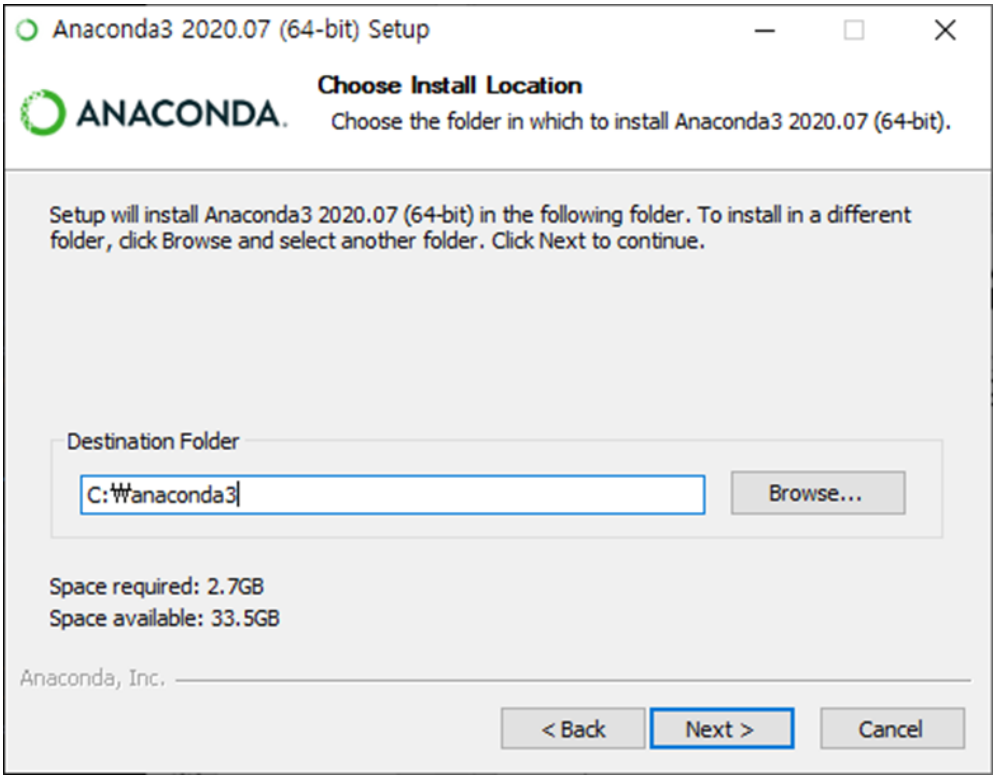
I Agree 버튼 클릭



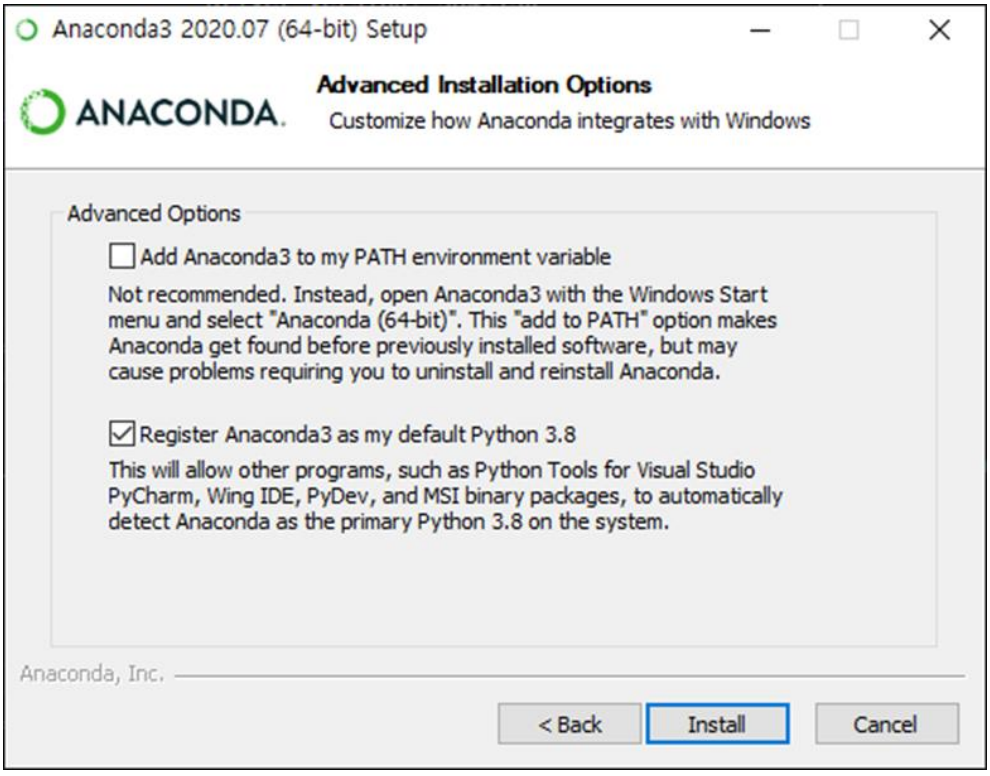
저장 위치 선택:

위치를 변경하고 싶지 않다면 Next 클릭 아니면 Browse를 클릭하고 폴더를 선택해라.

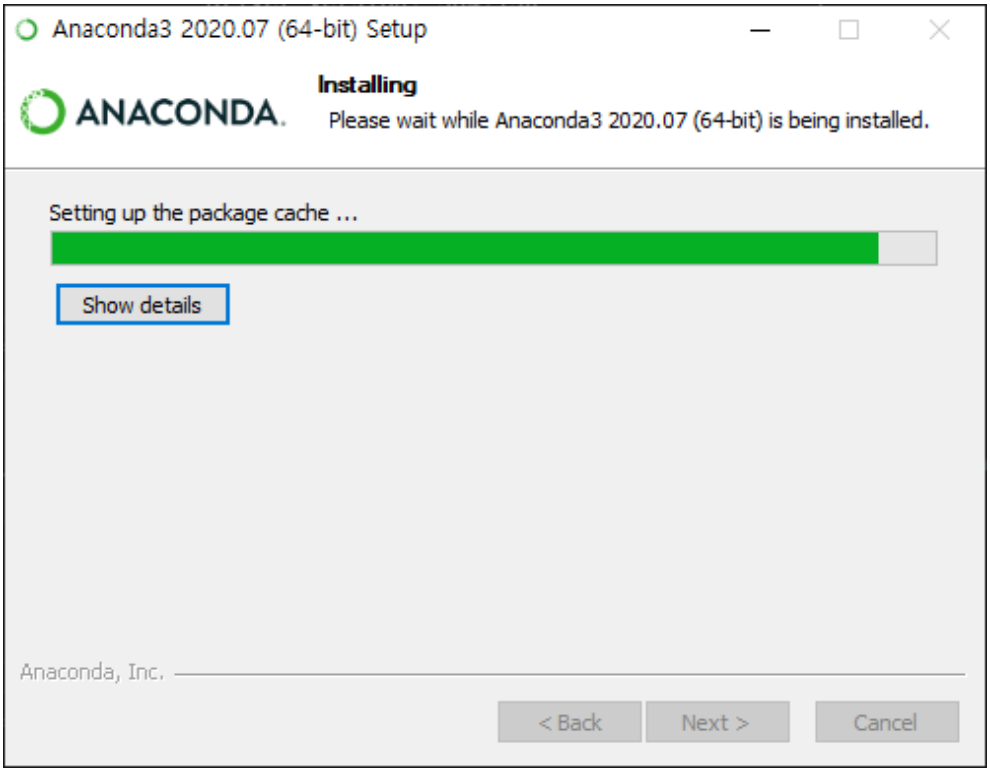
계속하려면 Next를 클릭.



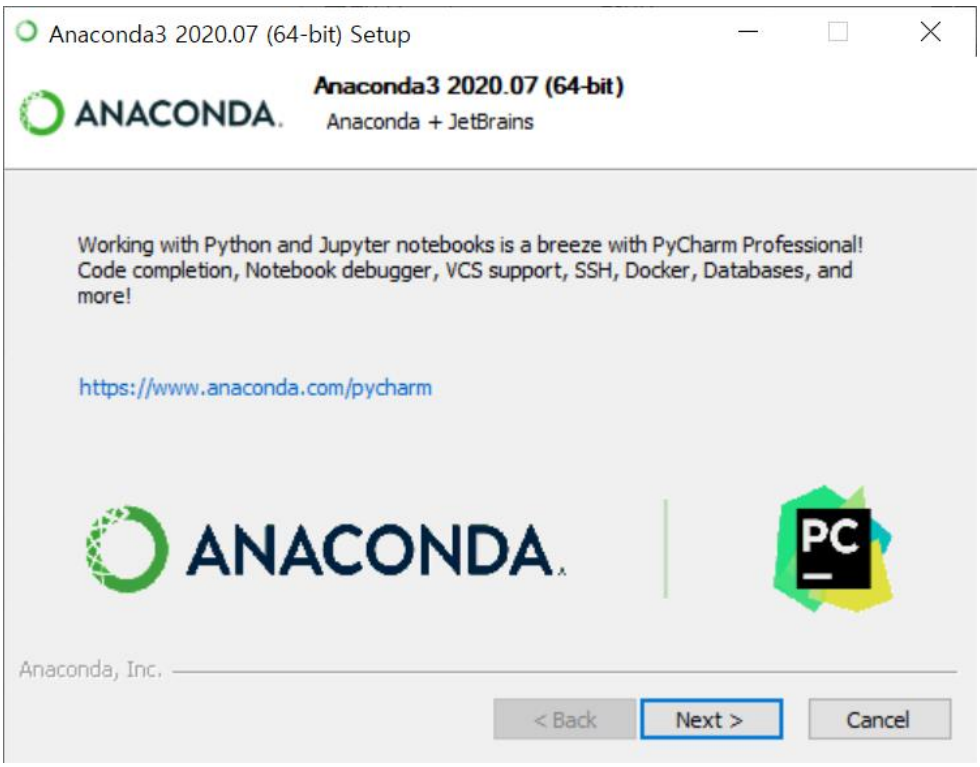
Choose the option like the image. Install 클릭



Wait for installing.

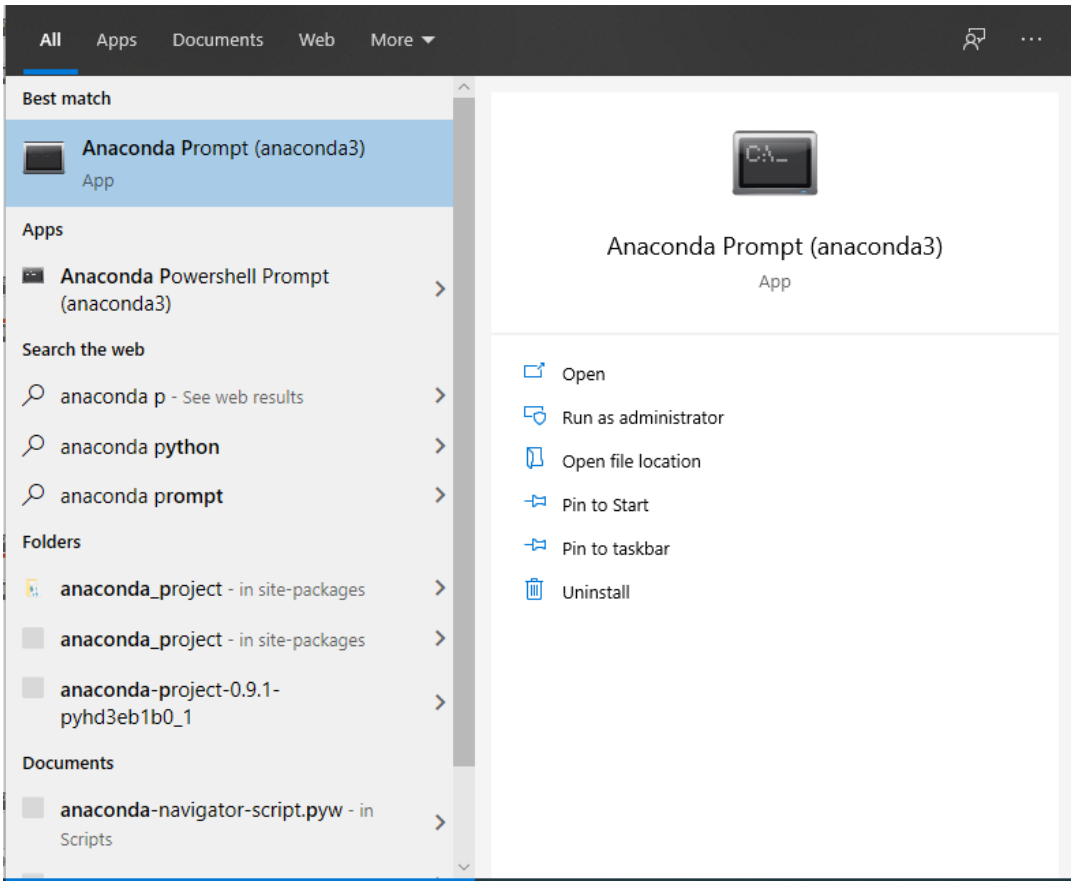


Click Next to continue.

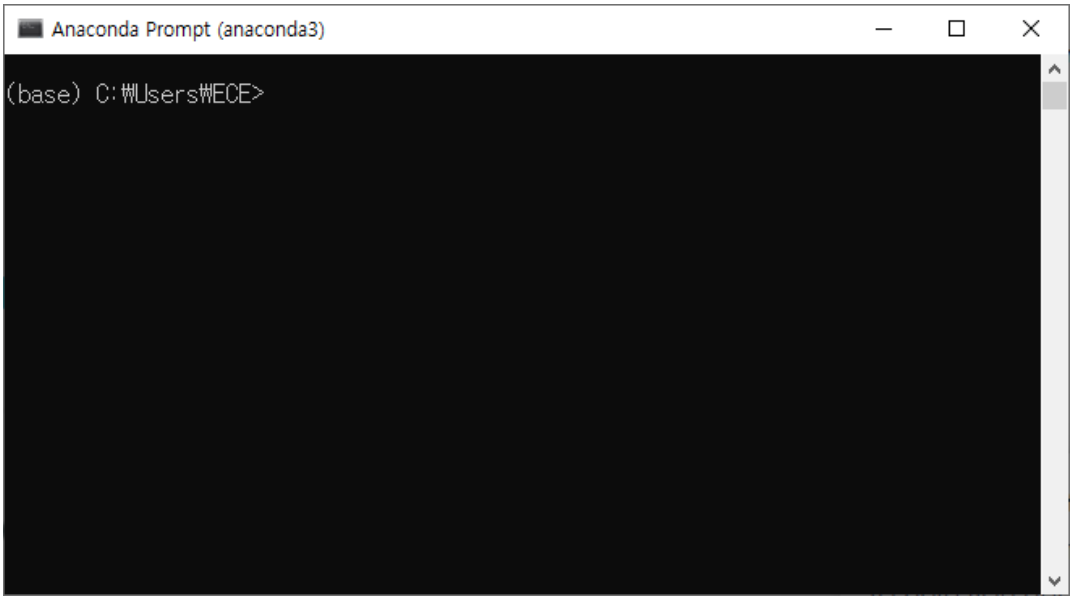


Finish 클릭. You've already finished installing Anaconda program.

Anaconda 를 실행하려면, "Anaconda"를 검색해서 Anaconda Prompt 를 클릭한다 (like image)

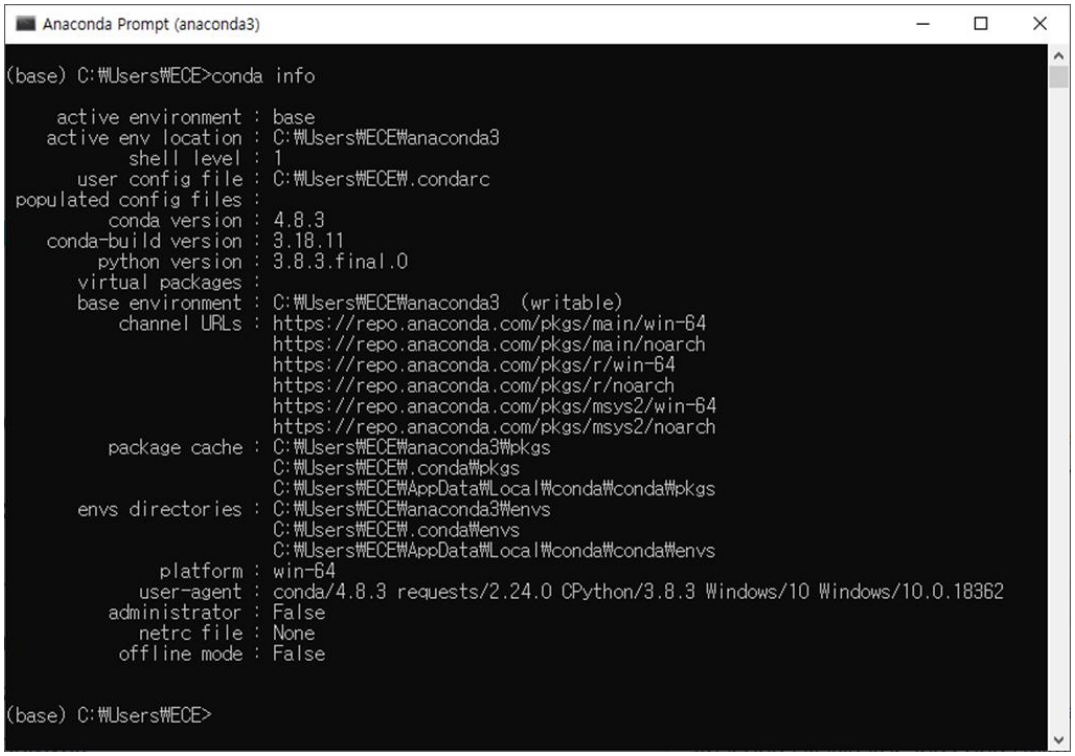


이 스크린이 나타날 것이다.

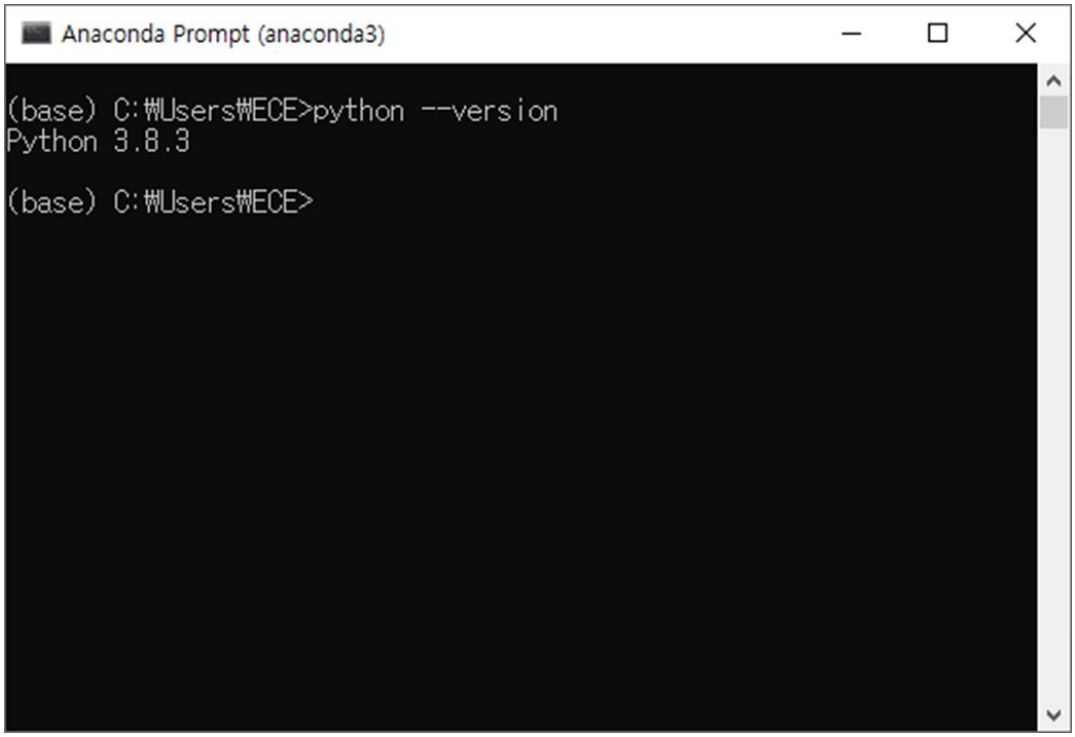


Anaconda 설치 확인: run command: >conda info

Anaconda 의 정보가 창에 뜬다.



Python version 확인하기: run command: >python - -version



b) virtual environment 생성 (conda virtual environment)

- virtual environment 생성:

  Run command: >conda create --user --name=tenseal python=3.7

- enviroment 활성화:

  Run command: >conda activate tenseal

- Virtual Environment 끄기:

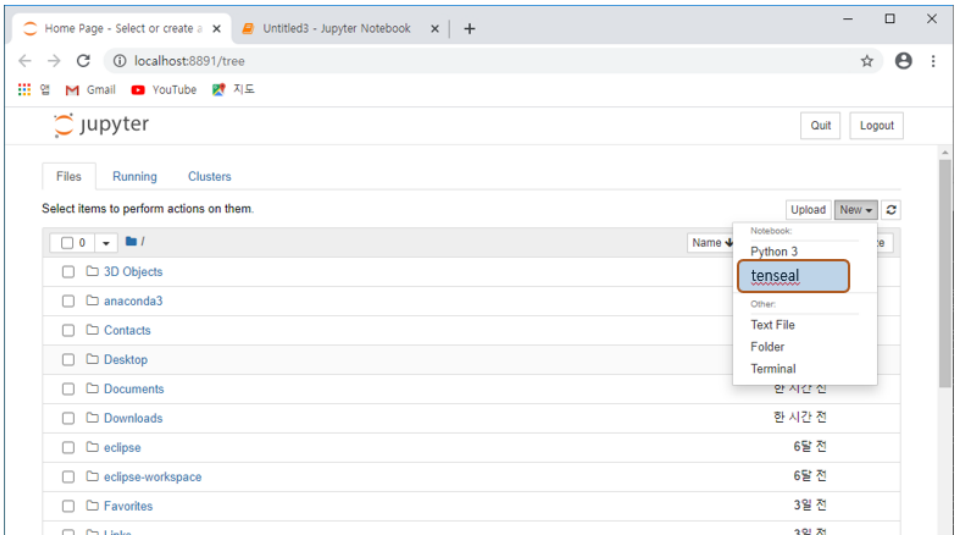
  Run command: >conda deactivate

- To install jupyter notebook in a Conda virtual environment:

  Run commands respectively:

  - (base) > conda activate tenseal
  - (tenseal) > conda install pytorch torchvision -c pytorch
  - (tenseal) > pip install tenseal pytest pandas matplotlib scikit-image
  - (tenseal) > conda install pywin32 –y
  - (tenseal) > pip install jupyter
  - (tenseal) > python -m ipykernel install --user --name=tenseal
  - (tenseal) > jupyter notebook

The screen tab of Jupyter notebook is like this under image. Tenseal 선택



tenseal 선택

4. Jupyter Notebook을 통해 실습 실험

TenSEALContext 는 사용자가 값을 제공해주지 않았을때 사용하는 default scale 로 사용되는 global_scale 도 가지고 있다.

```
# this should throw an error as the global_scale isn't defined yet
try:
    print("global_scale:", context.global_scale)
except ValueError:
    print("The global_scale isn't defined yet")

# you can define it to 2 ** 20 for instance
context.global_scale = 2 ** 20
print("global_scale:", context.global_scale)
```

```
The global_scale isn't defined yet
global_scale: 1048576.0
```

CKKS encryption을 이용한 vector 및 metric 계산

All modules are imported here. Make sure everything is installed by running the cell below.

```
import torch
from torchvision import transforms
from random import randint
import pickle
from PIL import Image
import numpy as np
from matplotlib.pyplot import imshow
from typing import Dict

import tenseal as ts
```

먼저 CKKS TenSEAL context 을 생성해야한다.

In the example, it specifies:

- scheme type: ts.SCHEME_TYPE.CKKS
- poly_modulus_degree: 8192.
- coeff_mod_bit_sizes: The coefficient modulus sizes, here [60, 40, 40, 60]. This means that the coefficient modulus will contain 4 primes of 60 bits, 40 bits, 40 bits, and 60 bits.

```
import tenseal as ts

# Setup TenSEAL context
context = ts.context(
            ts.SCHEME_TYPE.CKKS,
            poly_modulus_degree=8192,
            coeff_mod_bit_sizes=[60, 40, 40, 60]
          )
context
```

```
<tenseal.enc_context.Context at 0x1e423611cc8>
```

- global_scale: the scaling factor, here set to 2^{40}.
- optionally, TenSEAL supports switching between the public key and symmetric key encryption. 기본적으로 우리는 public-key encryption.

```
def context():
    context = ts.context(ts.SCHEME_TYPE.CKKS, 8192, coeff_mod_bit_sizes=[60, 40, 40, 60])
    context.global_scale = pow(2, 40)
    context.generate_galois_keys()
    return context

context = context()
```

PlainTensor class works as a translation layer from common tensor representations to the encrypted forms offered by TenSEAL. 그래서 우리는 plain tensor 를 생성해야한다.

```
plain1 = ts.plain_tensor([1,2,3,4], [2,2])

print(" First tensor: Shape = {} Data = {}".format(plain1.shape, plain1.tolist()))

plain2 = ts.plain_tensor(np.array([5,6,7,8]).reshape(2,2))
print(" Second tensor: Shape = {} Data = {}".format(plain2.shape, plain2.tolist()))
```

새로운 encrypted tensor 를 생성하기 위해, TenSEAL 은 인코딩과 인크립션을 자동으로 시행한다. 이것은 CKKS 와 BFV schemes 둘다 적용된다.

Encrypt 된 tensor 는 PlainTensor 를 encrypt 하고 ciphertexts 와 shapes 를 저장한다.

We have a few variants of encrypted tensors:

- CKKSVector - for 1D float arrays. This version has a smaller memory footprint, but it is less flexible.

```
v1 = [0, 1, 2, 3, 4]
v2 = [4, 3, 2, 1, 0]

# encrypted vectors
enc_v1 = ts.ckks_vector(context, v1)
enc_v2 = ts.ckks_vector(context, v2)
```

```
result = enc_v1 + enc_v2
print(result.decrypt())
```

```
[4.000000000023885, 4.00000000273579, 3.9999999997188076, 3.9999999964564825,
4.000000000356075]
```

```
result = enc_v1.dot(enc_v2)
print(result.decrypt())
```

```
[10.000002197170463]
```

- CKKSTensor - for N-dimensional float arrays. 이 버전은 reshaping 이나 broadcasting 과 같은 encrypted data 에서의 tensorial operation 을 지원한다.

```
encrypted_tensor1 = ts.ckks_tensor(context, plain1)
encrypted_tensor2 = ts.ckks_tensor(context, plain2)

print(" Shape = {}".format(encrypted_tensor1.shape))
print(" Encrypted Data = {}.".format(encrypted_tensor1))


encrypted_tensor_from_np = ts.ckks_tensor(context, np.array([5,6,7,8]).reshape([2,2]))
print(" Shape = {}".format(encrypted_tensor_from_np.shape))
```

```
 Shape = [2, 2]
 Encrypted Data = <tenseal.tensors.ckkstensor.CKKSTensor object at 0x000001E433B54E88>.
 Shape = [2, 2]
```

CKKS tensors variants 에서 지원하는 operation 을 나타낸 표이다.

Operation	Description
negate	Negate an encrypted tensor
square	Compute the square of an encrypted tensor
power	Compute the power of an encrypted tensor
add	Addition between two encrypted tensors
add_plain	Addition between an encrypted tensor and a plain tensor
sub	Subtraction between two encrypted tensors
sub_plain	Subtraction between an encrypted tensor and a plain tensor
mul	Multiplication between two encrypted tensors
mul_plain	Multiplication between an encrypted tensor and a plain tensor
dot	Dot product between two encrypted tensors
dot_plain	Dot product between an encrypted tensor and a plain tensor
polyval	Polynomial evaluation with an encrypted tensor as variable
matmul	Multiplication between an encrypted vector and a plain matrix
matmul_plain	Encrypted matrix multiplication with plain vector

Decrypt Function:

```
def decrypt(enc):
    return enc.decrypt().tolist()
```

Addition of two encrypted tensors:

```
result = encrypted_tensor1 + encrypted_tensor2
print("Plain equivalent: {} + {}\nDecrypted result: {}.".format(plain1.tolist(), plain2.tolist(), decrypt(result)))
```

```
Plain equivalent: [[1.0, 2.0], [3.0, 4.0]] + [[5.0, 6.0], [7.0, 8.0]]
Decrypted result: [[6.000000000337081, 7.999999997494824], [10.000000000222101, 12.00000000016119]].
```

Subtraction of two encrypted tensors:

```
result = encrypted_tensor1 - encrypted_tensor2
print("Plain equivalent: {} - {}\nDecrypted result: {}.".format(plain1.tolist(), plain2.tolist(), decrypt(result)))
```

```
Plain equivalent: [[1.0, 2.0], [3.0, 4.0]] - [[5.0, 6.0], [7.0, 8.0]]
Decrypted result: [[-3.9999999977378096, -3.99999999949312], [-3.999999999091931, -3.999999999987354]].
```

Multiplication of two encrypted tensors:

```
result = encrypted_tensor1 * encrypted_tensor2
print("Plain equivalent: {} * {}\nDecrypted result: {}.".format(plain1.tolist(), plain2.tolist(), decrypt(result)))
```

```
Plain equivalent: [[1.0, 2.0], [3.0, 4.0]] * [[5.0, 6.0], [7.0, 8.0]]
Decrypted result: [[5.000000674377046, 12.000001600067868], [21.000002820456974, 32.00000429328671]].
```

Multiplication with plain tensor:

```
plain = ts.plain_tensor([5,6,7,8], [2,2])
result = encrypted_tensor1 * plain

print("Plain equivalent: {} * {}\nDecrypted result: {}.".format(plain1.tolist(), plain.tolist(), decrypt(result)))
```

```
Plain equivalent: [[1.0, 2.0], [3.0, 4.0]] * [[5.0, 6.0], [7.0, 8.0]]
Decrypted result: [[5.000000679808972, 12.000001604494834], [21.000002820711217, 32.00000429160511]].
```

Negation:

```
result = -encrypted_tensor1

print("Plain equivalent: -{}\nDecrypted result: {}.".format(plain1.tolist(), decrypt(result)))
```

```
Plain equivalent: -[[1.0, 2.0], [3.0, 4.0]]
Decrypted result: [[-1.0000000012996357, -1.999999999000853], [-3.000000000565085, -4.000000000086918]].
```

Power:

```
result = encrypted_tensor1 ** 3
print("Plain equivalent: {} ^ 3\nDecrypted result: {}.".format(plain1.tolist(), decrypt(result)))
```

```
Plain equivalent: [[1.0, 2.0], [3.0, 4.0]] ^ 3
Decrypted result: [[1.000000808301769, 8.000006420369674], [27.000021744549603, 64.0000515070098]].
```

Polynomial evaluation: $1 + X^2 + X^3$

```
result = encrypted_tensor1.polyval([1,0,1,1])

print("X = {}".format(plain1.tolist()))
print("1 + X^2 + X^3 = {}.".format(decrypt(result)))
```

```
X = [[1.0, 2.0], [3.0, 4.0]]
1 + X^2 + X^3 = [[3.0000009414271123, 13.000006953370637], [37.00002295356839, 81.00005365547193]].
```

Sigmoid approximation: $\sigma(x) = 0.5 + 0.197x - 0.004x^3$

```
result = encrypted_tensor1.polyval([0.5, 0.197, 0, -0.004])


print("X = {}".format(plain1.tolist()))
print("0.5 + 0.197 X - 0.004 x^X = {}.".format(decrypt(result)))
```

```
X = [[1.0, 2.0], [3.0, 4.0]]
0.5 + 0.197 X - 0.004 x^X = [[0.6930000201194458, 0.862000032614671], [0.9829999825524341, 1.0319998610369074]].
```

이제, we give you some hints on how to benchmark you homomorphic encryption application, 그리고 가장 적절한 parameters 를 선택한다.

CKKS encryption의 가각 parameter의 역활

Scaling Factor $\triangle = 2^p$

- Computation의 precision를 결정한다.
- The scaling factor determines the precision after the radix point. (Think of this precision in the fixed-point sense)
- CKKS erases 12~25 least significant bits (depending on the multiplicative depth)
    - Each addition and multiplication may consume up to 1 bit
- Hence the value of p for desired precision v is something like $p \fallingdotseq v + 20$
    - It can be adjusted as needed, depending on the multiplicative depth of the computation
    - Just like floating-point arithmetic, approximate homomorphic encryption introduces an error, and errors from prior computations get accumulated.
- In TenSEAL, as in Microsoft SEAL, the degree of the polynomial modulus must be a power of 2 (e.g. 1024, 2048, 4096, 8192, 16384, or 32768).

Ciphertext modulus q

- Functional parameter that determines how many computations are allowed (how much noise can be tolerated)
- Often set implicitly using the value of multiplicative depth specified the user
- Each arithmetic operation increses the noise, and q should be large enough to accommodate the noise from all arithmetic operations
- From the noise perspective, multiplication is much costlier than addition
- The coefficient modulus directly affects:
    - The size of ciphertext elements
    - The length of the list indicates the level of the scheme(or the number of

encrypted multiplications supported).

- The security level (bigger is worse).

Ciphertext dimension N

- Minimum value is computed based on the desired security level and ciphertext modulus q.
- It is also double the size of the vector of encrypted real numbers, i.e., N = 2n
- The polynomial modulus(N in the diagram) directly affects:
  - The number of coefficients in plaintext polynomials
  - The size of ciphertext elements
  - The computational performance of the scheme (bigger is worse)
  - The security level (bigger is better).

Context serialization and Security Key

The TenSEAL context is required for defining the perfomance and security of your application. In a client-server setup, it is required only on the initial handshake, as the ciphertexts can be linked with an existing context on deserialization.

- Experiment of the size of the context, depending on the input parameters.
- 어떤 security key를 이용하느냐에 따라 context serialization 크기가 달라지는 것을 확인 할 수 이쇼다.
  - public_key
  - secret_key
  - galois_keys
  - relin_keys

이 실험에서 ckks homomorphic parameters 는 (8192, [40, 20, 40])으로 진행.

```
import tenseal as ts
import tenseal.sealapi as seal
import random
import pickle
import numpy as np
import struct
import sys
import math
import array
from IPython.display import HTML, display
import tabulate
import pytest
```

```
def convert_size(size_bytes):
    if size_bytes == 0:
        return "0B"
    size_name = ("B", "KB", "MB", "GB", "TB", "PB", "EB", "ZB", "YB")
    i = int(math.floor(math.log(size_bytes, 1024)))
    p = math.pow(1024, i)
    s = round(size_bytes / p, 2)
    return "%s %s" % (s, size_name[i])

# Type of Encryption
enc_type_str = {
    ts.ENCRYPTION_TYPE.SYMMETRIC : "symmetric",
    ts.ENCRYPTION_TYPE.ASYMMETRIC : "asymmetric",
}

# Select Ecryption Scheme ( CKKS or BFV )
scheme_str = {
    ts.SCHEME_TYPE.CKKS : "ckks",
    ts.SCHEME_TYPE.BFV : "bfv",
}

def decrypt(enc):
    return enc.decrypt()
```

```
ctx_size_benchmarks = [["Encryption Type",
                        "Scheme Type",
                        "Polynomial modulus",
                        "Coefficient modulus sizes",
                        "Saved keys",
                        "Context serialized size", ]]
```

```
for enc_type in [ts.ENCRYPTION_TYPE.SYMMETRIC, ts.ENCRYPTION_TYPE.ASYMMETRIC]:
    for (poly_mod, coeff_mod_bit_sizes) in [
        (8192, [40, 20, 40]),
    ]:
        context = ts.context(
            scheme=ts.SCHEME_TYPE.CKKS,
            poly_modulus_degree=poly_mod,
            coeff_mod_bit_sizes=coeff_mod_bit_sizes,
            encryption_type=enc_type,
        )
        context.generate_galois_keys()
        context.generate_relin_keys()
```

```
        ser = context.serialize(save_public_key=True,
                                save_secret_key=True,
                                save_galois_keys=True,
                                save_relin_keys=True)
        ctx_size_benchmarks.append([enc_type_str[enc_type],
                                    scheme_str[ts.SCHEME_TYPE.CKKS],
                                    poly_mod,
                                    coeff_mod_bit_sizes,
                                    "all",
                                    convert_size(len(ser))])

        if enc_type is ts.ENCRYPTION_TYPE.ASYMMETRIC:
            ser = context.serialize(save_public_key=True,
                                    save_secret_key=False,
                                    save_galois_keys=False,
                                    save_relin_keys=False)
            ctx_size_benchmarks.append([enc_type_str[enc_type],
                                        scheme_str[ts.SCHEME_TYPE.CKKS],
                                        poly_mod,
                                        coeff_mod_bit_sizes,
                                        "Public key",
                                        convert_size(len(ser))])

        ser = context.serialize(save_public_key=False,
                                save_secret_key=True,
                                save_galois_keys=False,
                                save_relin_keys=False)
        ctx_size_benchmarks.append([enc_type_str[enc_type],
                                    scheme_str[ts.SCHEME_TYPE.CKKS],
                                    poly_mod,
                                    coeff_mod_bit_sizes,
                                    "Secret key",
                                    convert_size(len(ser))])

        ser = context.serialize(save_public_key=False,
                                save_secret_key=False,
                                save_galois_keys=True,
                                save_relin_keys=False)
        ctx_size_benchmarks.append([enc_type_str[enc_type],
                                    scheme_str[ts.SCHEME_TYPE.CKKS],
                                    poly_mod,
                                    coeff_mod_bit_sizes,
                                    "Galois keys",
                                    convert_size(len(ser))])

        ser = context.serialize(save_public_key=False,
                                save_secret_key=False,
                                save_galois_keys=False,
                                save_relin_keys=True)
        ctx_size_benchmarks.append([enc_type_str[enc_type],
                                    scheme_str[ts.SCHEME_TYPE.CKKS],
                                    poly_mod,
                                    coeff_mod_bit_sizes,
                                    "Relin keys",
                                    convert_size(len(ser))])
```

```
        ser = context.serialize(save_public_key=False,
                                save_secret_key=False,
                                save_galois_keys=False,
                                save_relin_keys=False)
        ctx_size_benchmarks.append([enc_type_str[enc_type],
                                    scheme_str[ts.SCHEME_TYPE.CKKS],
                                    poly_mod,
                                    coeff_mod_bit_sizes,
                                    "none",
                                    convert_size(len(ser))])

display(HTML(tabulate.tabulate(ctx_size_benchmarks, tablefmt='html')))
```

Encryption Type	Scheme Type	Polynomial modulus	Coefficient modulus sizes	Saved keys	Context serialized size
symmetric	ckks	8192	[40, 20, 40]	all	124.88 KB
symmetric	ckks	8192	[40, 20, 40]	Secret key	124.88 KB
symmetric	ckks	8192	[40, 20, 40]	Galois keys	11.89 MB
symmetric	ckks	8192	[40, 20, 40]	Relin keys	503.56 KB
symmetric	ckks	8192	[40, 20, 40]	none	91.0 B
asymmetric	ckks	8192	[40, 20, 40]	all	376.15 KB
asymmetric	ckks	8192	[40, 20, 40]	Public key	251.43 KB
asymmetric	ckks	8192	[40, 20, 40]	Secret key	124.81 KB
asymmetric	ckks	8192	[40, 20, 40]	Galois keys	11.89 MB
asymmetric	ckks	8192	[40, 20, 40]	Relin keys	503.64 KB
asymmetric	ckks	8192	[40, 20, 40]	none	89.0 B

- The symmetric encryption creates smaller contexts than the public key ones.
- Decreasing the length of the coefficient modulus decreases the size of the context but also the depth of available multiplications.
- Decreasing the coefficient modulus sizes reduces the context size, but impacts the precision as well (for CKKS).
- Galois keys increase the context size only for public contexts (without the secret key). Send them only when you need to perform ciphertext rotations on the other end.
- Re-linearization keys increase the context size only for public contexts. Send them only when you need to perform multiplications on ciphertexts on the other end.
- When we send the secret key, the Re-linearization/Galois key can be re-generated on the other end without sending them.

Ciphertext serialization

We next review how different parameters impact the ciphertext serialization. The first observation here is that the symmetric/asymmetric

encryption switch doesn't actually impact the size of the ciphertext, only of the context. We will review the benchmarks only for the asymmetric scenario.

```
data = [random.uniform(-10, 10) for _ in range(10 ** 3)]
network_data = pickle.dumps(data)
print("Plain data size in bytes {}".format(convert_size(len(network_data))))
```

```
Plain data size in bytes 8.8 KB
```

```
def display_param(test_param):
    enc_type = ts.ENCRYPTION_TYPE.ASYMMETRIC
    ct_size_benchmarks_ckks = [["Encryption Type",
                        "Scheme Type",
                        "Polynomial modulus",
                        "Coefficient modulus sizes",
                        "Precision",
                        "Ciphertext serialized size",
                        "Encryption increase ratio"
                       ]]

    for (poly_mod, coeff_mod_bit_sizes, prec) in test_parameters_ckks:
        context = ts.context(
            scheme=ts.SCHEME_TYPE.CKKS,
            poly_modulus_degree=poly_mod,
            coeff_mod_bit_sizes=coeff_mod_bit_sizes,
            encryption_type=enc_type,
        )
        scale = 2 ** prec
        ckks_vec = ts.ckks_vector(context, data, scale)

        enc_network_data = ckks_vec.serialize()
        ct_size_benchmarks_ckks.append([enc_type_str[enc_type],
                                        scheme_str[ts.SCHEME_TYPE.CKKS],
                                        poly_mod, coeff_mod_bit_sizes,
                                        "2**{}".format(prec),
                                        convert_size(len(enc_network_data)),
                                        round(len(enc_network_data) / len(network_data), 2)
                                       ])
    display(HTML(tabulate.tabulate(ct_size_benchmarks_ckks, tablefmt='html')))
```

```
test_parameters_ckks = [
    (2**13, [60, 40, 60], 40),
    (2**13, [60, 40, 60], 20),
    (2**13, [60, 40, 60], 10),
    (2**13, [60, 40, 60], 5),
    (2**13, [40, 20, 40], 40),
    (2**13, [40, 20, 40], 20),
    (2**13, [40, 20, 40], 10),
    (2**13, [40, 20, 40], 5),
    (2**13, [20, 20, 20], 20),
    (2**13, [20, 20, 20], 10),
    (2**13, [20, 20, 20], 5)
]
display_param(test_parameters_ckks)
```

Encryption Type	Scheme Type	Polynomial modulus	Coefficient modulus sizes	Precision	Ciphertext serialized size	Encryption increase ratio
asymmetric	ckks	8192	[60, 40, 60]	2**40	229.77 KB	26.12
asymmetric	ckks	8192	[60, 40, 60]	2**20	229.81 KB	26.12
asymmetric	ckks	8192	[60, 40, 60]	2**10	229.82 KB	26.13
asymmetric	ckks	8192	[60, 40, 60]	2**5	229.83 KB	26.13
asymmetric	ckks	8192	[40, 20, 40]	2**40	153.35 KB	17.43
asymmetric	ckks	8192	[40, 20, 40]	2**20	153.18 KB	17.41
asymmetric	ckks	8192	[40, 20, 40]	2**10	154.49 KB	17.56
asymmetric	ckks	8192	[40, 20, 40]	2**5	153.22 KB	17.42
asymmetric	ckks	8192	[20, 20, 20]	2**20	104.78 KB	11.91
asymmetric	ckks	8192	[20, 20, 20]	2**10	104.58 KB	11.89
asymmetric	ckks	8192	[20, 20, 20]	2**5	104.63 KB	11.89

- The polynomial modulus increase results in a ciphertext increase.
- The length of coefficient modulus sizes impacts the ciphertext size.
- The values of the coefficient modulus sizes impact the ciphertext size, as well as the precision.
- For a fixed set of polynomial modulus and coefficient modulus sizes, changing the precision doesn't affect the ciphertext size.

Activation and Pooling in Homomorphic Encryption

표준 심층 신경망과의 주요 편차는 최대 함수가 HE 친화적이지 않기 때문에 MaxPooling 이 아닌 AveragePooling 을 사용하고. ReLu 또는 Sigmoid 와 같은 비다 항 함수는 마찬가지로 HE 친화적이지 않기 때문에 또 다른 변경 사항은 제곱 활성화 함수를 사용하는 것이다.

```
import torch
from torchvision import datasets
import torchvision.transforms as transforms
import numpy as np

torch.manual_seed(73)

train_data = datasets.MNIST('data', train=True, download=True, transform=transforms.ToTensor())
test_data = datasets.MNIST('data', train=False, download=True, transform=transforms.ToTensor())

batch_size = 64

train_loader = torch.utils.data.DataLoader(train_data, batch_size=batch_size, shuffle=True)
test_loader = torch.utils.data.DataLoader(test_data, batch_size=batch_size, shuffle=True)
```

```
class ConvNet(torch.nn.Module):
    def __init__(self, hidden=64, output=10):
        super(ConvNet, self).__init__()
        self.conv1 = torch.nn.Conv2d(1, 4, kernel_size=5, padding=0, stride=2)
        self.fc1 = torch.nn.Linear(144, hidden)
        self.fc2 = torch.nn.Linear(hidden, output)

    def forward(self, x):
        x = torch.nn.functional.relu(torch.nn.functional.max_pool2d(self.conv1(x), kernel_size=2, stride=2))

        # flattening while keeping the batch axis
        x = x.view(-1, 144)
        x = torch.nn.functional.relu(self.fc1(x))
        x = self.fc2(x)
        return x
```

```
def train(model, train_loader, criterion, optimizer, n_epochs=10):
    # model in training mode
    model.train()
    for epoch in range(1, n_epochs+1):

        train_loss = 0.0
        for data, target in train_loader:
            optimizer.zero_grad()
            output = model(data)
            loss = criterion(output, target)
            loss.backward()
            optimizer.step()
            train_loss += loss.item()

        # calculate average losses
        train_loss = train_loss / len(train_loader)

        print('Epoch: {} \tTraining Loss: {:.6f}'.format(epoch, train_loss))

    # model in evaluation mode
    model.eval()
    return model
```

```
model = ConvNet()
criterion = torch.nn.CrossEntropyLoss()
optimizer = torch.optim.Adam(model.parameters(), lr=0.001)
model = train(model, train_loader, criterion, optimizer, 10)
```

```
Epoch: 1        Training Loss: 0.516468
Epoch: 2        Training Loss: 0.220972
Epoch: 3        Training Loss: 0.158488
Epoch: 4        Training Loss: 0.126927
Epoch: 5        Training Loss: 0.105530
Epoch: 6        Training Loss: 0.092391
Epoch: 7        Training Loss: 0.083163
Epoch: 8        Training Loss: 0.074145
Epoch: 9        Training Loss: 0.068467
Epoch: 10       Training Loss: 0.064212
```

```
def test(model, test_loader, criterion):
    # initialize lists to monitor test loss and accuracy
    test_loss = 0.0
    class_correct = list(0. for i in range(10))
    class_total = list(0. for i in range(10))

    # model in evaluation mode
    model.eval()
```

```
    for data, target in test_loader:
        output = model(data)
        loss = criterion(output, target)
        test_loss += loss.item()
        # convert output probabilities to predicted class
        _, pred = torch.max(output, 1)
        # compare predictions to true label
        correct = np.squeeze(pred.eq(target.data.view_as(pred)))
        # calculate test accuracy for each object class
        for i in range(len(target)):
            label = target.data[i]
            class_correct[label] += correct[i].item()
            class_total[label] += 1

    # calculate and print avg test loss
    test_loss = test_loss/len(test_loader)
    print(f'Test Loss: {test_loss:.6f}\n')

    for label in range(10):
        print(
            f'Test Accuracy of {label}: {int(100 * class_correct[label] / class_total[label])}% '
            f'({int(np.sum(class_correct[label]))}/{int(np.sum(class_total[label]))})'
        )

    print(
        f'\nTest Accuracy (Overall): {int(100 * np.sum(class_correct) / np.sum(class_total))}% '
        f'({int(np.sum(class_correct))}/{int(np.sum(class_total))})'
    )
```

```
test(model, test_loader, criterion)
```

```
Test Loss: 0.066041

Test Accuracy of 0: 99% (971/980)
Test Accuracy of 1: 99% (1124/1135)
Test Accuracy of 2: 98% (1016/1032)
Test Accuracy of 3: 97% (988/1010)
Test Accuracy of 4: 98% (967/982)
Test Accuracy of 5: 98% (877/892)
Test Accuracy of 6: 97% (938/958)
Test Accuracy of 7: 98% (1008/1028)
Test Accuracy of 8: 97% (948/974)
Test Accuracy of 9: 95% (968/1009)

Test Accuracy (Overall): 98% (9805/10000)
```

위에 실험을 MaxPooling 이 아닌 AveragePooling 을 사용하고, ReLu Activation 아닌 Square Activation 을 사용할 경우 아래와 같이 ConvNet 를 바꿀 수 있다.

```
class ConvNet(torch.nn.Module):
    def __init__(self, hidden=64, output=10):
        super(ConvNet, self).__init__()
        self.conv1 = torch.nn.Conv2d(1, 4, kernel_size=5, padding=0, stride=2)
        self.fc1 = torch.nn.Linear(144, hidden)
        self.fc2 = torch.nn.Linear(hidden, output)

    def forward(self, x):
        x = torch.nn.functional.avg_pool2d(self.conv1(x), kernel_size=2, stride=2)
        # the model uses the square activation function
        x = x * x
        # flattening while keeping the batch axis
        x = x.view(-1, 144)
        x = self.fc1(x)
        x = x * x
        x = self.fc2(x)
        return x
```

5. 실습 실험 후 보고서에 포함될 내용

a. CKKS를 통해 encrypted_tensor1의 Polynomial evaluation $3x^3 - 34x^2 + 18x - 145$ encryption 하여 decryption list를 작성하시오

b. Polynomial modulus를 8192와 Coefficient modulus sizes를 [40, 21, 21, 21, 21, 21, 21, 40]으로 사용시 security key를 이용하느냐에 따라 context serialization 크기가 달라지는 결과 작성하시오

c. Ciphertext serialization 실험에서 아래와 같은 test_parameters를 사용시, Ciphertext serialized size와 Encryption increase ratio가 어떻게 나오는지 작성하시오

(2**13, [60, 60], 20), (2**13, [60, 60], 10), (2**13, [60, 60], 5), (2**13, [40, 40], 38), (2**13, [40, 40], 20), (2**13, [40, 40], 10), (2**13, [40, 40], 5), (2**13, [20, 20], 18), (2**13, [20, 20], 10), (2**13, [20, 20], 5),

(2**12, [40, 20, 40], 40), (2**12, [40, 20, 40], 20), (2**12, [40, 20, 40], 10), (2**12, [40, 20, 40], 5), (2**12, [30, 20, 30], 40), (2**12, [30, 20, 30], 10), (2**12, [30, 20, 30], 5), (2**12, [20, 20, 20], 20), (2**12, [20, 20, 20], 10), (2**12, [20, 20, 20], 5)

d. Homomorphic Encryption에서 Pooling function은 MaxPooling이 아닌 AveragePooling을 사용하고. Activation function은 ReLu 아닌 Square Activation을 사용한다. 마지막 실험에서 Average Pooling와 Square Activation를 사용시 Test Accuracy를 출력하고, 그래프를 통해 MaxPooling/ReLu model Test Accuracy와 AveragePooling/Square model Test Accuracy를 비교하시오

제출 기한 : 2021.11.24(수) 오후 11:59

제출 양식 : 이름_학번_보고서7.pdf (ex. 홍길동_2021-12345_보고서7.pdf)

*보고서는 pdf로 변환하여 제출