Extension of the CKKS Cryptosystem: Part 2

Introduction to Homomorphic Cryptosystems - Lecture 7

THE BINARY STEP FUNCTION

The Binary Step Function

Definition

The binary step function (also called Heaviside step function) is zero for negative and one for positive arguments:

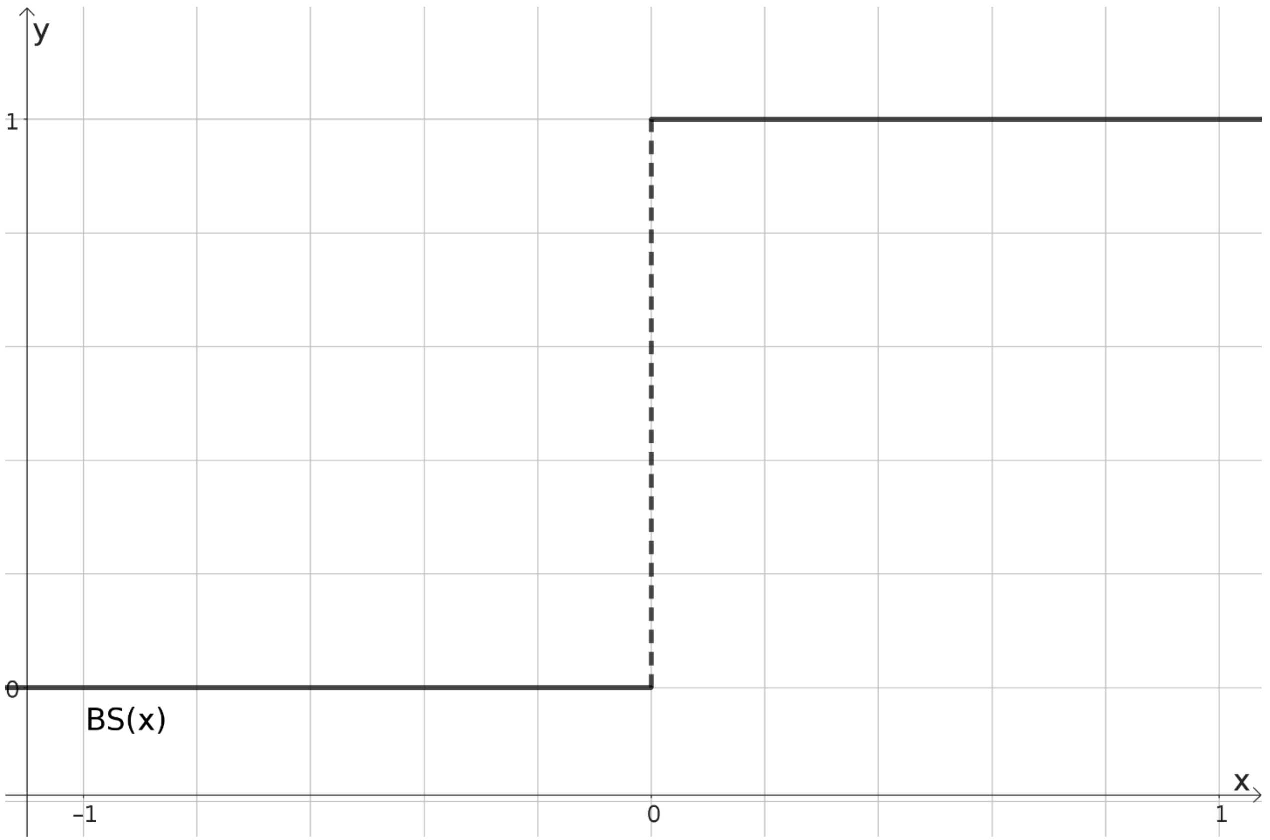
$$BS(x) = \begin{cases} 0 & \text{if } x < 0 \\ 1 & \text{if } x > 0 \end{cases}$$

What about the zero argument?

Depends on the use case. For our use case we define $BS(0) = \frac{1}{2}$.

Use Cases

- Activation function for neural networks
- Filter



The Binary Step Function

Mathematical definition

We want to calculate the BS function homomorphically, so we need a definition based on our limited functions:

$$BS(x) = \frac{\operatorname{sgn}(x) + 1}{2}$$

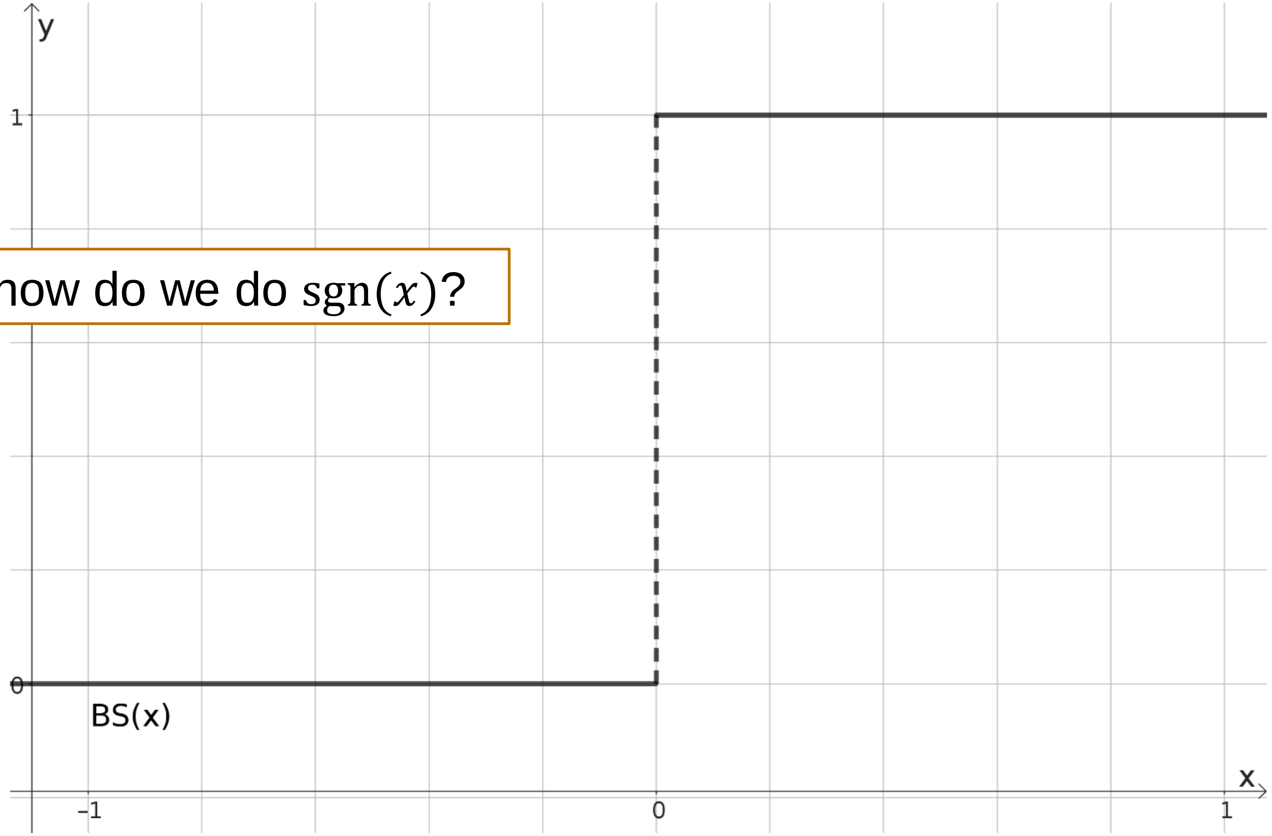
But how do we do $\operatorname{sgn}(x)$?

$$\operatorname{sgn}(x) = \frac{x}{\sqrt{x^2}}$$

This leaves us with this definition:

$$BS(x) = \frac{1}{2} * \left(x * \frac{1}{\sqrt{x^2}} + 1 \right)$$

Which we can implement homomorphically, as we have already extended CKKS by division and square root.



The Binary Step Function

Homomorphic Implementation

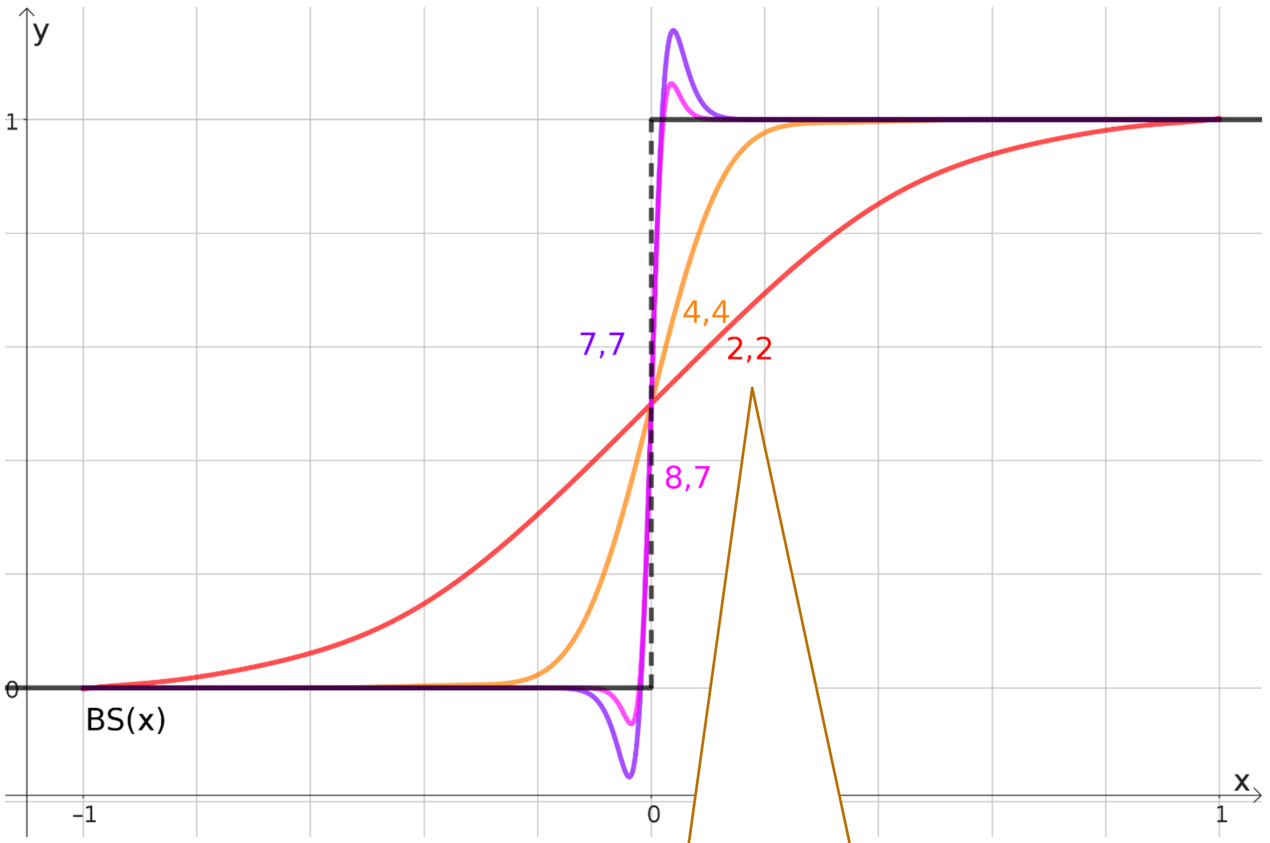
Depending on the number of iterations we do for the division and square root, we achieve varying levels of precision when approximating the BS function homomorphically.

The function contains two complex operations (which we already implemented):

Inverse

$$BS(x) = \frac{1}{2} * \left(x * \frac{1}{\sqrt{x^2}} + 1 \right)$$

square root



<square root iterations>,<inverse iterations>

How can we use the Binary Step Function?

Example Task

From a given set of tuples:

$$\{(\lambda_1, L_1), (\lambda_2, L_2), \dots (\lambda_n, L_n)\},\ \lambda_i \in \mathbb{R}, L_i \in \mathbb{R}$$

Find the λ_x who's corresponding L_x is the minimum of all $L_i, 0 < i \leq n$.

This is obviously easy on unencrypted numbers:

```
def find_smallest_L(lambdas,L):
  min = -1
  min_L = MAX

  for i in range(0, len(L))
    if L[i] < min_L
      min_L = L[i]
      min = i

  return lambdas[min]
```

```
def find_smallest_L(lambdas,L):
  min_L = min(L)

  for i in range(0, len(L))
    if L[i] == min_L
      return lambdas[i]
```

But we cannot do this on encrypted numbers!

How can we use the Binary Step Function?

Example Task

From a given set of tuples:

$$\{(\lambda_1, L_1), (\lambda_2, L_2), \dots (\lambda_n, L_n)\}, \lambda_i \in \mathbb{R}, L_i \in \mathbb{R}$$

Find the λ_{min} who's corresponding L_{min} is the minimum of all $L_i, 0 < i \leq n$.

Homomorphic Solution

With the help of the BS function, we can solve the task even on encrypted numbers. We need two steps:

1. Find $L_{min} = \min_i L_i = \min(\min(\min(L_1, L_2), L_3), \dots)$

   As we already implemented the $\min$ function, this step is no problem.

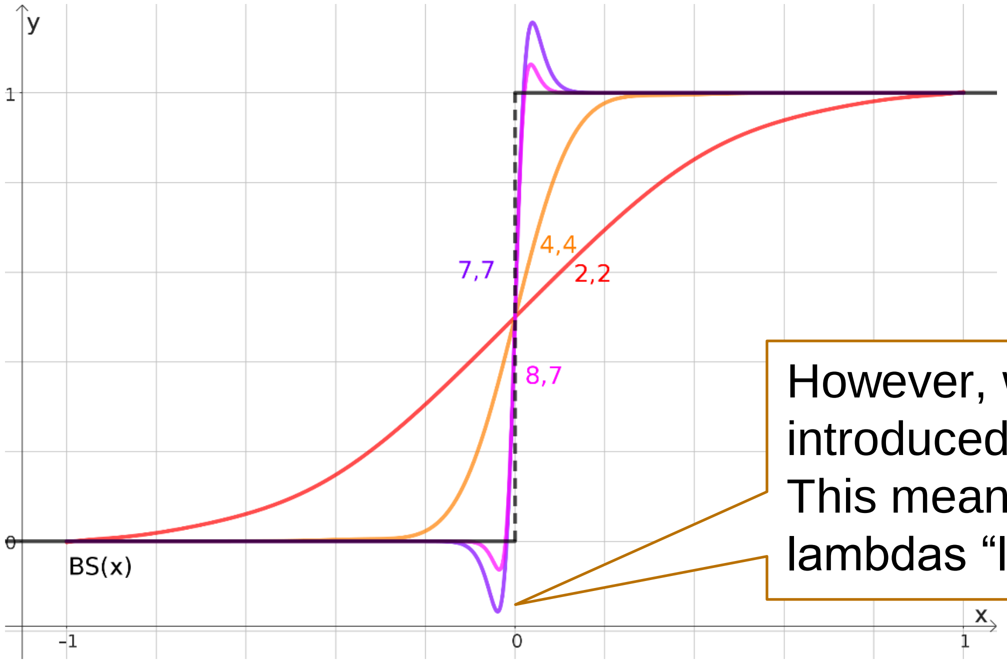
2. Calculate λ_{min}:

$$\lambda_{min} = 2 * \sum_{i=1}^{n} BS(L_{min} - L_i) * \lambda_i$$

The Binary Step Function

Scaling Factor

As we saw already, the BS function becomes less accurate when approaching zero:



However, we cannot get rid of error introduced by the BS function.
This means, that small shares of the other lambdas “leak” into our solution.

This is why we normally include a scaling factor to all calculations involving the BS function:

$$\lambda_{min} = 2 * \sum_{i=1}^{n} BS(s(L_{min} - L_i)) * \lambda_i$$

How can we use the Binary Step Function?

Example Task

From a given set of tuples:

$$\{(\lambda_1, L_1), (\lambda_2, L_2), \dots (\lambda_n, L_n)\},\ \lambda_i \in \mathbb{R}, L_i \in \mathbb{R}$$

Find the λ_{min} who's corresponding L_{min} is the minimum of all $L_i, 0 < i \leq n$.

Homomorphic Solution

With the help of the BS function, we can solve the task even on encrypted numbers. We need two steps:

1. Find $L_{min} = \min_i L_i = \min(\min(\min(L_1, L_2), L_3), \dots)$
   As we already implemented the $\min$ function, this step is no problem.

2. Calculate λ_{min}:

$$\lambda_{min} = 2 * \sum_{i=1}^{n} BS(s(L_{min} - L_i)) * \lambda_i$$

This is a very interesting solution! Can we generalise this to get a function that allows us to do **arbitrary case differentiations**?

EQUALS AND SMALLER THAN FUNCTION

Equals Function

Goal

We want to use the BS function to create a conditional assignment. In pseudo code it would look like this:

```
def if_eq(a,b,c,d):
  if a == b
    return c
  else
    return d
```

Solution

$$EQ(a,b,c,d) = c * BS(a-b) * BS(b-a) * 4 + d * \left(BS(a-b) - BS(b-a)\right)^2$$

Properties

$$EQ(a,b,c,d) = \begin{cases} c & \text{if } a = b \\ d & \text{if } a < b \\ d & \text{if } a > b \end{cases}$$

Smaller Than Function

Goal

We want to use the BS function to create a conditional assignment. In pseudo code it would look like this:

```
def if_st(a,b,c,d):
  if a < b
    return c
  else
    return d
```

Solution

$$HELP(a,b,c) = \frac{1}{2}\Big(c - c\big(BS(a-b) - BS(b-a)\big)\Big) - \frac{1}{2}c * BS(a-b)BS(b-a) * 4$$
$$ST(a,b,c,d) = EQ(HELP(a,b,c), 0, d, c)$$

Properties

$$ST(a,b,c,d) = \begin{cases} d & \text{if } a = b \\ c & \text{if } a < b \\ d & \text{if } a > b \end{cases}$$

But: It’s more a theory

EQ and ST are demonstrations on how we can work with the limited operations in a FHE cryptosystems.

Their real use depend on different factors:

- Are there different options? (Newton-Raphson, etc..)
- What is the order of magnitude of the numbers?
- What is the distance of the compared numbers?

A big distance can increase the error significantly, as both numbers are part of the equation and the BS implementation is not perfect.

Conditional Branching

EQ and ST (and thus, abusing the BS function) enable more complex operations.
In some way they resemble conditional branching, as they also allow to change the control flow based on conditions:

```
def example(bool, input):
  if(bool == 0):
    input = <operation1>
    input = <operation2>
    […]
    return input
  else:
    input = <operation3>
    input = <operation4>
    […]
    return input
```

Input: b, i

$$c_1 = EQ(b, 0,1,0)$$
$$c_2 = EQ(b, 0,0,1)$$

$$r = c_1 * <\text{operation1}> + c_2 * <\text{operation3}>$$
$$r = c_1 * <\text{operation2}> + c_2 * <\text{operation4}>$$
$$[\ldots]$$

Summary – What did we learn today?

Binary Step Function
The BS function is zero for negative values and one for positive values.
It can be implemented homomorphically, but the error is significant.

Conditional Branching
The BS function can be used to implement conditional branching.
This allows the execution of operations based on a condition.