USING MACHINE LEARNING ON ENCRYPTED DATA

Catalin Emanuel CIOBOTA [1]

Abstract: *This post discusses cutting-edge cryptography techniques. Do not use examples in this blog post for production applications. Always consult a professional cryptographer before using cryptography.*

In the following lines we will discuss data encryption techniques using advanced encryption techniques. The examples presented are not recommended for use in production applications. Before applying encryption to data applied to the services of a professional.

Keywords: machine learning, cryptography.

Introduction

The present research applies to new machine learning applications developed for example with Flux.js and which are intended to be applied and implemented in other applications or users. How can we do that? There are several types and the most common would be the implementation of API functions or we send the model to users and advise them to launch it locally using their data. There are a number of problems using this technique:

1. Machine learning models are large and users may not have as much storage space or may run learning models.

2. Models change frequently and we do not want to send the new model to every change.

3. New models are difficult to generate, both human resources and hardware resources are needed that the development company wants to recover by billing these services to users.

The accepted solution is to declare the API function on the cloud. This type of machine learning business offered as a service is in vogue now. Many types of such products have appeared in recent years, and can be found on cloud platforms. The problem that arises is: is the data of those who use such services secure? Remember the user sends the data to be processed on a remote server. Is the server reliable? There are ethical and legal issues that govern how this type of data can be used. For example, in finance or even in medical services, sending data to third parties cannot be done. How can we solve this type of problem?

[1] Drd,, Valahia University, Targoviste, ciobota_catalin@yahoo.com

New research in this area may make it possible to perform calculations using data without decryption. We will make an example in which someone sends data (images) to an API function in the cloud, which will execute the machine learning model and receive the encrypted response. The user data was not decrypted, the one who manages the server does not have access to the data, these being encrypted, nor can the prediction mode be calculated. How will we do this? We will use a machine learning model to recognize letters and numbers from handwriting (the MNIST set will be used).

HE generally

The calculation used to calculate encrypted data is called "secure calculation" and is widely used in research, with many applications and techniques for various scenarios. In this article we will use the "homomorphic encryption" technique. In the applications that use this technique the following operations are used:

```
•  pub_key, eval_key, priv_key = keygen()
•  encrypted = encrypt(pub_key, plaintext)
•  decrypted = decrypt(priv_key, encrypted)
•  encrypted' = eval(eval_key, f, encrypted)
```

The first functions are simple functions, common to those who use asymmetric cryptography. The last function is the important one. It processes the f functions on the encryption side and displays another encrypted value depending on the function evaluation. The property itself is what provides the name of the homomorphic calculus. The evaluation function is displayed according to encryption:

f (decrypt (priv_key, encrypted)) == decrypt (priv_key, eval (eval_key, f, encrypted)).

The functions f are accepted in the calculation depending on the encryption and operations. For a single f a "partially homomorphic" scheme is used. If the function f is used as a series of arbitrary circuits, the calculation will be called "somewhat homomorphic" or "completely homomorphic" if such a circuit is unlimited. It is even possible to transform certain data through a fully "homomorphic" technique and such a technique is called "bootstrapping". The completely homomorphic encryption technique is recent and was published by Craig Gentry[2] in 2009. There are also commercial solutions that implement this technique. Example: Microsoft Seal[3] and Palisade[4]. We will use CKKS encryption in examples.

[2] https://www.cs.cmu.edu/~odonnell/hits09/gentry-homomorphic-encryption.pdf
[3] https://github.com/microsoft/SEAL
[4] https://palisade-crypto.org/

CKKS High Level

CKKS[5] (named after Cheon-Kim-Kim-Song, the authors of the 2016 paper that proposed it) is a homomorphic encryption scheme that allows homomorphic evaluation of the following primitive operations:

- Elementary addition of the length n vectors of complex numbers
- Elementary multiplication of length n complex vectors
- Rotation of elements in the vector (in the direction of changing the circulation)
- Complex conjugation of vector elements

Condition n here is structured according to the Security and precision applied. It has a high standard. The example applied by us will have the value of 4096 - we must consider the security, but also the scaling in log n.

Operations using CKKS are complex. They give approximate results and users must check accurately so as not to affect the correctness of the results.

This kind of restriction is not uncommon for developers. Applications that use GPUs calculate data using number vectors. Floating point numbers use algorithm selection, use multithreading and this leads to a complexity of implementation.

In the following we will try to apply and implement the operations proposed in Julia in order to show how we can use the library in REPL.

```
 using ToyFHE

# Let's play with 8 element vectors
 N = 8;

# Choose some parameters - we'll talk about it later
 𝓡 = NegacyclicRing(2N, (40, 40, 40))
ℤ1329227997568081457402701207104248257/(x¹⁶ + 1)

# We'll use CKKS
params = CKKSParams(𝓡)
CKKS parameters

# We need to pick a scaling factor for a numbers - again we'll talk about
that later
Tscale = FixedRational{2^40}
FixedRational{1099511627776,T} where T

# Let's start with a plain Vector of zeros
plain = CKKSEncoding{Tscale}(zero(𝓡))
```

[5] https://eprint.iacr.org/2016/421.pdf

```
8-element CKKSEncoding{FixedRational{1099511627776,T} where T} with
indices 0:7:
 0.0 + 0.0im
 0.0 + 0.0im
 0.0 + 0.0im
 0.0 + 0.0im
 0.0 + 0.0im
 0.0 + 0.0im
 0.0 + 0.0im
 0.0 + 0.0im

# Ok, we're ready to get started, but first we'll need some keys
kp = keygen(params)
CKKS key pair

kp.priv
CKKS private key

kp.pub
CKKS public key

# Alright, let's encrypt some things:
foreach(i->plain[i] = i+1, 0:7); plain
8-element CKKSEncoding{FixedRational{1099511627776,T} where T} with
indices 0:7:
 1.0 + 0.0im
 2.0 + 0.0im
 3.0 + 0.0im
 4.0 + 0.0im
 5.0 + 0.0im
 6.0 + 0.0im
 7.0 + 0.0im
 8.0 + 0.0im

c = encrypt(kp.pub, plain)
CKKS ciphertext (length 2, encoding
CKKSEncoding{FixedRational{1099511627776,T} where T})
```

```
# And decrypt it again
decrypt(kp.priv, c)
8-element CKKSEncoding{FixedRational{1099511627776,T} where T} with
indices 0:7:
 0.9999999999995506 - 2.7335193113350057e-16im
 1.9999999999989408 - 3.885780586188048e-16im
  3.000000000000205 + 1.6772825551165524e-16im
  4.000000000000538 - 3.885780586188048e-16im
  4.999999999998865 + 8.382500573679615e-17im
  6.000000000000185 + 4.996003610813204e-16im
  7.000000000001043 - 2.0024593503998215e-16im
  8.000000000000673 + 4.996003610813204e-16im

# Note that we had some noise. Let's go through all the primitive
operations we'll need:
decrypt(kp.priv, c+c)
8-element CKKSEncoding{FixedRational{1099511627776,T} where T} with
indices 0:7:
 1.9999999999991012 - 5.467038622670011e-16im
 3.9999999999978817 - 7.771561172376096e-16im
   6.00000000000041 + 3.354565110233105e-16im
  8.000000000001076 - 7.771561172376096e-16im
   9.99999999999773 + 1.676500114735923e-16im
  12.00000000000037 + 9.992007221626409e-16im
 14.000000000002085 - 4.004918700799643e-16im
 16.000000000001346 + 9.992007221626409e-16im

csq = c*c
CKKS ciphertext (length 3, encoding
CKKSEncoding{FixedRational{1208925819614629174706176,T} where T})

decrypt(kp.priv, csq)
8-element CKKSEncoding{FixedRational{1208925819614629174706176,T} where T}
with indices 0:7:
 0.9999999999991012 - 2.350516767363621e-15im
 3.9999999999957616 - 5.773159728050814e-15im
  9.000000000001226 - 2.534464540987068e-15im
 16.000000000004306 - 2.220446049250313e-15im
  24.99999999998865 + 2.0903753311370056e-15im
  36.00000000000222 + 4.884981308350689e-15im
 49.000000000014595 + 1.0182491378134327e-15im
  64.00000000001077 + 4.884981308350689e-15im
```

It can be seen that CSS is different from ciphertext. I gave a 3-digit ciphertext on a larger scale. We would like to reduce these things before other calculations or run out of "space". It can be done as follows:

```
# To get back down to length 2, we need to `keyswitch` (aka
# relinerarize), which requires an evaluation key. Generating
# this requires the private key. In a real application we would
# have generated this up front and sent it along with the encrypted
# data, but since we have the private key, we can just do it now.

ek = keygen(EvalMultKey, kp.priv)
CKKS multiplication key

csq_length2 = keyswitch(ek, csq)
CKKS ciphertext (length 2, encoding
CKKSEncoding{FixedRational{1208925819614629174706176,T} where T})

# Getting the scale back down is done using modswitching.
csq_smaller = modswitch(csq_length2)
CKKS ciphertext (length 2, encoding
CKKSEncoding{FixedRational{1.099511626783e12,T} where T})

# And it still decrypts correctly (though note we've lost some precision)
decrypt(kp.priv, csq_smaller)
8-element CKKSEncoding{FixedRational{1.099511626783e12,T} where T} with
indices 0:7:
 0.9999999999802469 - 5.005163520332181e-11im
 3.9999999999957723 - 1.0468514951188039e-11im
  8.999999999998249 - 4.7588542623100616e-12im
 16.000000000023014 - 1.0413447889166631e-11im
 24.999999999955193 - 6.187833723406491e-12im
 36.000000000002345 + 1.860733715346631e-13im
  49.00000000001647 - 1.442396043149794e-12im
 63.999999999988695 - 1.0722489563648028e-10im
```

Modswitch (short for module switching) can reduce the size of the encrypted data module, and will lead to the fact that we cannot always do so.

```
ℛ # Remember the ring we initially created
ℤ_1329227997568081457402701207104248257/(x^16 + 1)

 ToyFHE.ring(csq_smaller) # It shrunk!
ℤ_1208925820144593779331553/(x^16 + 1)
```

There is one last operation we will need: rotations. Like switching the above keys, it requires an evaluation key (also called a galois key):

```
gk = keygen(GaloisKey, kp.priv; steps=2)
CKKS galois key (element 25)

decrypt(circshift(c, gk))
decrypt(kp, circshift(c, gk))
8-element CKKSEncoding{FixedRational{1099511627776,T} where T} with
indices 0:7:
  7.000000000001042 + 5.68459112632516e-16im
  8.000000000000673 + 5.551115123125783e-17im
  0.999999999999551 - 2.308655353580721e-16im
 1.9999999999989408 + 2.7755575615628914e-16im
  3.000000000000205 - 6.009767921608429e-16im
  4.000000000000538 + 5.551115123125783e-17im
  4.999999999998865 + 4.133860996136768e-17im
  6.000000000000185 - 1.6653345369377348e-16im

# And let's compare to doing the same on the plaintext

circshift(plain, 2)
8-element OffsetArray(::Array{Complex{Float64},1}, 0:7) with eltype
Complex{Float64} with indices 0:7:
 7.0 + 0.0im
 8.0 + 0.0im
 1.0 + 0.0im
 2.0 + 0.0im
 3.0 + 0.0im
 4.0 + 0.0im
 5.0 + 0.0im
 6.0 + 0.0im
```

I showed how to use the HE library. We will have to build the neural network application using these functions, and form it.

The machine learning model

To understand machine learning models or use the FLUX.js[6] library you can follow tutorials on Julia Academy[7]. We will continue to use the convolutional neural network model in the Flux zoo. We will have the same data preparation models and we will modify the models.

[6] https://fluxml.ai/Flux.jl/stable/

[7] https://juliaacademy.com/p/introduction-to-machine-learning

```
function reshape_and_vcat(x)
    let y=reshape(x, 64, 4, size(x, 4))
        vcat((y[:,i,:] for i=axes(y,2))...)
    end
end

model = Chain(
    # First convolution, operating upon a 28x28 image
    Conv((7, 7), 1=>4, stride=(3,3), x->x.^2),
    reshape_and_vcat,
    Dense(256, 64, x->x.^2),
    Dense(64, 10),
)
```

The model described is the one applied in the work "Secure Outsourced Matrix Computation and Application to Neural Networks"[8]. It will use the same scheme model but with the following differences:

1. The paper encrypts the model, we will not do it for simplicity

2. In our case we will apply vectors in each layer

3. In our case we will have a higher accuracy (approximately 98.6% compared to 98.1%).

Another thing is given by the activation functions x. ^ 2. If in the other models tanh or relu functions are used, light functions in cases like plain text, in the case of encryption functions they become very difficult to apply (we have to find polynomial approximation). Softmax was removed from the base model but I applied a logitcrossentropy function. It could also be done with softmax and we could do it with decryption on the client.

Performing the operations efficiently

In this chapter we will explain the type of operation that can be done. We can apply:

• Convolutions

• Square in the element

• Multiplication of the matrix

Square in the element is an easy function as we can read above. We will talk in the following about the other two. The examples will be based on a size of 64 (element vector 4096).

[8] https://eprint.iacr.org/2018/1041.pdf

Convolution

What is convulsion and how does it work? Take a 7x7 window (as in the example) from the initial matrix and each element in the window will be multiplied by an element in the convolution mask. Move the window over others (for example step 3 - so we will move 3 elements) and repeat the process (having the same mask). An example of a 3x3 convolution with step (2,2) can be seen below. Blue - input and green - output.

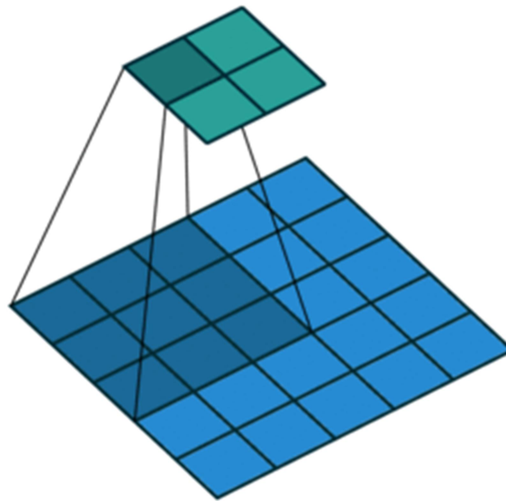

Figure 1 https://github.com/vdumoulin/conv_arithmetic

We can see that we have convolutions on 4 channels. The process will be repeated 3 times with various masks. I checked what we need to do and now apply. A preprocessing model can be made per client to simplify the works.

What will we do:

We will divide and preprocess several convolution models (we will extract 7x7 models from the images), and we will obtain 64 matrices of size 7x7. We will add the same position in each window in a vector and we will get a vector with 64 elements in each image. For a 64x64 element vector a batch of 64 (49 64x64 size matrices) will be obtained. The convolution will become a multiplication. We will scalarly multiply the whole matrix with the appropriate elemental mask and we will thus obtain the convolution or more precisely the result. It can be implemented as follows:

```
function public_preprocess(batch)
    ka = OffsetArray(0:7, 0:7)
    # Create feature extracted matrix
    I = [[batch[i'*3 .+ (1:7), j'*3 .+ (1:7), 1, k] for i'=ka, j'=ka] for
k = 1:64]

    # Reshape into the ciphertext
    Iᵢⱼ = [[I[k][l...][i,j] for k=1:64, l=product(ka, ka)] for i=1:7,
j=1:7]
end

Iᵢⱼ = public_preprocess(batch)

# Evaluate the convolution
weights = model.layers[1].weight
conv_weights = reverse(reverse(weights, dims=1), dims=2)
conved = [sum(Iᵢⱼ[i,j]*conv_weights[i,j,1,channel] for i=1:7, j=1:7) for
channel = 1:4]
conved = map(((x,b),)->x .+ b, zip(conved, model.layers[1].bias))
```

which (modulo a reordering of the dimension) gives the same answer as, but using operations

```
model.layers[1](batch)
```

Adding the encryption operations, we have:

```
Iᵢⱼ = public_preprocess(batch)
C_Iᵢⱼ = map(Iᵢⱼ) do Iij
    plain = CKKSEncoding{Tscale}(zero(plaintext_space(ckks_params)))
    plain .= OffsetArray(vec(Iij), 0:(N÷2-1))
    encrypt(kp, plain)
end

weights = model.layers[1].weight
conv_weights = reverse(reverse(weights, dims=1), dims=2)
conved3 = [sum(C_Iᵢⱼ[i,j]*conv_weights[i,j,1,channel] for i=1:7, j=1:7)
for channel = 1:4]
conved2 = map(((x,b),)->x .+ b, zip(conved3, model.layers[1].bias))
conved1 = map(ToyFHE.modswitch, conved2)
```

A keyswitch will not be required because the weights are public and we have not extended the ciphertext.

Matrix multiply

One of the main elements in multiplying matrices is that we can rearrange the indices. We will take into account the fact that the elements of the matrix that belong to the vector can be sorted. In case of moving the vector with a multiple size of the row size, an interesting effect will be obtained, namely the rotation of the columns. This is a sufficient hypothesis in multiplying the matrix. We will show the following below:

```
function matmul_square_reordered(weights, x)
    sum(1:size(weights, 1)) do k
        # We rotate the columns of the LHS and take the diagonal
        weight_diag = diag(circshift(weights, (0,(k-1))))
        # We rotate the rows of the RHS
        x_rotated = circshift(x, (k-1,0))
        # We do an elementwise, broadcast multiply
        weight_diag .* x_rotated
    end
end

function matmul_reorderd(weights, x)
    sum(partition(1:256, 64)) do range
        matmul_square_reordered(weights[:, range], x[range, :])
    end
end

fc1_weights = model.layers[3].W
x = rand(Float64, 256, 64)
@assert (fc1_weights*x) ≈ matmul_reorderd(fc1_weights, x)
```

Making it nicer

We can make things look even better. The code works and let's show its running (without configuring the parameters):

```
ek = keygen(EvalMultKey, kp.priv)
gk = keygen(GaloisKey, kp.priv; steps=64)

Iᵢⱼ = public_preprocess(batch)
C_Iᵢⱼ = map(Iᵢⱼ) do Iij
    plain = CKKSEncoding{Tscale}(zero(plaintext_space(ckks_params)))
    plain .= OffsetArray(vec(Iij), 0:(N÷2-1))
    encrypt(kp, plain)
end

weights = model.layers[1].weight
conv_weights = reverse(reverse(weights, dims=1), dims=2)
conved3 = [sum(C_Iᵢⱼ[i,j]*conv_weights[i,j,1,channel] for i=1:7, j=1:7)
for channel = 1:4]
conved2 = map(((x,b),)->x .+ b, zip(conved3, model.layers[1].bias))
conved1 = map(ToyFHE.modswitch, conved2)

Csqed1 = map(x->x*x, conved1)
Csqed1 = map(x->keyswitch(ek, x), Csqed1)
Csqed1 = map(ToyFHE.modswitch, Csqed1)

function encrypted_matmul(gk, weights, x::ToyFHE.CipherText)
    result = repeat(diag(weights), inner=64).*x
    rotated = x
    for k = 2:64
        rotated = ToyFHE.rotate(gk, rotated)
        result += repeat(diag(circshift(weights, (0,(k-1)))), inner=64) .*
rotated
    end
    result
end
```

```
fq1_weights = model.layers[3].W
Cfq1 = sum(enumerate(partition(1:256, 64))) do (i,range)
    encrypted_matmul(gk, fq1_weights[:, range], Csqed1[i])
end

Cfq1 = Cfq1 .+ OffsetArray(repeat(model.layers[3].b, inner=64), 0:4095)
Cfq1 = modswitch(Cfq1)

Csqed2 = Cfq1*Cfq1
Csqed2 = keyswitch(ek, Csqed2)
Csqed2 = modswitch(Csqed2)

function naive_rectangular_matmul(gk, weights, x)
    @assert size(weights, 1) < size(weights, 2)
    weights = vcat(weights, zeros(eltype(weights), size(weights, 2)-
size(weights, 1), size(weights, 2)))
    encrypted_matmul(gk, weights, x)
end

fq2_weights = model.layers[4].W
Cresult = naive_rectangular_matmul(gk, fq2_weights, Csqed2)
Cresult = Cresult .+ OffsetArray(repeat(vcat(model.layers[4].b,
zeros(54)), inner=64), 0:4095)
```

We will propose some abstractions that will make things easier. We will move from encryption and machine learning to code creation and implementation. Julia allows abstractions and we will build some. The whole convolution extraction process can be encapsulated, bringing it to a customized matrix:

```
using BlockArrays

"""
    ExplodedConvArray{T, Dims, Storage} <: AbstractArray{T, 4}

Represents a an `nxmx1xb` array of images, but rearranged into a
series of convolution windows. Evaluating a convolution compatible
with `Dims` on this array is achievable through a sequence of
scalar multiplications and sums on the underling storage.
"""
struct ExplodedConvArray{T, Dims, Storage} <: AbstractArray{T, 4}
    # sx*sy matrix of b*(dx*dy) matrices of extracted elements
    # where (sx, sy) = kernel_size(Dims)
    #       (dx, dy) = output_size(DenseConvDims(...))
    cdims::Dims
    x::Matrix{Storage}
    function ExplodedConvArray{T, Dims, Storage}(cdims::Dims,
storage::Matrix{Storage}) where {T, Dims, Storage}
        @assert all(==(size(storage[1])), size.(storage))
        new{T, Dims, Storage}(cdims, storage)
    end
end
Base.size(ex::ExplodedConvArray) = (NNlib.input_size(ex.cdims)..., 1,
size(ex.x[1], 1))

function ExplodedConvArray{T}(cdims, batch::AbstractArray{T, 4}) where {T}
    x, y = NNlib.output_size(cdims)
    kx, ky = NNlib.kernel_size(cdims)
    stridex, stridey = NNlib.stride(cdims)
    kax = OffsetArray(0:x-1, 0:x-1)
    kay = OffsetArray(0:x-1, 0:x-1)
    I = [[batch[i'*stridex .+ (1:kx), j'*stridey .+ (1:ky), 1, k] for
i'=kax, j'=kay] for k = 1:size(batch, 4)]
    Iᵢⱼ = [[I[k][l...][i,j] for k=1:size(batch, 4), l=product(kax, kay)]
for (i,j) in product(1:kx, 1:ky)]
    ExplodedConvArray{T, typeof(cdims), eltype(Iᵢⱼ)}(cdims, Iᵢⱼ)
end

function NNlib.conv(x::ExplodedConvArray{<:Any, Dims},
weights::AbstractArray{<:Any, 4}, cdims::Dims) where {Dims<:ConvDims}
    blocks = reshape([
Base.ReshapedArray(sum(x.x[i,j]*weights[i,j,1,channel] for i=1:7, j=1:7),
(NNlib.output_size(cdims)...,1,size(x, 4)), ()) for channel = 1:4
],(1,1,4,1))
    BlockArrays._BlockArray(blocks, BlockArrays.BlockSizes([8], [8],
[1,1,1,1], [64]))
End
```

Here I used BlockArrays to represent the 8x8x4x64 matrix as 4 8x8x1x64 matrices. It can be seen that we already have a more beautiful model on the unencrypted matrices:

```
cdims = DenseConvDims(batch, model.layers[1].weight; stride=(3,3),
padding=(0,0,0,0), dilation=(1,1))
DenseConvDims: (28, 28, 1) * (7, 7) -> (8, 8, 4), stride: (3, 3) pad: (0,
0, 0, 0), dil: (1, 1), flip: false

a = ExplodedConvArray{eltype(batch)}(cdims, batch);

model(a)
10×64 Array{Float32,2}:
[snip]
```

How do we implement things in encryption? We will have to do two operations:

1. Create the ExplodedConvArray structure and we will get an encrypted text for each field. The operations in the structure have the same functions as those in the original structure and have the same functions as the homomorphic ones.

2. Certain functions will need to be intercepted and have different results.

Julia has the opportunity to apply these two things. This is done through your own compiler through the Cassette.jl[9] function. Requirement number 2 can be rewritten as follows:

```
# Define Matrix multiplication between an array and an encrypted block
array
function (*::Encrypted{typeof(*)})(a::Array{T, 2},
b::Encrypted{<:BlockArray{T, 2}}) where {T}
    sum(a*b for (i,range) in enumerate(partition(1:size(a, 2),
size(b.blocks[1], 1))))
end

# Define Matrix multiplication between an array and an encrypted array
function (*::Encrypted{typeof(*)})(a::Array{T, 2}, b::Encrypted{Array{T,
2}}) where {T}
    result = repeat(diag(a), inner=size(a, 1)).*x
    rotated = b
    for k = 2:size(a, 2)
        rotated = ToyFHE.rotate(GaloisKey(*), rotated)
        result += repeat(diag(circshift(a, (0,(k-1)))), inner=size(a, 1))
.* rotated
    end
    result
end
```

In the final implementation the user must apply everything very quickly:

[9] https://github.com/JuliaLabs/Cassette.jl

```
kp = keygen(ckks_params)
ek = keygen(EvalMultKey, kp.priv)
gk = keygen(GaloisKey, kp.priv; steps=64)

# Create evaluation context
ctx = Encrypted(ek, gk)

# Do public preprocessing
batch = ExplodedConvArray{eltype(batch)}(cdims, batch);

# Run on encrypted data under the encryption context
Cresult = ctx(model)(encrypt(kp.pub, batch))

# Decrypt the answer
decrypt(kp, Cresult)
```

Of course, certain things may not be optimal. $\mathscr{R}$ ring - the function that modifies mods, keyswitch, etc. - can lead to the establishment of certain rules between accuracy, security and performance. The final product implies that the compiler analyzes, runs encrypted and parameterizes the code. Finally it generates the program.

Conclusion

The automatic creation and execution of safe calculations is a desideratum of any programmer and system. The metaprogramming system built by Julia can be used as a development platform. There are already attempts in this regard made by RAMPARTS to bring the Julia code to a library (PALISADE FHE). Lately, computing systems have achieved the performance of cryptographic information in homomorphic system with effective evaluation come close to practical utility. The future is near. Using research in the development of algorithms, homomorphic encryption will become mass technology in the field of user data protection.

References

1. https://github.com/JuliaCrypto/ToyFHE.jl
2. https://github.com/FluxML/Flux.jl
3. https://aws.amazon.com/machine-learning/
4. https://github.com/JuliaLabs/Cassette.jl
5. https://eprint.iacr.org/2018/1041.pdf
6. https://juliaacademy.com/p/introduction-to-machine-learning
7. https://fluxml.ai/Flux.jl/stable/
8. https://eprint.iacr.org/2016/421.pdf
9. https://palisade-crypto.org/
10. https://github.com/microsoft/SEAL
11. https://www.cs.cmu.edu/~odonnell/hits09/gentry-homomorphic-encryption.pdf