실험 8 Encrypted Convolution on MNIST with CKKS Homomorphic

작성자: 위두랑가

실습 조교: 박천명, 이염미

1. 실험 목적

PC에서 MNIST model을 이용하여 CKKS homomorphic encryption을 수행한다. MNIST 데이터셋을 다룰때는 컨볼루션 레이어로 만들어진 간단한 신경망을 사용할 수 있다. 여기서 CKKS scheme에 대한 곱셈 수의 제한이 주어졌을때 단순성을 위해 square activation function을 사용한다.

2. 실험 전에 준비해야할 내용

a) Machine Learning Model

모델은 아래 layer들의 순서이다:

- Convolution: Convolution with 4 kernels. Shape of the kernel is 7x7. Strides are 3x3.
- Activation: Square activation function.
- Linear Layer 1: Input size: 256. Output size: 64.
- Activation: Square activation function.
- Linear Layer 2: Input size: 64. Output size: 10.

메모리와 연산을 가장 작게 유지하기 위해 우리는 주로 single cipher text 를 사용한다. 이 모델에는 두가지 다른 표현이 있다. 하나는 컨볼루션용이고 다른 하나는 선형 레이어용이다. 선형 레이어의 input vector 는 cipher texts 의 slot 에 맞기 위해 여러번 복제된 것이다. 그래서 하나의 cipher text 는 선형층의 모든 input 을 가지고 있다.

컨볼루션:

컨볼루션에서는 몇 가지 방법을 사용할 수 있다. 그 중 하나는 그림 7.1 처럼 2D 컨볼루션을 single matrix 곱셈 연산으로 변환하는 알고리즘으로 잘 알려져있는 것이다.

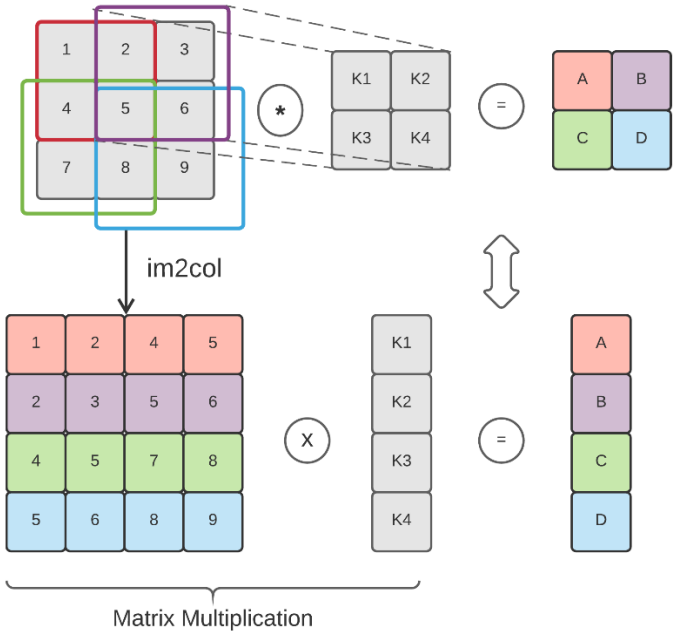


Figure 1. Image to column convolution.

그러나 이것은 input matrix의 구성을 특별하게 재배치해야한다. 그리고 ciphertext에서는 쉽게 할 수 없기 때문에 우리는 encryption 이전 단계에서 진행해야한다. Convolution을 수행하기 위해 우리는 먼저 input matrix에

im2col encoding을 하고 이것을 single ciphertext로 encrypt한다. Matrix가 vertical scan을 이용해 vector로 변환된 것은 아무런 의미가 없다. 이제 encrypted matrix와 plain vector끼리 행렬 곱셈을 시행한다. 이것은 먼저 kernel의 모든 요소를 n번 복제하여 새로운 flattened kernel을 만든다. 여기서 n은 window의 개수이다. 우리는 ciphertext-plaintext 곱셈을 시행한다. 이 과정은 Figure 2, Figure 3에 묘사되어 있다.

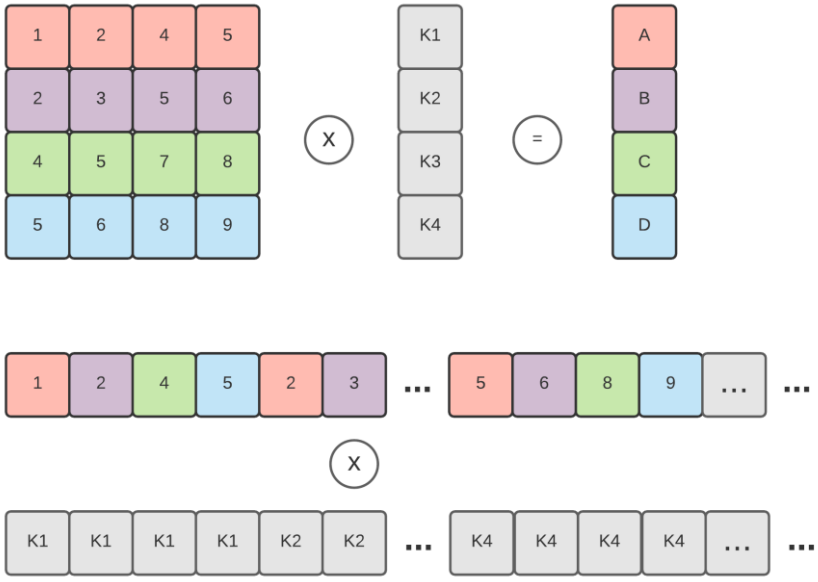


Figure 2. Image to column convolution with CKKS - step 1.

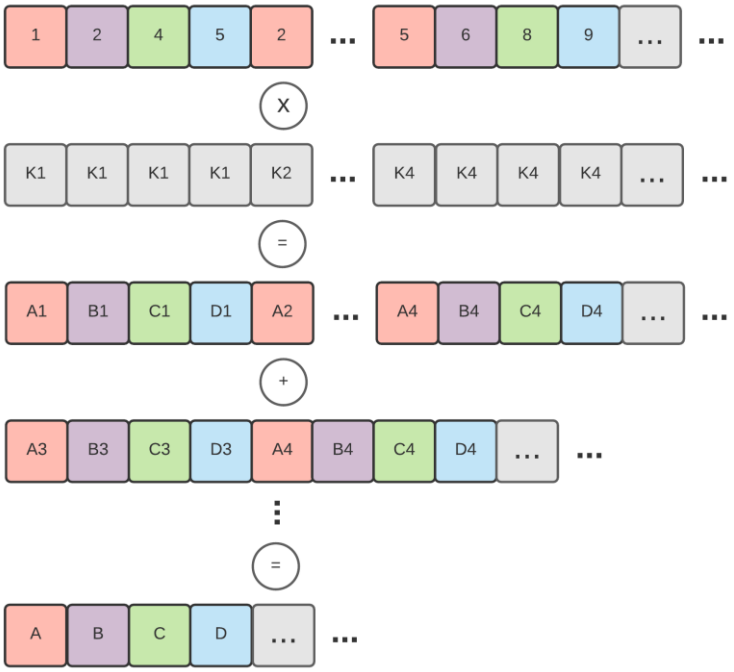


Figure 3. Image to column convolution with CKKS - step 2.

여러개의 kernel이 사용되었다면 우리는 다른 ciphertext 결과를 내는 이 작업을 여러번 수행해야한다. 이 ciphertext는 나중에 flattened vector로 결합된다. 그래서 모든 convolution은 64개의 유용한 slot을 가진 ciphertext를 도출한다. 그리고 4 kernel의 결과를 결합하여 256개의 유용한 slot이되고 이것이 첫번째 선형 레이어의 input이 된다.

선형 레이어:

선형 레이어는 bias의 vector-matrix 곱셈과 덧셈으로 요약된다. Matrix와 bias는 encrypt되지 않았다. Vector-matrix 곱셈은 Halevi와 Shoup의 diagonal method를 기반으로 시행되었다. 이것은 약간의 다른 rotation을 가지고 여러개의 ciphertext-plaintext 곱셈들이 누적된 것이다. 우리는 plain matrix에서 모든 diagonal을 반복하고 여기에 왼쪽으로 n slot만큼 rotate한 ciphertext을 곱한다. 여기서 n은 index of the diagonal이다.

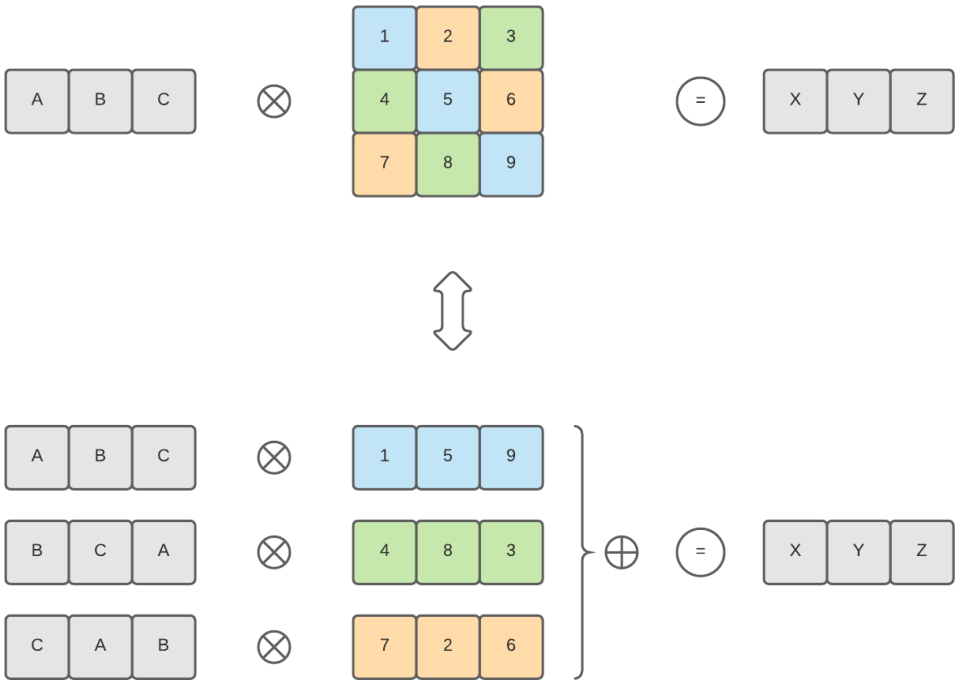


Figure 4. Vector-Matrix Multiplication.

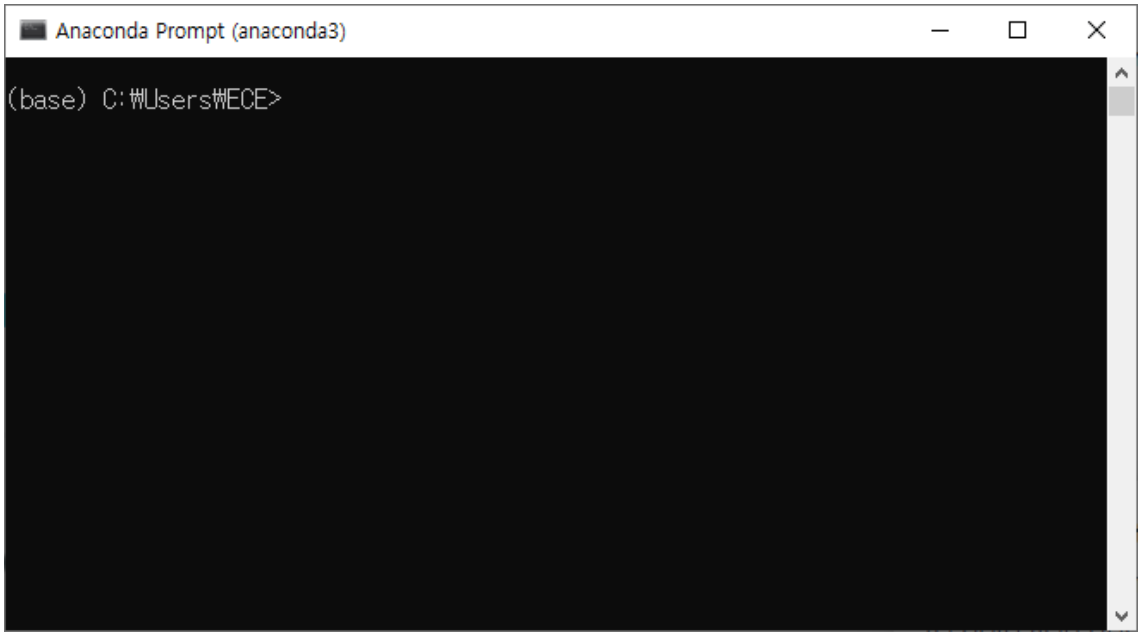
Square Activation:

Square activation은 꽤 간단하다. 우리는 그냥 ciphertext를 자기자신과 곱한다.

여기서 우리는 정확히 6번의 곱셈을 한다. Convolution에서 2번, first square activation에서 1번, first linear layer에서 1번, second square activation에서 1번, last linear layer에서 1번이다.

3. 실습 실험을 진행하기 위한 환경 세팅

실험 7에서 생성한 Anaconda virtual environment (tenseal 환경)에서 이번 실험을 진행한다. Anaconda Prompt 프로그램을 열어 conda activate tenseal 명령어를 통해 tenseal 환경을 활성화할 수 있다.

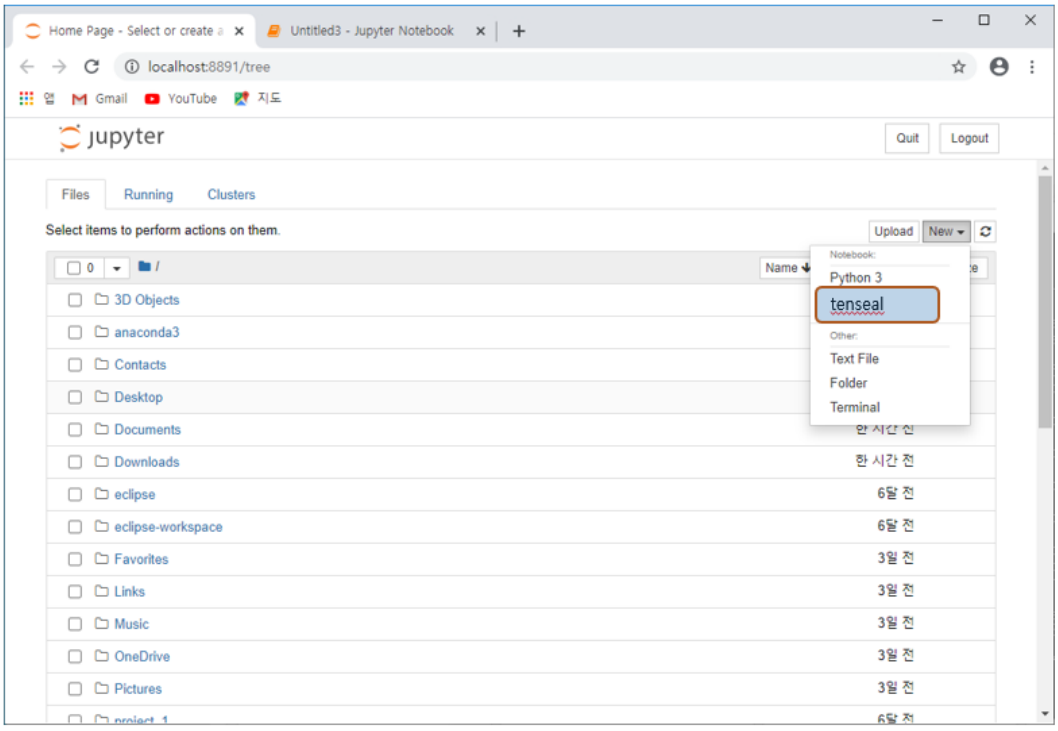


- enviroment 활성화:

  (base) > conda activate tenseal

  (tenseal) >

The screen tab of Jupiter notebook is like this under image. tenseal 선택



tenseal 선택

4. Jupyter Notebook을 통해 실습 실험

Homomorphic encryption을 통해 neural network 모델을 다루는 방법에 대해 알아본다. 이번 실험에서는 지난 실험에서 배운 모든 operation을 다룰 수 있는 TenSEAL library를 이용할 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=7, padding=0, stride=3)
        self.fc1 = torch.nn.Linear(256, hidden)
        self.fc2 = torch.nn.Linear(hidden, output)

    def forward(self, x):
        x = self.conv1(x)
        # the model uses the square activation function
        x = x * x
        # flattening while keeping the batch axis
        x = x.view(-1, 256)
        x = self.fc1(x)
        x = x * 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)
```

screen에 결과값을 얻는다:

```
Epoch: 1        Training Loss: 0.392145
Epoch: 2        Training Loss: 0.131439
Epoch: 3        Training Loss: 0.090824
Epoch: 4        Training Loss: 0.070182
Epoch: 5        Training Loss: 0.059312
Epoch: 6        Training Loss: 0.049881
Epoch: 7        Training Loss: 0.045489
Epoch: 8        Training Loss: 0.038426
Epoch: 9        Training Loss: 0.035883
Epoch: 10       Training Loss: 0.031704
```

test set으로 정확도를 확인한다:

```
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)
```

결과를 얻는다:

```
Test Loss: 0.099073

Test Accuracy of 0: 99% (971/980)
Test Accuracy of 1: 99% (1130/1135)
Test Accuracy of 2: 97% (1005/1032)
Test Accuracy of 3: 98% (995/1010)
Test Accuracy of 4: 97% (960/982)
Test Accuracy of 5: 97% (869/892)
Test Accuracy of 6: 97% (938/958)
Test Accuracy of 7: 96% (994/1028)
Test Accuracy of 8: 96% (937/974)
Test Accuracy of 9: 96% (978/1009)

Test Accuracy (Overall): 97% (9777/10000)
```

Now start the encrypted evaluation that will use the pre-trained model:

```
import tenseal as ts


class EncConvNet:
    def __init__(self, torch_nn):
        self.conv1_weight = torch_nn.conv1.weight.data.view(
            torch_nn.conv1.out_channels, torch_nn.conv1.kernel_size[0],
            torch_nn.conv1.kernel_size[1]
        ).tolist()
        self.conv1_bias = torch_nn.conv1.bias.data.tolist()

        self.fc1_weight = torch_nn.fc1.weight.T.data.tolist()
        self.fc1_bias = torch_nn.fc1.bias.data.tolist()

        self.fc2_weight = torch_nn.fc2.weight.T.data.tolist()
        self.fc2_bias = torch_nn.fc2.bias.data.tolist()


    def forward(self, enc_x, windows_nb):
        # conv layer
        enc_channels = []
        for kernel, bias in zip(self.conv1_weight, self.conv1_bias):
            y = enc_x.conv2d_im2col(kernel, windows_nb) + bias
            enc_channels.append(y)
        # pack all channels into a single flattened vector
        enc_x = ts.CKKSVector.pack_vectors(enc_channels)
        # square activation
        enc_x.square_()
        # fc1 layer
        enc_x = enc_x.mm(self.fc1_weight) + self.fc1_bias
        # square activation
        enc_x.square_()
        # fc2 layer
        enc_x = enc_x.mm(self.fc2_weight) + self.fc2_bias
        return enc_x

    def __call__(self, *args, **kwargs):
        return self.forward(*args, **kwargs)
```

```python
def enc_test(context, model, test_loader, criterion, kernel_shape, stride):
    # 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))

    for data, target in test_loader:
        # Encoding and encryption
        x_enc, windows_nb = ts.im2col_encoding(
            context, data.view(28, 28).tolist(), kernel_shape[0],
            kernel_shape[1], stride
        )
        # Encrypted evaluation
        enc_output = enc_model(x_enc, windows_nb)
        # Decryption of result
        output = enc_output.decrypt()
        output = torch.tensor(output).view(1, -1)

        # compute loss
        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
        label = target.data[0]
        class_correct[label] += correct.item()
        class_total[label] += 1


    # calculate and print avg test loss
    test_loss = test_loss / sum(class_total)
    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))})'
    )
```

Encryption 계산 시간이 오래 걸릴 수 있기 때문에 MNIST test_dataset에서 500개의 data의 대해서만 inference해 본다. (10000의 dataset를 선택할 경우 5~6 시간 걸릴 수 있다)

```python
large_testset, small_testset = torch.utils.data.random_split(test_data, [9500, 500])
```

```python
# Load one element at a time
test_loader = torch.utils.data.DataLoader(small_testset, batch_size=1, shuffle=True)
# required for test_data
kernel_shape = model.conv1.kernel_size
stride = model.conv1.stride[0]
```

```
## Encryption Parameters

# controls precision of the fractional part
bits_scale = 26

# Create TenSEAL context
context = ts.context(
    ts.SCHEME_TYPE.CKKS,
    poly_modulus_degree=8192,
    coeff_mod_bit_sizes=[31,
                         bits_scale,
                         bits_scale,
                         bits_scale,
                         bits_scale,
                         bits_scale,
                         bits_scale,
                         31]
)

# set the scale
context.global_scale = pow(2, bits_scale)

# galois keys are required to do ciphertext rotations
context.generate_galois_keys()
```

이러한 매개변수를 선택한 이유:

- 주어진 보안레벨(e.g. 128-bits security)과 polynomial modulus degree (e.g. 8192)에서 효율적 modulus(*sum(coeff_mod_bit_sizes)*)의 bit count 에 상한선이 있다. 만약 상한선을 넘었다면 필요한 보안 레벨을 맞추기 위해 더 높은 polynomial modulus degree 를 사용해야 한다.
- Multiplicative 깊이는 coefficient modulus 를 조정하는 prime 의 숫자로 조절된다.
- rescaling ciphertexts 를 케어하기 때문에 *coeff_mod_bit_sizes[1: -1]* 의 모든 구성요소는 TenSEAL 에서 동일해야 한다. 또한 encryption 동안 같은 숫자의 bits(e.g. 2 ^ 26)를 스케일로 사용해야한다.
- integer coefficients 의 다항식으로 인코딩되기 전 plaintexts 가 곱해진 값이므로 이 스케일은 분수 파트의 예측을 컨트롤한다.

20비트 이상의 스케일에서 시작하여 우리는 이것들의 middle prime까지

선택해야하기 때문에 이것은 벌써 120비트를 넘긴다. 이 coefficient modulus의 하한값과 128비트의 보안레벨을 가지고 우리는 최소 8192의 polynomial modulus degree가 필요하다. Higher degree를 선택하는 상한값은 218이다. 손실과 정확도를 확인하며 Coefficient modulus를 바르게 예측하기 위해 다른 값들을 시도해보면 우리는 26bit의 scale과 prime에서 끝난다. 또한 마지막 coefficient modulus에서 정수부분을 위해 5비트가 필요하다.

You have done encrypted inference on a test-set of random 500 elements.

```
import time
t1 = time.time()

enc_model = EncConvNet(model)
enc_test(context,
         enc_model,
         test_loader,
         criterion,
         kernel_shape,
         stride)

t2 = time.time()
```

```
def sec_to_hours(seconds):
    a=str(seconds//3600)
    b=str((seconds%3600)//60)
    c=str((seconds%3600)%60)
    d=["{} hours {} mins {} seconds".format(a, b, c)]
    return d
sec_to_hours(t2 -t1)
```

And get the output like this:

5. 실험 후 보고서에 포함될 내용

i. 일반적인 방식으로 계산한 MNIST model 테스트 결과와 homomorphic encryption을 통해서 계산한 결과를 확인하여 차이가 무엇인지 작성하시오.

ii. 테스트 데이터를 homomorphic encryption을 통해서 계산하기 위해 걸리는 시간을 작성하시오.

iii. 이 실험에서 MNIST test_dataset를 large_testset와 small_testset으로 나눠서 small_testset를 homomorphic encryption을 통해서 계산한 결과를 확인해 봤습니다. large_testset를 homomorphic encryption을 통해서 계산 시 결과와 걸리는 시간이 작성하시오. (과제)