Neural network inferencing for PyTorch and TensorFlow with ONNX, WebAssembly System Interface, and WASI NN

Radu Matei

July 7, 2021

This article originally appeared in the Microsoft DeisLabs blog[1].

WASI NN[2] is a proposal that allows WebAssembly guest modules running outside the browser to perform neural network inferencing by using host-provided implementations that can leverage CPU multi-threading, host optimizations, or hardware devices such as GPUs or TPUs. This article explores the goals of WASI NN, existing implementations, and details a new experimental implementation targeting ONNX, the Open Neural Network Exchange[3], which allows the usage of models built with PyTorch or TensorFlow from guest WebAssembly modules.

The implementation for ONNX runtimes with WASI NN can be found on GitHub[4].

The WASI Neural Network API is a new WebAssembly System Interface[5] proposal that allows guest WebAssembly modules running outside the browser, in WASI runtimes, access to highly optimized inferencing runtimes for machine learning workloads. Andrew Brown has an excellent article on the BytecodeAlliance blog about the motivations of WASI NN[6], but in short, the proposed API describes a way for guest modules to load a pre-built machine learning model, provide input tensors, and execute inferences on the highly optimized runtime provided by the WASI host. One of the most important things to note about the WASI NN API is that it is model agnostic, and so far quite simple:

- `load` a model using one or more opaque byte arrays
- `init_execution_context` and bind some tensors to it using `set_input`

[1]https://deislabs.io/posts/wasi-nn-onnx
[2]https://github.com/WebAssembly/wasi-nn
[3]https://onnx.ai/
[4]https://github.com/deislabs/wasi-nn-onnx
[5]https://wasi.dev/
[6]https://bytecodealliance.org/articles/using-wasi-nn-in-wasmtime

- `compute` the ML inference using the bound context
- retrieve the inference result tensors using `get_output`

As it is obvious from the API, there is no assumption around the way the neural network has been built – as long as the host implementation understands the opaque byte array as a neural network model, it can `load` it and perform inferences when `compute` is called using the input tensors. The first implementation for WASI NN in Wasmtime is for the OpenVINO™ platform[7], and Andrew Brown has another excellent article describing the implementation details[8]. This article explores how to add an implementation that performs inferences on the host for ONNX models.

ONNX, or the Open Neural Network Exchange[9], is an open format which defines a common set of machine learning operators and file format that ensure the interoperability between different frameworks (such as PyTorch, TensorFlow, or CNTK), with a common runtime and hardware access through ONNX runtimes. Two of the most popular machine learning frameworks, PyTorch and TensorFlow, have libraries that allow developers to convert built models to the ONNX format, then run them using an ONNX runtime. This means that by adding ONNX support to WASI NN, guest WebAssembly modules can perform inferences for both PyTorch and TensorFlow models converted to the common format – which makes it even easier to use a wider array of models from the ecosystem.

Because the WASI ecosystem is written in Rust, an ONNX implementation for WASI NN needs the underlying runtime to either be built in Rust, or have Rust bindings for its API – and this article describes building and using two such implementations for WASI NN, each presenting their own advantages and drawbacks that will be discussed later:

- one based on the native ONNX runtime[10], which uses community-built Rust bindings[11] to the runtime's C API.
- one based on the Tract crate[12], which is a native inference engine for running ONNX models, written in Rust.

Implementing WASI NN for a new runtime

Implementing WASI NN for a new runtime means providing an implementation for the WITX definitions of the API[13]. For example, the API used to load a new model is defined as follows:

```
(module $wasi_ephemeral_nn
  (import "memory" (memory))
```

[7]https://github.com/bytecodealliance/wasmtime/tree/main/crates/wasi-nn
[8]https://bytecodealliance.org/articles/implementing-wasi-nn-in-wasmtime
[9]https://onnx.ai/
[10]https://github.com/microsoft/onnxruntime
[11]https://github.com/nbigaouette/onnxruntime-rs
[12]https://github.com/sonos/tract
[13]https://github.com/WebAssembly/wasi-nn/blob/main/phases/ephemeral/witx/wasi_ephemeral_nn.witx

```
  (@interface func (export "load")
    (param $builder $graph_builder_array)
    (param $encoding $graph_encoding)
    (param $target $execution_target)

    (result $error (expected $graph (error $nn_errno)))
  )
)
```

Then, Wasmtime tooling can be used to generate Rust bindings and traits for the API that can then be implemented for a specific runtime:

```
pub trait WasiEphemeralNn {
    fn load(
        &mut self,
        builder: &GraphBuilderArray,
        encoding: GraphEncoding,
        target: ExecutionTarget,
    ) -> Result<Graph>;
}
```

> An article describing how to implement a new WebAssembly API from WITX for Wasmtime can be found here[14].

The two implementations (the one that uses the ONNX C API and the other using Tract) are fairly similar – they both implement the Rust trait defined by `WasiEphemeralNn`, which defines the following functions from the WASI NN API:

- `load` – this provides the actual model as opaque byte arrays, as well as the model encoding (ONNX for this implementation) and the execution target. This function has to store the model bytes so that guests can later instantiate it.
- `init_execution_context` instantiates an already loaded model – but because input tensors have not been provided yet, it only creates the environment necessary for the guest to `set_input`.
- `set_input` can be called multiple times, with the guest setting the input tensors and their shapes.
- `compute` is called by the guest once it has defined all input tensors, and it performs the actual inference using the optimized runtime.
- `get_output` is called by the guest once the runtime finished an inference, which then writes the `i-th` output tensor to a buffer the guest supplied.

During their lifetime, guests can perform any number of inferences, on any number of different neural network models. This is assured by the internal state of the runtime, which keeps track different concurrent requests (the specific

[14]https://radu-matei.com/blog/wasm-api-witx/

implementations can be found on GitHub[15]). The project also comes with a binary helper that mimics the Wasmtime CLI, with added support for both ONNX runtime implementations.

First, let's consider the official ONNX implementation[16] – it is built in C++, and provides a highly efficient runtime for inferencing. It comes with APIs for a number of different languages[17], including Python, Java, JavaScript, C#, or Objective-C, all of them through accessing the ONNX shared libraries (`.dll`, `.so`, or `.dylib`, depending on the operating system). To use them from Rust, the `onnxruntime-rs` crate[18] offers bindings to the underlying C API of the runtime, with a nice wrapper that makes using this API from Rust much easier than directly accessing the `unsafe` functionality exposed by the C API. Keep in mind, however, that this is not an official project, and currently targets a slightly older ONNX version, 1.6.

To execute the inference using the native ONNX runtime, first download the ONNX runtime 1.6 shared library[19] and unarchive it, then build the project (this is temporary, until proper release binaries are provided in the repository). At this point, the helper `wasmtime-onnx` binary can be used to execute WebAssembly modules that use WASI NN to perform inferences. The following examples use the integration tests from the project repository[20], and use the SqueezeNet[21] and MobileNetV2[22] models for image classification.

```
$ cargo build --release
$ LD_LIBRARY_PATH=<PATH-TO-ONNX>/onnx/onnxruntime-linux-x64-1.6.0/lib \
RUST_LOG=wasi_nn_onnx_wasmtime=info,wasmtime_onnx=info \
         ./target/release/wasmtime-onnx \
        tests/rust/target/wasm32-wasi/release/wasi-nn-rust.wasm \
        --dir tests/testdata \
        --invoke test_squeezenet

integration::inference_image:
results for image "tests/testdata/images/n04350905.jpg"
class=n04350905 suit of clothes (834); probability=0.21431354
class=n03763968 military uniform (652); probability=0.18545522

execution time: 74.3381ms with runtime: C
```

The previous command can be translated as follows: using the `wasmtime-onnx` binary, which provides a host implementation for performing ONNX inferences using the native ONNX runtime, start the WebAssembly module

[15]https://github.com/deislabs/wasi-nn-onnx/tree/main/crates/wasi-nn-onnx-wasmtime/src
[16]https://github.com/microsoft/onnxruntime
[17]https://www.onnxruntime.ai/docs/reference/api/
[18]https://github.com/nbigaouette/onnxruntime-rs
[19]https://github.com/microsoft/onnxruntime/releases/tag/v1.6.0
[20]https://github.com/deislabs/wasi-nn-onnx/blob/main/tests/rust/src/main.rs
[21]https://github.com/onnx/models/tree/master/vision/classification/squeezenet
[22]https://github.com/onnx/models/tree/master/vision/classification/mobilenet

`wasi-nn-rust.wasm` and invoke the `test_squeezenet` function, which loads the SqueezeNet model and performs an inference on a picture of Grace Hopper[23]. Because the project does not yet enable GPU usage for the native ONNX runtime (see issue #9[24]), the inference is performed on the CPU, and is multi-threaded by default. This means that after GPU is enabled, the inference time will be even lower. But this comes at the cost of ease of configuration – because this runtime uses bindings to the ONNX runtime's C API, the shared libraries first have to be downloaded and configured (same for the GPU support, where in addition to the proper ONNX release with GPU support, the graphics drivers will also have to be properly configured).

This is the main reason a second implementation is provided here – Tract[25] is an ONNX runtime implemented purely in Rust, and does not need any shared libraries. However, it only passes *successfully about 85% of ONNX backend tests*, it does not implement internal multi-threading or GPU access, and the inference times on the CPU are slightly higher than for the native ONNX runtime. The same binary, ONNX model, and WebAssembly module can be used to run the same inference – the only difference is passing the `--tract` flag, informing the runtime to use the alternative implementation for ONNX:

```
$ RUST_LOG=wasi_nn_onnx_wasmtime=info,wasmtime_onnx=info \
         ./target/release/wasmtime-onnx \
        tests/rust/target/wasm32-wasi/release/wasi-nn-rust.wasm \
        --dir tests/testdata \
        --invoke test_squeezenet \
        --tract

integration::inference_image:
results for image "tests/testdata/images/n04350905.jpg"
class=n04350905 suit of clothes (834); probability=0.21431345
class=n03763968 military uniform (652); probability=0.18545584

execution time: 90.6102ms with runtime: Tract
```

> Note that `LD_LIBRARY_PATH` is omitted in this example, but it still has to be passed for now, depending on whether support for both implementations has been compiled. In future releases, compile-time flags and features will choose between the two, and the flag will no longer need to be passed when only the Tract runtime has been compiled. Also see issues #11[26] and #16[27].

The relative performance between the two can be seen in the inference times, and in most cases, the Tract runtime will yield slightly higher latency on the same hardware – but the fact that it comes with no runtime dependencies means that,

[23] https://en.wikipedia.org/wiki/Grace_Hopper
[24] https://github.com/deislabs/wasi-nn-onnx/issues/9
[25] https://github.com/sonos/tract
[26] https://github.com/deislabs/wasi-nn-onnx/issues/11
[27] https://github.com/deislabs/wasi-nn-onnx/issues/16

for non-critical scenarios, or when running on CPU-only machines, configuring and running this project becomes significantly easier (i.e. downloading a single binary).

Initial *relative* performance

A few notes on performance:

- this represents *very early* data, based on a limited number of runs and models, and should only be interpreted in terms of the relative performance that can be expected between running the same inference natively, through WASI NN, or purely in WebAssembly.
- the ONNX runtime is running multi-threaded on the CPU *only*, as the GPU is not yet enabled.
- in each case, all tests are executing the same ONNX model on the same images.
- all WebAssembly modules (both those built with WASI NN and the ones running pure Wasm) are run with Wasmtime v0.28[28], with caching enabled, and no other special optimizations. For the WebAssembly examples, Wasm module instantiation time on the tested hardware accounts for around 16 ms on average, so in reality, the actual inference time is very close to native performance.
- there are known limitations in both runtimes that, when fixed, should also significantly improve the inference performance.
- pre- and post-processing of input and output tensors still takes place in WebAssembly, so as runtimes, compilers, and libraries add support for SIMD[29], this should also be improved.

The following charts represent the total inference times for running the SqueezeNet and MobileNetV2 models on CPU-only hardware, natively, with WASI NN, and then purely in WebAssembly.

Being a much smaller module, the inference times for SqueezeNet are smaller relative to MobileNetV2, but the relative performance difference can still be observed:

As more modules and GPU support are added, the performance benchmarks will be updated, but there is a trend expected to be seen – regardless of the neural network used, the native ONNX runtime should be faster (or much faster, when GPU support is enabled) than the Tract runtime, which in turn is around 3 to 4 times faster than running purely in WebAssembly, with both WASI NN implementations slightly slower than their natively run counterparts – the difference being mainly because of the module instantiation times.

[28]https://github.com/bytecodealliance/wasmtime/releases/tag/v0.28.0
[29]https://github.com/WebAssembly/simd/blob/main/proposals/simd/SIMD.md

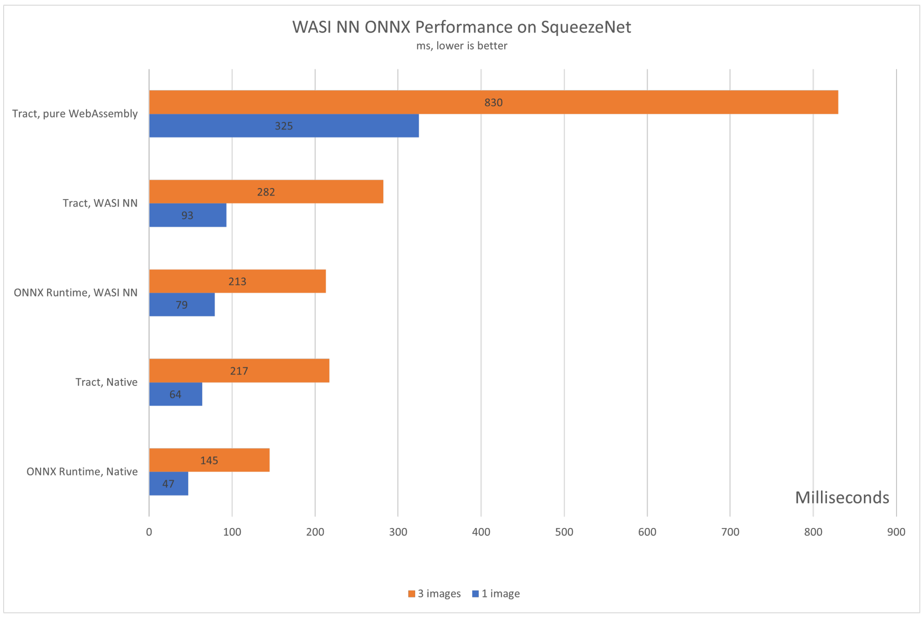

Figure 1: WASI NN SqueezeNet performance

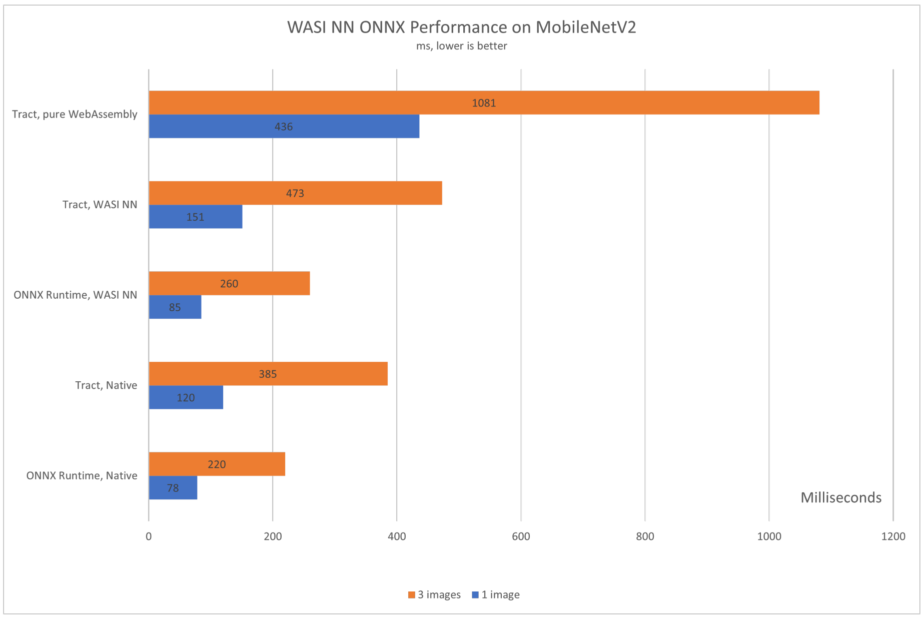

Figure 2: WASI NN MobileNetV2 performance

Writing WebAssembly modules, pre- and post-processing

Building WebAssembly modules that use WASI NN has to be done by using the client bindings – in this case, a slightly modified version that includes the ONNX variant for the graph encoding `enum`[30]. This API is still very early, and requires the use of `unsafe` Rust in quite a few places, but future releases should provide a much safer API that will use `ndarray`[31].

The one thing that is always required when performing inference on a pre-built neural network is understanding how input data has to be pre-processed when generating the input tensors, and how to interpret the output tensors – in the case of SqueezeNet and MobileNetV2, *the images have to be loaded in to a range of [0, 1] and then normalized using `mean = [0.485, 0.456, 0.406]` and `std = [0.229, 0.224, 0.225]`.* After the inference, *post-processing involves calculating the `softmax` probability scores for each class and sorting them to report the most probable classes.* This is, of course, dependent on how each neural network is built, and should be understood before trying to perform inferences, and usually, the ONNX models repository[32] provides enough information and Python implementations on how to perform pre- and post-processing, which can be adapted into the language used to build the Wasm module. For example, let's explore how to pre-process images for the two models (more Rust examples can be found in the Tract repository[33]):

```rust
pub fn image_to_tensor<
  S: Into<String> +
  AsRef<std::path::Path> + Debug
>(
    path: S,
    height: u32,
    width: u32,
) -> Result<Vec<u8>, Error> {
    let image = image::imageops::resize(
        &image::open(path)?,
        width,
        height,
        ::image::imageops::FilterType::Triangle,
    );


    let mut array = ndarray::Array::from_shape_fn(
      (1, 3, 224, 224),
      |(_, c, j, i)| {
        let pixel = image.get_pixel(i as u32, j as u32);
        let channels = pixel.channels();

        // range [0, 255] -> range [0, 1]
```

[30]https://github.com/radu-matei/wasi-nn-guest/tree/onnx
[31]https://docs.rs/ndarray/0.15.3/ndarray/
[32]https://github.com/onnx/models/
[33]https://github.com/sonos/tract/tree/main/examples

```
        (channels[c] as f32) / 255.0
    });

    // Normalize channels to
    // mean and standard deviation on each channel.
    let mean = [0.485, 0.456, 0.406];
    let std = [0.229, 0.224, 0.225];
    for c in 0..3 {
        let mut channel_array = array.slice_mut(s![0, c, .., ..]);
        channel_array -= mean[c];
        channel_array /= std[c];
    }

    Ok(f32_vec_to_bytes(array.as_slice().unwrap().to_vec()))
}
```

This is the Rust implementation for the pre-processing steps described above – load the image, resize it to 224 x 224, then scale each pixel value and normalize the resulting tensor. The part worth exploring in more detail is the final `f32_vec_to_bytes` function – up until the last line of the function, the image had been transformed into an `ndarray::Array` (which, for people used to data science in Python, should be very similar to `numpy.ndarray`). The last line has to transform the `Array` first into a uni-dimensional `f32` array, then into a bytes array, since this is how the WASI API transfers data. Then, it's the runtime's responsibility to recreate the tensor properly using its desired data type, shapes and dimensions.

Ideally, future releases of the bindings will allow guest modules to simply pass an `ndarray::Array`, and perform the transformation automatically, based on the shape and data type.

Current limitations

The following represents a non-exhaustive list of known limitations of the implementation. Depending on when this article is read, some of them might be already resolved, and others discovered or introduced:

- only FP32 tensor types are currently supported (#20[34]) – this is related to with the way state is tracked internally. It should not affect popular models (such as computer vision scenarios), but it represents the main limitation for now.
- GPU execution is not yet enabled in the native ONNX runtime (#9[35]) – the C headers for the GPU API have to be used when generating the bindings for the ONNX runtime.

If you are interested in contributing, around performance, GPU support, or

[34]https://github.com/deislabs/wasi-nn-onnx/issues/20
[35]https://github.com/deislabs/wasi-nn-onnx/issues/9

compatibility, please visit the repository and issue queue[36] for an updated list of open issues.

Conclusion

This article explores the WASI NN proposal and describes how the new implementation for ONNX is built, how it works, and how to execute such a runtime with Wasmtime, with a few notes around pre- and post-processing for input tensors when building guest WebAssembly modules in Rust.

Big shout-out to Andrew Brown for his work on WASI NN[37], to Jiaxiao Zhou[38] for helping out with the ONNX implementation, to Nicolas Bigaouette[39] for his work on the ONNX Rust bindings, and to all the people at Sonos building Tract[40] and the ONNX maintainers[41].

This project expands the possibility for running real-world neural networks from WebAssembly runtimes by providing near-native performance for executing inferences for PyTorch or TensorFlow models through ONNX. This implementation, as well as WASI NN, are still very early, but results for both ONNX runtimes and OpenVINO™ are promising, particularly when combining this type of workload with outbound networking support[42], or using cloud services from WebAssembly modules[43].

[36]https://github.com/deislabs/wasi-nn-onnx/issues
[37]https://bytecodealliance.org/articles/using-wasi-nn-in-wasmtime
[38]https://twitter.com/jiaxiao_zhou
[39]https://github.com/nbigaouette
[40]https://github.com/sonos/tract
[41]https://github.com/microsoft/onnxruntime
[42]https://radu-matei.com/blog/wasi-experimental-http/
[43]https://radu-matei.com/blog/using-azure-services-wasi/