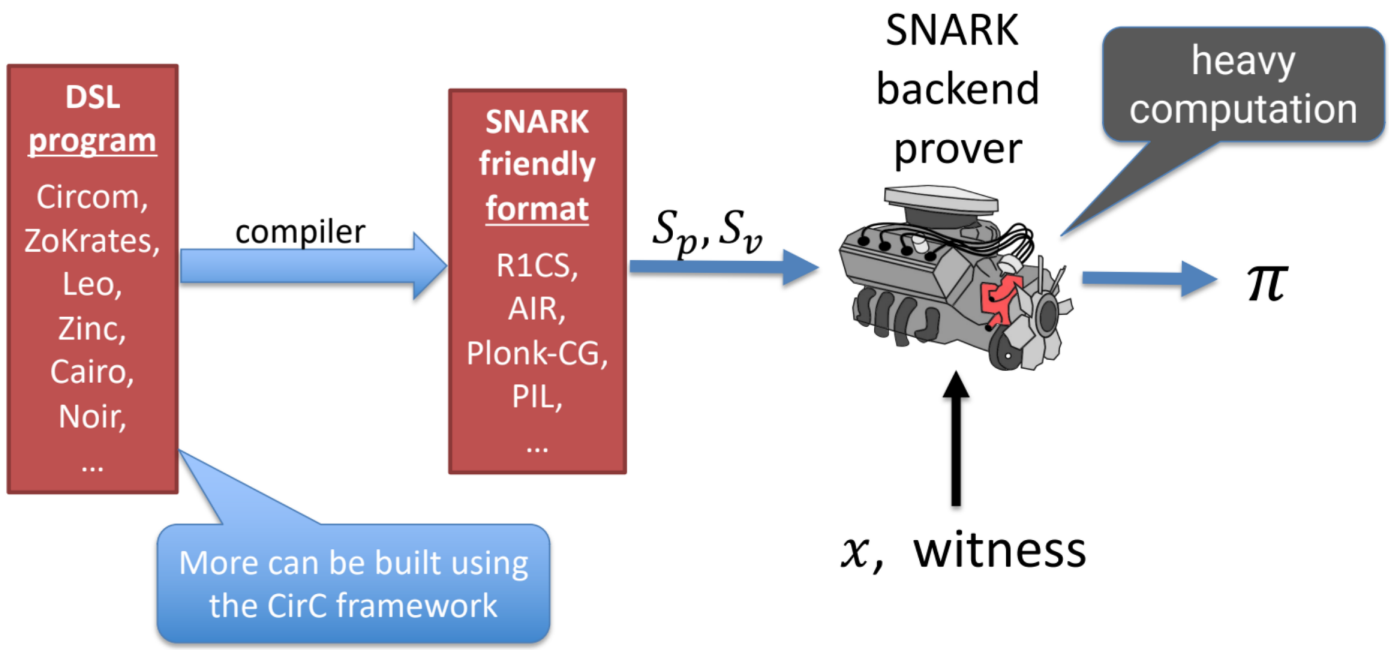
Lesson 6

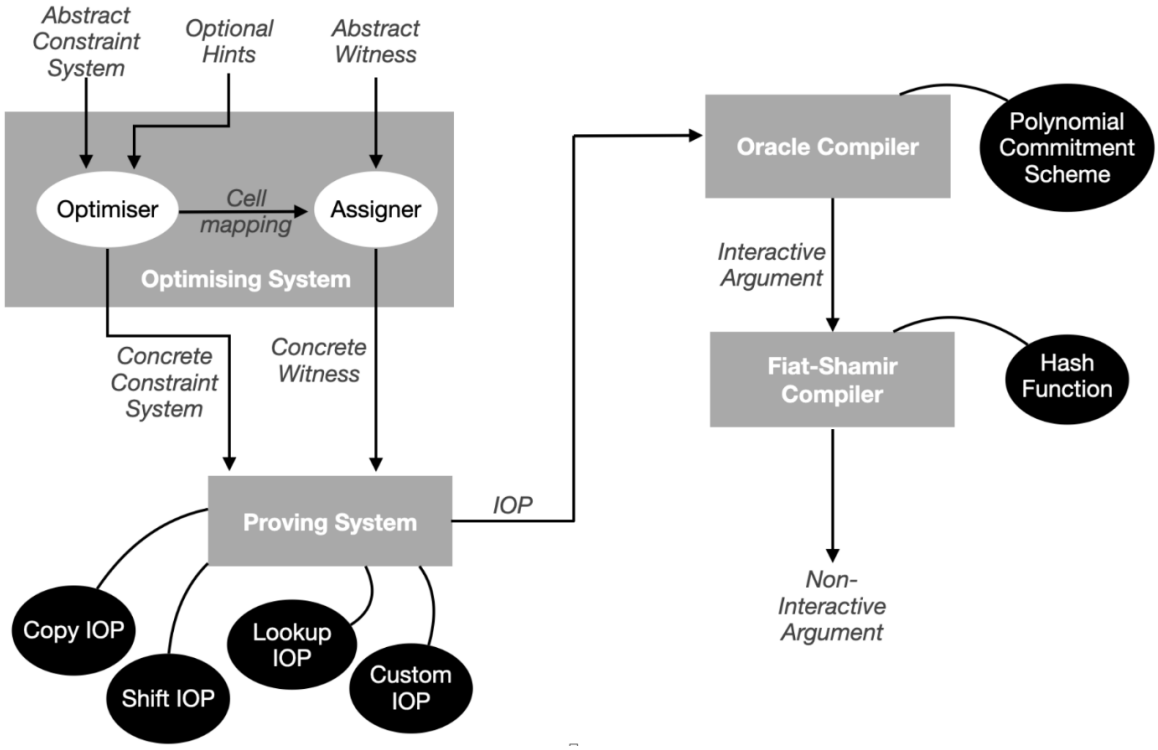
Today's topics

- zkSNARK process
- Polynomial Commitment Schemes
- Fiat Shamir Heuristic
- Activation Functions
- ONNX

A SNARK software system



A zero-knowledge proof typically consists of the following components:



Polynomial Commitment Schemes

Introduction

A polynomial commiment is a short object that "represents" a polynomial, and allows you to verify evaluations of that polynomial, without needing to actually contain all of the data in the polynomial.

That is, if someone gives you a commitment c representing $P(x)$, they can give you a proof that can convince you, for some specific z, what the value of $P(z)$ is.
There is a further mathematical result that says that, over a sufficiently big field, if certain kinds of equations (chosen before z is known) about polynomials evaluated at a random z are true, those same equations are true about the whole polynomial as well.

For example, if $P(z).Q(z) + R(z) = S(z) + 5$ for a particular z, then we know that it's overwhelmingly likely that $P(x).Q(x) + R(x) = S(x) + 5$ in general.
Using such polynomial commitments, we could very easily check all of the above polynomial

equations above - make the commitments, use them as input to generate z, prove what the evaluations are of each polynomial at z, and then run the equations with these evaluations instead of the original polynomials.

A general approach is to have the evaluations in a merkle tree, the leaves of which the verifier can select at random, along with merkle proof of their membership.

Role in ZKPs

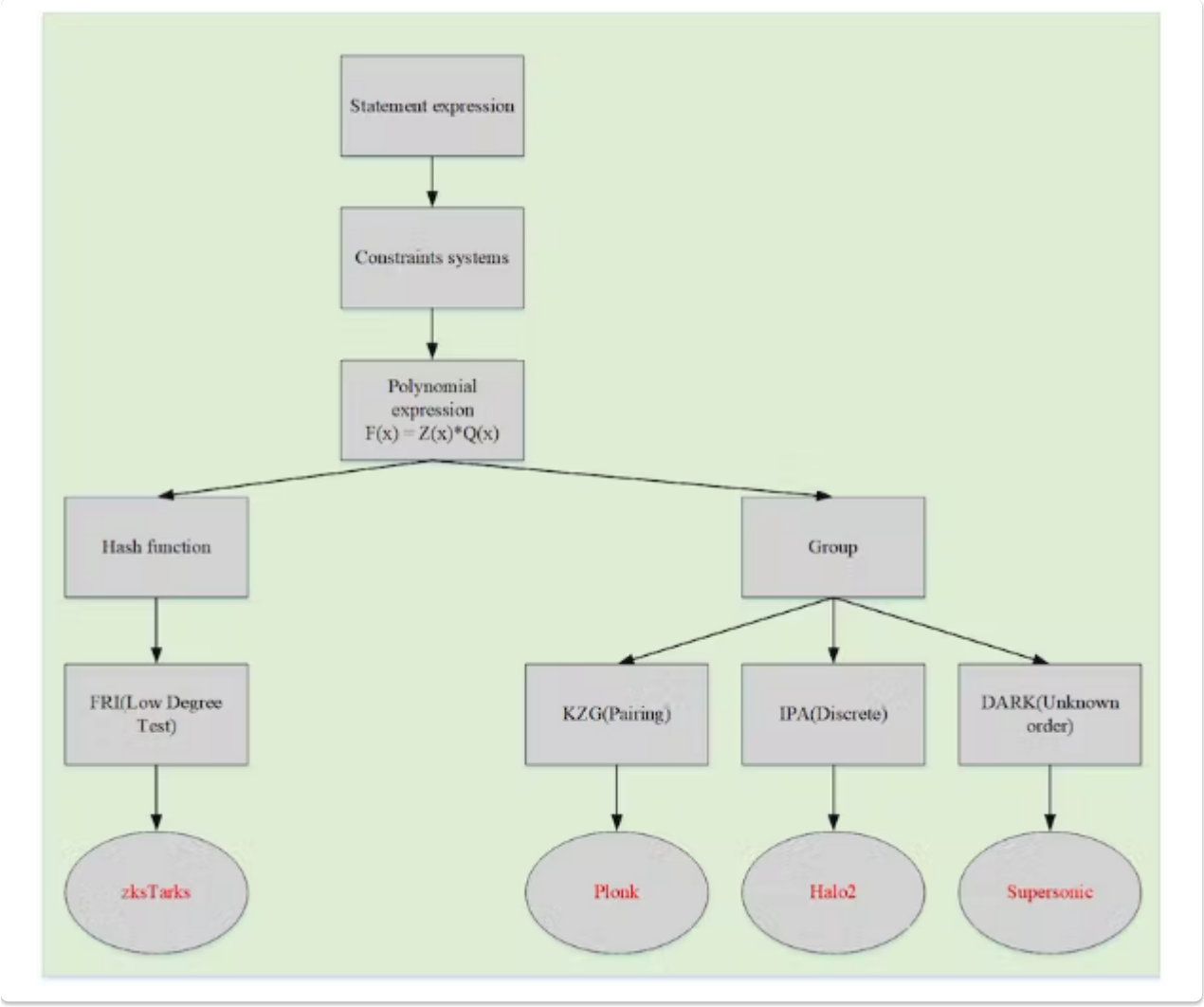
Commitment schemes generally allow the properties of

1. Binding. Given a commitment c, it is hard to compute a different pair of message and randomness whose commitment is c.
   This property guarantees that there is no ambiguity in the commitment scheme, and thus after c is published it is hard to open it to a different value.
2. Hiding. It is hard to compute any information about m given c.

Given the size of the polynomials used in ZKPs , with say 10^8 terms, they help with succinctness

by reducing the size of the information that needs to be passed between the prover and verifier.

Types of PCS



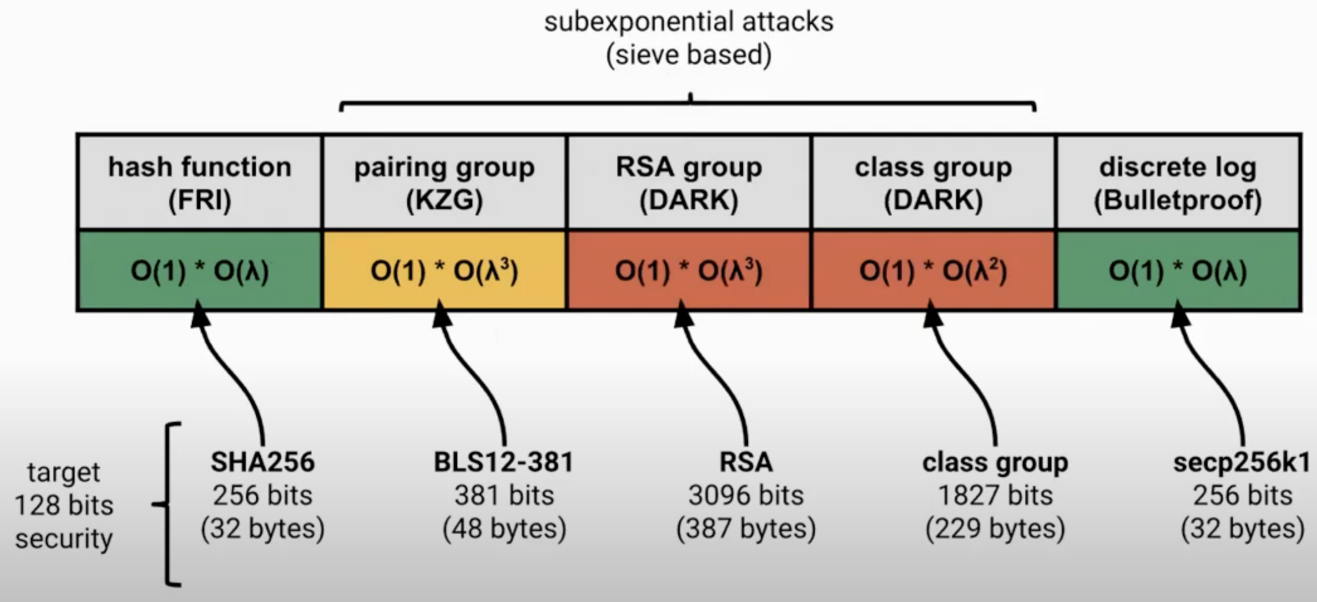
Comparison of Schemes and their underlying assumptions

Taken from [ZKP Study Group VideoSlides]

	hash function	pairing group	RSA group	class group	discrete log
transparent					
succinct					
unbounded					
updatable		curve-specific			
post-quantum					

	hash function	pairing group	UO group	discrete log group
proof size	$O(\log^2(d))$	$O(1)$	$O(\log(d))$	$O(\log(d))$
verifier time	$O(\log^2(d))$	$O(1)$	$O(\log(d))$	$O(d)$
prover time	$O(d*\log(d))$	$O(d)$	$O(d*\log(d))$	$O(d)$

	hash function (FRI)	pairing group (KZG)	UO group (DARK)	discrete log group (Bulletproof)
proof size	$O(\log^2(d))$	$O(1)$	$O(\log(d))$	$O(\log(d))$
verifier time	$O(\log^2(d))$	$O(1)$	$O(\log(d))$	$O(d)$
prover time	$O(d*\log(d))$	$O(d)$	$O(d)$	$O(d)$



KZG

See article
You could create a commitment by taking the coefficients of a polynomial and using these as the leaves of a merkle tree and giving the root as the commitment.
The problem with this simple approach is the amount of computation that needs to be done to show the proof, and the proof has the coefficients in the clear.

A better approach has the following features

1. The commitment size is one group element of an elliptic group that admits pairings. For example, with BLS12_381, that would be 48 bytes.
2. The proof size, *independent* from the size of the polynomial, is also always only one group element. Verification, also independent from the size of the polynomial, requires a two group multiplications and two pairings, no matter what the degree of the polynomial is.

3. The scheme *mostly* hides the polynomial – indeed, an infinite number of polynomials will have exactly the same Kate commitment. However, it is not perfectly hiding: If you can guess the polynomial (for example because it is very simple or in a small set of possible polynomials) you can find out which polynomial was committed to.

Trusted setup procedure

1. From a field $\mathbb{F}_p$ we get a random point s
2. We then create $\{G, sG, s^2G, \dots\}$ up to the degree of the polynomials we are expecting.
   This is our structured reference string
3. The s value is the 'toxic waste' and is deleted
4. The prover will need to evaluate $f(s)$ later but doesn't know s, but does have the s values 'hidden' in the SRS

Homomorphic properties of commitments

We rely upon the fact that
$com(A) = com(B)$ iff $A = B$

and $com(A+B) = com(A) + com(B)$

Creating the commitment

So our original polynomial is $f(x) =$
$f_0 + f_1 x + f_2 x^2 + \ldots$
and the prover wants to evaluate this at s
We can rewrite our SRS as

$$\{G, sG, s^2G, \ldots\} = \{T_0, T_1, T_2, \ldots\}$$

then we want to rewrite $f(s)$ in terms of
$f_0 T_0 + f_1 T_1 + f_2 T_2 + \ldots$
$= f_0 G + f_1 sG + f_2 s^2 G$
$= (f_0 + f_1 s + f_2 s^2 + \ldots)G$
$= f(s)G$
this is our commitment to f
$= com(f)$
so our commitment is hidden behind the group element G
and $f(s)G$ is a curve point.

Evaluation Proof

The verifier will want the commitment $com(f)$
and also $f(z)$
for some random $z \in \mathbb{F}_p$

Now the verifier wants an evaluation proof $f(z)$
given z and $com(f)$

The equation $f(x) - f(z)$ has a root at z
and we can do our rewrite technique to express this as
$(x - z)h(x)$

So the verifier can ask for an evaluation of $f(z)$ knowing that $h(x)$ must exist if the equalities are all correct

Fiat Shamir Heuristic

The final part of the process is to make our proof non interactive, to understand the process I find thinking in terms of interaction helps, and the final part is just squashing this interaction into one step.
For this we use the ubiquitous Fiat Shamir Heuristic.

Public coin protocols

This terminology simply means that the verifier is using randomness when sending queries to the prover , i.e. like flipping a coin

Random oracle model

Both parties are given blackbox access to a random function

Fiat Shamir Heuristic

In an interactive process, the prover and a verifier engage in a multiple interactions, such as in the billiard ball example, to verify a proof. However, this interaction may not be convenient or efficient, with blockchains, we would like only a single message sent to a verifier smart contract to be sufficient.

The Fiat-Shamir heuristic addresses this limitation by converting an interactive scheme into a non-interactive one, where the prover can produce a convincing proof without the need for interactive communication.
This is achieved by using a cryptographic hash function.

The general process is

1. Setup: The verifier generates a public key and a secret key. The public key is made public, while the secret key remains private.
2. Commitment: The verifier commits to a randomly chosen challenge, typically by hashing it together with some additional data. The commitment is sent to the prover.
3. Response: The prover uses the commitment received from the verifier and its secret key to compute a response. The response is generated by applying the cryptographic hash function to the commitment and the prover's secret key.

4. Verification: The verifier takes the prover's response and checks if it satisfies certain conditions. These conditions are typically defined by the original scheme.

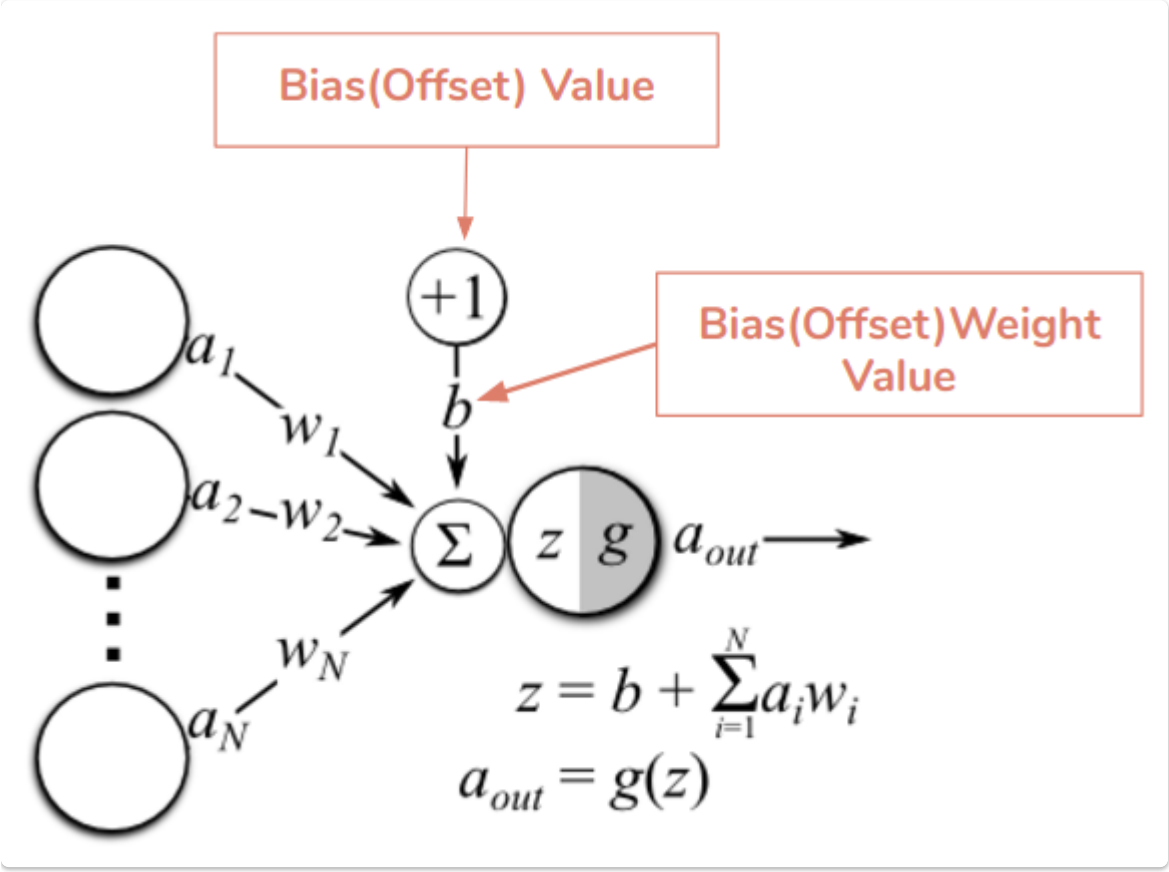
The Fiat-Shamir heuristic is secure under the assumption that the underlying identification scheme is secure and the cryptographic hash function used is collision-resistant. It provides a way to convert interactive protocols into more efficient and practical non-interactive ones without compromising security.

When the prover and verifier are interacting, there are a small number of queries from the verifier that would allow the prover to cheat, because we don't have complete soundness.

In the Fiat Shamir heuristic we assume that the prover is computationally bounded (i.e this is an argument of knowledge as opposed to a proof).

Activation functions

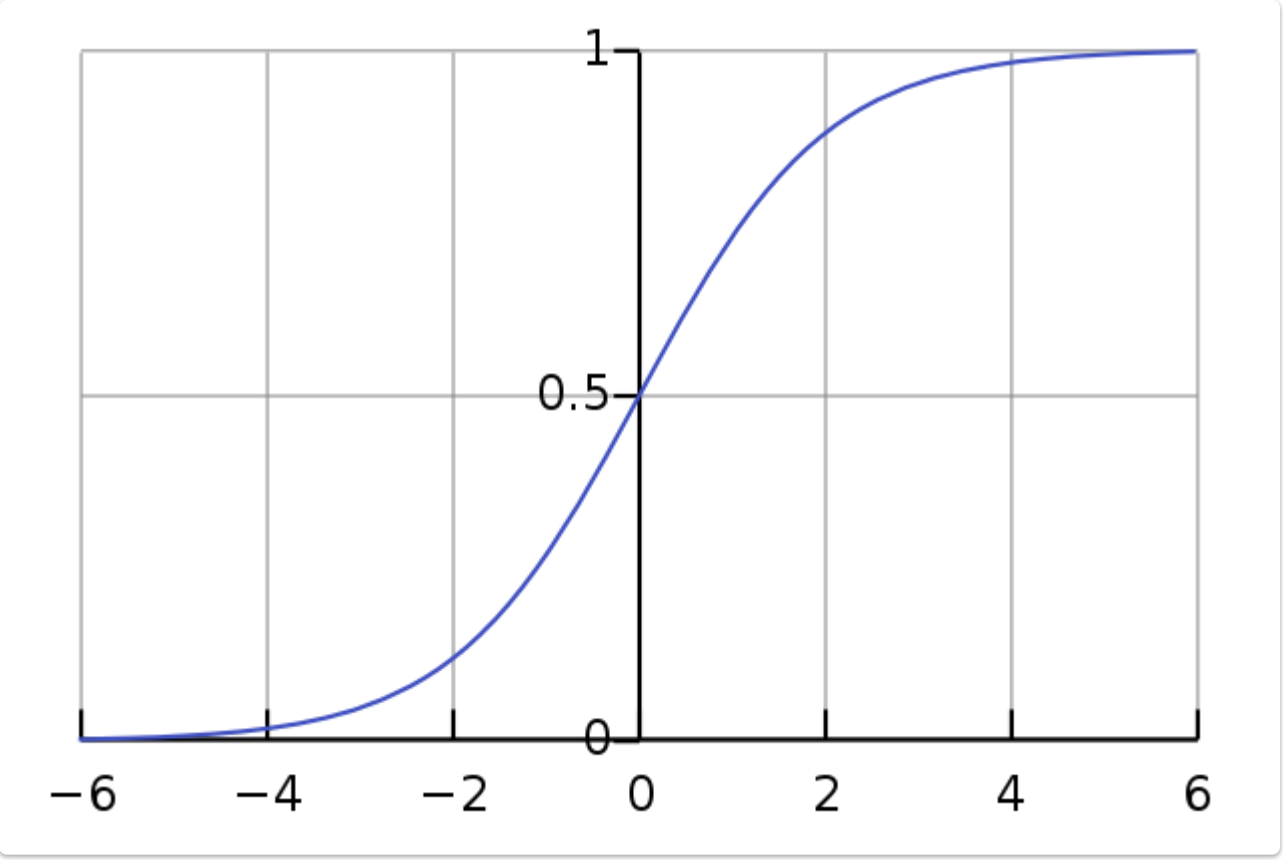
An activation function takes the summed inputs to the node and produces an output of a certain type.



Non linear activation functions have proved popular, they have been successful in learning complex structures in the input data.

Sigmoid

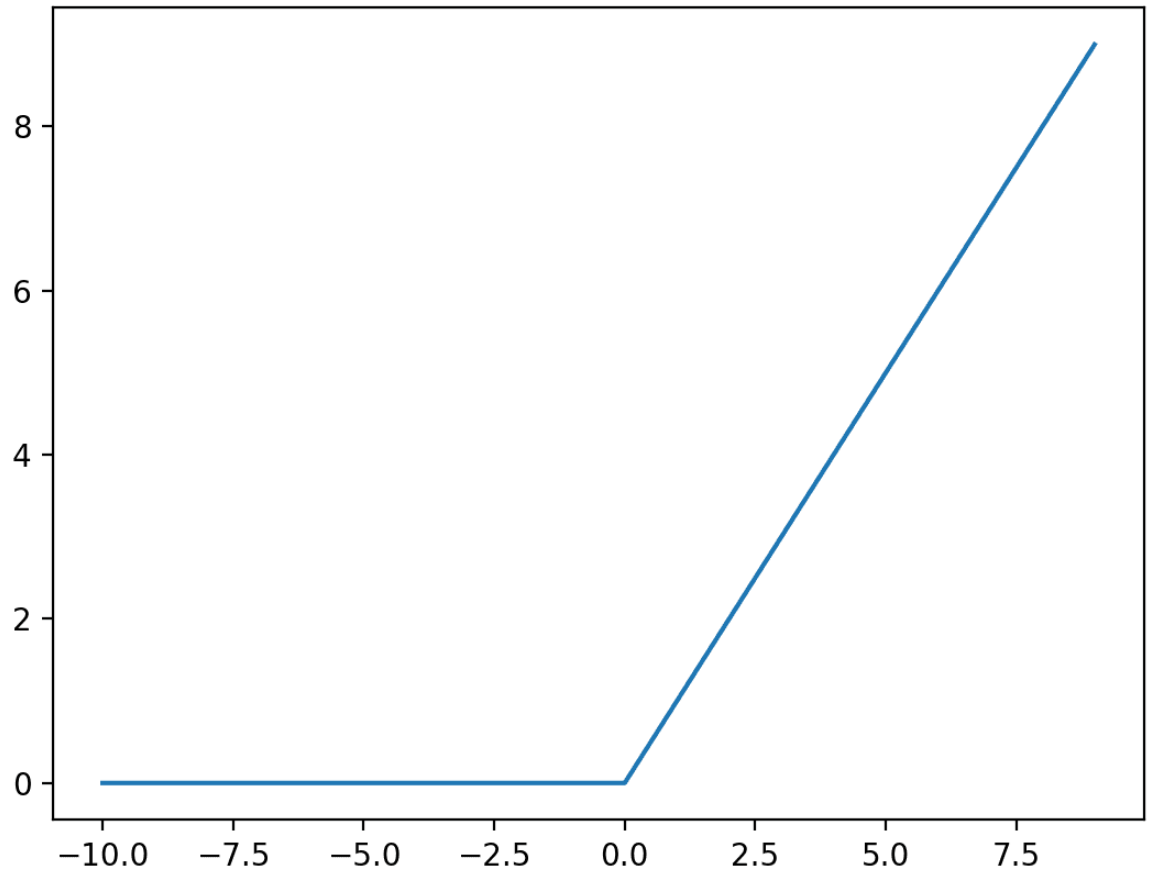
The sigmoid activation function transforms values to the range 0 to 1.



Relu

See Article

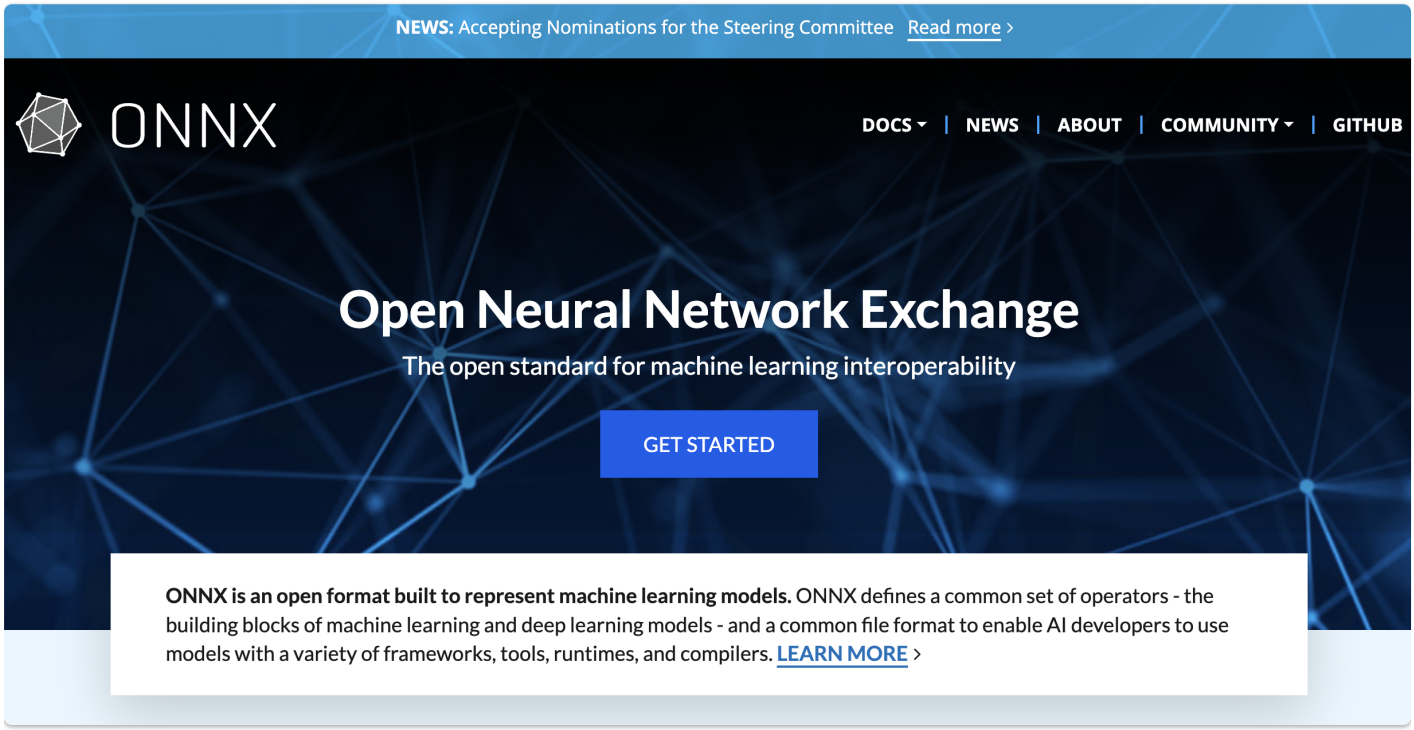
The Rectified Linear activation function will output the input directly if it is positive, otherwise it will output zero.



The Relu function is popular as it acts like a linear activation function, while still allowing complex structures to be learnt.
The linear nature of the function allows optimisations to be applied more easily.

ONNX

See Site



ONNX is an open format built to represent machine learning models. ONNX defines a common set of operators - the building blocks of machine learning and deep learning models - and a common file format to enable AI developers to use models with a variety of frameworks, tools, runtimes, and compilers

ONNX provides a definition of an extensible computation graph model, as well as definitions of built-in operators and standard data types.

Each computation dataflow graph is structured as a list of nodes that form an acyclic graph.

Nodes have one or more inputs and one or more outputs.
Each node is a call to an operator. The graph also has metadata to help document its purpose, author, etc.

Operators are implemented externally to the graph, but the set of built-in operators are portable across frameworks.
Every framework supporting ONNX will provide implementations of these operators on the applicable data types.

EZKL allows you to build a computational graph and output it in an ONNX format.

Orion provides essential components and a new ONNX runtime for building verifiable Machine Learning models based on STARKS.