Implement new operators in Orion

?

The Orion Framework offers an open-source ONNX runtime implementation for Validity and ZK Machine Learning. Are you interested in contributing? We sincerely appreciate your interest. This is exactly how we'll build a robust and transparent AI ecosystem! In this tutorial, you'll learn how to contribute to theOrion repository by implementing from scratch a new operator.

Throughout this tutorial, any concept that is directly explained in the official documentation will be met with a reference guiding you to the respective source. Feel free to dive in.

Code Structure

Orion repo uses Scarb, a Cairo package manager. You can find all information about Scarb and Cairo installation here .

The repository is structured as follows:

```

Copy . ├── LICENSE ├── README.md ├── Scarb.toml ├── book.json ├── cairo_project.toml ├── docgen ├── docs ├── nodegen ├── src │ ├── lib.cairo │ ├── numbers │ ├── numbers.cairo │ ├── operators │ ├── operators.cairo │ ├── tests │ ├── tests.cairo │ └── utils.cairo └── target

```

In thesrc directory, you'll find four distinct folders:

- numbers
- : This folder contains a complete implementation of Signed Integer and Fixed Point.
- operators
- : This directory includes a set of functions and operations used in calculating neural network models.
- tests
- : This is the location where we'll test our code.
-

In this tutorial we will focus onoperators directory, as we will implement a new operator from scratch.

What are Orion Operators?

Orion operators serve as the foundational components of machine learning models compliant with ONNX ops. ONNX is an open format for representing machine learning models that allows interoperability between various deep learning frameworks. It enables models to be trained in one framework and deployed in another without the need for extensive model conversion.

Orion operators represent specific computations or operations performed by machine learning models. Each operator defines a specific functionality, such as convolution, pooling, activation functions, matrix operations, and more. These operators are defined using a set of attributes and inputs that describe their behaviour and dependencies.

Ensuring compatibility with ONNX operators facilitates integration into the ONNX ecosystem. This enables researchers and developers to pre-train models using their preferred framework, before executing verifiable inferences with Orion.

We implemented two different types of operators, each having their own trait:

- tensor (TensorTrait)
- : This represents a full implementation of multi-dimensional arrays.
- nn (NNTrait)
-
    - These are operators designed for building neural networks.
-

Use Resources:

- Full list of ONNX Operators
- .
- Current list of operators supported by Orion. *

How to contribute?

This tutorial will focus specifically on implementing a new Operator in the Orion repository, and will not cover the entirety of the contribution guidelines. If you intend to contribute to Orion, we kindly ask that you read carefully theContribution

Guidelines .

How to implement new Orion Operators?

In this section, I will guide you through the process of adding new operators to the Orion repository. To illustrate this, we will build the Softmax operator from scratch.

What is Softmax?

It is a non-linear activation function that takes a vector of real numbers as input and transforms them into a probability distribution over multiple classes. It's defined as follows:

! softmax (x i) = e x i ∑ j = 1 n e x j \text{softmax}(x_i) = \frac{e^{x_i}}{\sum_{j=1}^n e^{x_j}} In other words, the softmax function exponentiates each element of the input vector and divides it by the sum of exponentiated values across all elements. This normalization ensures that the output values lie between 0 and 1, and their sum adds up to 1, resembling a probability distribution.

Best practices before implementing an operator

Before implementing an operator in Orion, I recommend that you:

1. Read the corresponding ONNX operatordocumentation
2. .
3. Understand how theONNX backend
4. has implemented it. It's essential to maintain the operator's interface consistent with the one in ONNX.
5. Consider whether the operator should be implemented inNNTrait
6. orTensorTrait
7. .
8.

Start coding!

Step 1: Add softmax to NNTrait

Since Softmax is a neural network operator, it needs to be implemented inNNTrait . It accepts an input tensor of a generic type 'T' and an axis along which the softmax computation will occur. Given that the resulting values must range between 0 and 1, it should return a tensor of fixed-point numbers, retaining the same shape as the input tensor.

Insrc/operators/nn/core.cairo we add the softmax toNNTrait .

```

Copy traitNNTrait { //... fnsoftmax(tensor:@Tensor, axis:usize)->Tensor; }

```

Step 2: Add the business logic

In thesrc/operators/nn/functional directory, create a new file namedsoftmax.cairo .

Thefunctional folder is where all the business logic resides. All functions should be implemented with generic type.

A softmax function can be implemented as follows:

Softmax(input, axis) = Exp(input) / ReduceSum(Exp(input), axis=axis)

So we can leverage theexp andreduce_sum operators fromTensorTrait to implement softmax.

Here's the implementation insoftmax.cairo :

```

Copy useorion::operators::tensor::core::{Tensor,TensorTrait};

/// Cf: NNTrait::softmax docstring fnsoftmax< T, implTTensor:TensorTrait, implTTensor:TensorTrait, implTTensorDiv:Div>, implTCopy:Copy, implTDrop:Drop,

    ( z:@Tensor, axis:usize )->Tensor { letexp_tensor=z.exp(); letsum=exp_tensor.reduce_sum(axis,true); letsoftmax=exp_tensor/sum;

returnsoftmax; }
```

```

Step 3: Add softmax to the implementations

Now, we need to add the softmax function into the different representations. Innn/implementations/nn_fp8x23.cairo , import the business logic and add the softmax implementation.

```

Copy // In nn_fp8x23.cairo

implFP8x23NNofNNTrait { // [...] fnsoftmax(tensor:@Tensor, axis:usize)->Tensor { functional::softmax::softmax(tensor, axis) } }

```

Do the same for all other fixed point implementations (FP16x16NN ,FP32x32NN ,FP64x64NN ). As softmax only support fixed point tensors, it should panic for other implementations. Below an example withU32NN .

```

Copy // In nn_u32.cairo

implU32NNofNNTrait { // [...] fnsoftmax(tensor:@Tensor, axis:usize)->Tensor { panic(array!['notsupported!']) } }

```

Step 4: Write the docstring

Navigate back tooperators/nn/core.cairo and prior to the declaration of the softmax function, write the docstring and list it preceding the trait as shown below. This step is useful for generating the documentation during the preparation of your Pull Request, which can be achieved withscarb run docgen command. We use a docstring style similar toRust's docstring , with a few variations.

```

Copy /// Trait /// /// [...] /// softmax - Computes softmax activations. traitNNTrait{ /// [...] /// # NNTrait::softmax /// ///rust /// fn softmax(tensor: @Tensor<T>, axis: usize) -> Tensor<T>; /// /// /// Applies the Softmax function to an n-dimensional input Tensor rescaling them so that the elements of the n-dimensional output Tensor lie in the range [0,1] and sum to 1. /// /// /// \text{softmax}(x_i) = \frac{e^{x_i}}{\sum_{j=1}^n e^{x_j}} /// /// /// ## Args /// /// * tensor(@Tensor<T>) - The input tensor. /// * axis(usize) - The axis along which to compute the softmax. /// /// ## Returns /// /// A Tensor of fixed point numbers with the same shape than the input Tensor. /// /// ## Type Constraints /// /// Constrain input and output types to fixed point tensors. /// /// ## Examples /// /// rust /// use core::array::{ArrayTrait, SpanTrait}; /// /// use orion::operators::tensor::{TensorTrait, Tensor, FP8x23}; /// use orion::operators::nn::{NNTrait, FP8x23NN}; /// use orion::numbers::{FP8x23, FixedTrait}; /// /// fn softmax_example() -> Tensor<FP8x23> { /// let tensor = TensorTrait::<FP8x23>::new(/// shape: array![2, 2].span(), /// data: array![/// NNTrait::new(0, false), /// NNTrait::new(1, false), /// NNTrait::new(2, false), /// NNTrait::new(3, false), ///] /// .span(), ///); /// /// return NNTrait::softmax(@tensor, 1); /// } /// >>> [[2255697,6132911],[2255697,6132911]] /// // The fixed point representation of /// // [[0.2689, 0.7311],[0.2689, 0.7311]] /// /// fnsoftmax(tensor:@Tensor, axis:usize)->Tensor; }

```

Voilà! We have successfully implemented the softmax function inNNTrait !

How to test the Orion Operator?

Now, let's proceed to testing the softmax operator we've just implemented. When testing an operator in Orion, you should ensure to test across all types of implementation.

Since softmax employs fixed points for intermediate calculations and returns a tensor ofFixedType , it is essential to test it across all fixed point implementations. As of now, Orion supports two fixed point implementations:FP16x16 andFP8x23 .

To simplify the task of writing tests, and get closer to ONNX tests, we've designedNodegen ! It lets you write your test in Python/Numpy, then generate the following Cairo code:

- Input data
- Expected output data
- Your tests
-

First, we'll create asoftmax.py file in thenodegen/node directory. Next, we'll define a softmax function in python. You can find the python function inONNX implementation directory.

```

Copy importnumpyasnp

defsoftmax(x:np.ndarray,axis:int=-1)->np.ndarray: x_max=np.max(x, axis=axis, keepdims=True) tmp=np.exp(x-x_max) s=np.sum(tmp, axis=axis, keepdims=True) returntmp/s

```

Finally, we create a Softmax class, containing tests for each dtypes.

```

Copy importnumpyasnp fromnodegen.nodeimportRunAll from..helpersimportmake_test,to_fp,Tensor,Dtype,FixedImpl,Trait

defsoftmax(x:np.ndarray,axis:int=-1)->np.ndarray: x_max=np.max(x, axis=axis, keepdims=True) tmp=np.exp(x-x_max) s=np.sum(tmp, axis=axis, keepdims=True) returntmp/s

classSoftmax(RunAll):

@staticmethod

We test here with fp8x23 implementation.

deffp8x23():

Create a random numpy array:

x=np.random.randint(-3,3, (2,2)).astype(np.float64)

Ddefine the expected result:

y=softmax(x,0)

Convert the input and output to Tensor class, similar to Orion's Tensor struct:

x=Tensor(Dtype.FP8x23, x.shape,to_fp(x.flatten(), FixedImpl.FP8x23))

Convert the floats values in `y` to fixed points with `to_fp` method:

y=Tensor(Dtype.FP8x23, y.shape,to_fp(y.flatten(), FixedImpl.FP8x23))

Define the name of the generated folder.

name="softmax_fp8x23"

Invoke `make_test` method to generate corresponding Cairo tests:

make_test([x],# List of input tensors. y,# The expected output result. "NNTrait::softmax(@input_0, 0)",# The code signature. name,# The name of the generated folder. Trait.NN# The trait, if the function is present in either the TensorTrait or NNTrait.)

We test here with fp16x16 implementation.

@staticmethod deffp16x16(): x=np.random.uniform(-3,3, (2,2)).astype(np.float64) y=softmax(x,1)

x=Tensor(Dtype.FP16x16, x.shape,to_fp(x.flatten(), FixedImpl.FP16x16)) y=Tensor(Dtype.FP16x16, y.shape,to_fp(

y.flatten(), FixedImpl.FP16x16))

name="softmax_fp16x16" make_test([x], y,"NNTrait::softmax(@input_0, 1)", name, Trait.NN)

```

Once set up, you can generate tests and data by executingscarb run nodegen softmax .

The above code will generate 6 test files located intests/src/nodes . As an example, here's the content of thesoftmax_fp8x23.cairo generated file:

```

Copy // softmax_fp8x23.cairo modinput_0; modoutput_0;

useorion::operators::nn::NNTrait; useorion::numbers::FixedTrait; useorion::operators::nn::FP8x23NN; useorion::operators::tensor::FP8x23TensorPartialEq; useorion::utils::assert_eq;

[test]

[available_gas(2000000000)]

fntest_softmax_fp8x23() { letinput_0=input_0::input_0(); letz=output_0::output_0();

lety=NNTrait::softmax(@input_0,0);

assert_eq(y, z); }

```

If you'd like to expand the tests with additional cases, feel free to edit the generated Cairo file.

You're now ready to prepare your Pull Request. Please ensure you thoroughly read the Contribution Guidelines before making your first PR. Your contribution is greatly appreciated, and we sincerely value your interest .

Orion leverages Cairo to guarantee the reliability of inference, providing developers with a user-friendly framework to build complex and verifiable machine learning models. We invite the community to join us in shaping a future where trustworthy AI becomes a reliable resource for all.

Previous MNIST Classification with Orion Next Verifiable Linear Regression Model

Last updated2 months ago
```