TensorFlow inferencing using WebAssembly and WASI

Radu Matei

October 18, 2020

As edge devices become more powerful, being able to perform inferencing on trained neural network models without recompiling application components for each architecture becomes important, and WebAssembly could serve as the portable compilation target for such scenarios, running both in and outside browser environments.

In this article, we experiment with building a Rust program that performs image classification using the MobileNet V2 TensorFlow model, compile it to WebAssembly, and instantiate the module using two WebAssembly runtimes that use the WebAssembly System Interface (WASI), the native NodeJS WASI runtime, and Wasmtime. A special is given to writing model and image data into the module's linear memory, with implementations in both JavaScript and Rust. Finally, a simple prediction API is exemplified running on top of the Wasmtime runtime, and some limitations of this approach are discussed.

> The completed project can be found on GitHub.

TensorFlow, Rust, and WebAssembly

While there are still limitations to compiling some crates to WebAssembly, Rust is a programming language with excellent support for Wasm. Additionally, there is a Rust library that focuses on performing neural network model inferencing from Rust - `tract`, from Sonos - and while the crate is *very far from supporting any arbitrary [TensorFlow] model*, it can be used to run non-trivial models, and the project has an excellent quick start example that shows how to perform image classifications using the MobileNet V2 model, which will be used as a starting point for our Wasm module:

```
let model = tract_tensorflow::tensorflow()
    // load the model
    .model_for_path("mobilenet_v2_1.4_224_frozen.pb")?
    // specify input type and shape
    .with_input_fact(
        0,
```

```
        InferenceFact::dt_shape(
            f32::datum_type(),
            tvec!(1, 224, 224, 3)
        ),
    )
    // make the model runnable and fix its inputs and outputs
    .into_runnable()?;

// open image, resize it and make a Tensor out of it
let image = image::open("grace_hopper.jpg")?.to_rgb();
let image: Tensor = from_shape_fn(
    (1, 224, 224, 3),
    |(_, y, x, c)| {
        resized[(x as _, y as _)][c] as f32 / 255.0
    })
    .into();

// run the model on the input
let result = model.run(tvec!(image))?;
```

Source code adapted from the Sonos Tract examples.

The program above loads the TensorFlow model from a file, opens and resizes the target image to a resolution of 224 x 224 (which is the resolution of the training images for the MobileNet model), runs the model, and prints the class of the best prediction. Using Rust's `wasm32-wasi` compilation target, the project compiles to WebAssembly successfully, but executing it with Wasmtime (or any other runtime) results in a panic:

```
$ wasmtime run tf-example.wasm --dir=.
'The global thread pool has not been initialized.:
ThreadPoolBuildError { kind: IOError(Custom
{ error: "operation not supported on this platform" })}'

Caused by:
    0: failed to invoke `_start`
        ...
         8:  rayon_core::registry::global_registry
         9:  rayon_core::current_num_threads
         12: jpeg_decoder::decoder::Decoder<R>::decode_internal
         13: image::jpeg::decoder::JpegDecoder<R> as read_image
         16: image::io::free_functions::open_impl
         ...
```

The runtime error is caused by the `image` crate attempting to use multiple threads when loading the picture, and since there is no threads support in WebAssembly (here is an early proposal), the program panics. Fortunately, turning off all crate features except JPEG loading solves the problem:

```
image = { ... default-features = false, features = ["jpeg"] }
```

Because the program assumes both the model and image are in the current directory, we can use Wasmtime's `--dir` flag to grant the module permission to the current directory, and the program classifies the image as `military uniform` with a confidence of 32% (654 is the index of the `military uniform` label in the labels file of the model, and the image is that of Grace Hopper in uniform):

```
$ wasmtime run tf-example.wasm --dir=.
result: Some((0.32560226, 654))
```

However, getting the model and image from disk on every inference is not ideal, since I/O operations can be costly. Additionally, we might need classify images that are not on disk, but received by the runtime in some other ways (such as HTTP requests). In short, we need to pass both model and image data from the runtime to the module, using WebAssembly memory.

Using WebAssembly memory

Memory in WebAssembly is represented as a contiguous vector of uninterpreted bytes, with the memory size being a multiple of 64Ki (the length of one *memory page*). Lin Clark has an excellent explainer with code cartoons about WebAssembly memory, and in short, we will use it to pass arbitrary data between the Wasm runtime and guest modules.

Because the WebAssembly module is ultimately responsible for managing its own linear memories, it must export functionality to allocate memory, which the underlying host runtime can write into, and read from. In most cases, when using code generation libraries such as `wasm-bindgen`, this is handled by the library - but to better understand how everything works together, it is worth building our module without `wasm-bindgen`.

Recall the task at hand - pass model and image data from the runtime into the Wasm module's memory. Because both model and image data can be represented as arrays of 8-bit unsigned integers, we can write a single function, `alloc`, which allocates memory for a new `Vec<u8>` with capacity `len`. Before returning, the function calls `mem::forget` to take ownership of the memory block and ensure the vector's destructor is not called when the object goes out of scope. Finally, the function returns the pointer to the start of the memory block.

```rust
/// Allocate memory into the module's linear memory
/// and return the offset to the start of the block.
#[no_mangle]
pub extern "C" fn alloc(len: usize) -> *mut u8 {
    let mut buf = Vec::with_capacity(len);
    let ptr = buf.as_mut_ptr();

    std::mem::forget(buf);
```

```
    return ptr;
}
```

At this point, it is worth understanding how the WebAssembly module we are writing should expect to get the pointers and length of the model and image data - specifically, for each of the two input objects (model and image), the module expects a pointer (offset relative to the start of its entire linear memory) and the length of the object. Then, it uses `Vec::from_raw_parts` to create a `Vec<u8>` with the respective length and capacity (equal to the length) for each of the two objects:

```
/// This is the module's entry point for executing inferences.
/// It takes as arguments pointers to the start of the module's
/// memory blocks where the model and the image were copied,
/// as well as their lengths, meaning that callers of this
/// function must allocate memory for both the model and image
/// data using the `alloc` function, then copy it into the
/// module's linear memory at the pointers returned by `alloc`.
///
/// It retrieves the contents of the model and image, then calls
/// the `infer` function, which performs the prediction.
#[no_mangle]
pub unsafe extern "C" fn infer_from_ptrs(
    model_ptr: *mut u8,
    model_len: usize,
    img_ptr: *mut u8,
    img_len: usize,
) -> i32 {
    let model_bytes = Vec::from_raw_parts(
        model_ptr,
        model_len,
        model_len
    );
    let img_bytes = Vec::from_raw_parts(
        img_ptr,
        img_len,
        img_len
    );

    return infer(&model_bytes, &img_bytes);
}
```

As the comment suggests, host runtimes will have to call the module's `alloc` function for each of the two input objects, get the respective pointers returned by `alloc`, and copy the model and image data into the module's linear memory. Then, use the pointers and lengths to call `infer_from_ptrs`, which acts as the entrypoint for our module.

Finally, the only thing left to implement is the actual inference, which is similar to the example we saw earlier, with the only difference that it now takes the model and image data as arguments, rather than reading them from the file system (hence the slight changes in loading them):

```
/// Perform the inference given the contents of the
/// model and the image, and return the index of the
/// predicted class.
fn infer(model_bytes: &[u8], image_bytes: &[u8]) -> i32 {
    let mut model = std::io::Cursor::new(model_bytes);
    let model = tract_tensorflow::tensorflow()
        .model_for_read(&mut model)
    ...
    let image = image::load_from_memory(image_bytes);
    ...
    let result = model.run(tvec!(image));
    ...
}
```

We can now compile the code to the Rust `wasm32-wasi` target and get a WebAssembly module that can be instantiated and executed in any WASI compatible runtime, and in the following section we will explore running it in Node's WASI runtime and Wasmtime.

> The complete implementation can be found on GitHub.

Testing the module from Node's WASI runtime

This section will not focus on instantiating the module, but rather on allocating and copying the model and image data into the module's memory, as well as on invoking the inferencing function. For a guide on getting started with the NodeJS WASI runtime, you can check the article I wrote in July.

First, we need a helper method that copies a byte array into the module's memory. The function takes as arguments the actual bytes to copy and an instance of the module, calls the module's exported `alloc` function, and copies the bytes into the module's memory, starting at `ptr`:

```
// write `bytes` into the memory of `instance`
function writeGuestMemory(bytes, instance) {
  var len = bytes.byteLength;
  // call the module's `alloc` function
  var ptr = instance.exports.alloc(len);
  // create an array of length `len` in the module's memory,
  // starting at offset `ptr`
  var m = new Uint8Array(instance.exports.memory.buffer, ptr, len);
  // set the value of the array to `bytes`
  m.set(new Uint8Array(bytes.buffer));
```

```
  // return the offset
  return ptr;
}
```

Now that we have a way of writing data into the module's memory, we can use it to finally get predictions on images. The function below takes as parameters the contents of the MobilNet V2 model, the contents of an image, and a WebAssembly instance, calls the `writeGuestMemory` function defined above for each of the two objects we want to write into memory, then invokes the module's `infer_from_ptrs` exported function using the pointers to the objects and their lengths as parameters. The return value of the `infer_from_ptrs` function is the index of the predicted class - we use it together with the model's labels file in order to get a human-friendly description of the prediction:

```
// get a prediction given the MobileNet V2 model,
// an image, and an instance of the Wasm module
// that performs the inference.
function getPrediction(model_bytes, img_bytes, instance) {
  // write the contents of the model and image
  // in the memory of the module, using the
  // module's `alloc`exported function
  var mptr = writeGuestMemory(model_bytes, instance);
  var iptr = writeGuestMemory(img_bytes, instance);

  // invoke the module's exported `infer_from_ptrs`
  // function using the pointers and lengths of
  // the model and image, which returns an integer
  //  representing the index of the predicted class.
  let pred = instance.exports.infer_from_ptrs(
    mptr,
    model_bytes.length,
    iptr,
    model_bytes.length
  );

  // helper function to read the prediction
  // label from the index of the class
  return getLabel(pred);
}
```

The GitHub repository of this project contains the complete logic for instantiating the module, a compiled version of the WebAssembly module, the MobilNet V2 model, as well as a directory with a couple of images we can test:

```
$ node --experimental-wasi-unstable-preview1 \
       --experimental-wasm-bigint \
       test.js
predicting on file  golden-retriever.jpeg
```

```
inference time:  832 ms
prediction:  golden retriever

predicting on file  husky.jpeg
inference time:  541 ms
prediction:  Eskimo dog, husky
```

The program correctly uses the MobileNet V2 computer vision model and the WebAssembly module we wrote, and performs image classification on the test images.

Building a Rust host runtime with Wasmtime

When first experimenting with compiling the module, we used the Wasmtime CLI to quickly execute the module. This works great if the module needs to access files, for example, but we now need to write into the module's memory, and we cannot achieve that using the CLI - we now have to use the Wasmtime's excellent API. First, we need to implement a Rust function which, given a byte array and an instance of the WebAssembly module, writes the data into the module's memory - essentially the same functionality we implemented in JavaScript in the previous section:

```rust
/// Write a byte array into the instance's linear memory
/// and return the offset relative to the module's memory.
fn write_guest_memory(
    bytes: &Vec<u8>,
    instance: &Instance
) -> Result<isize, anyhow::Error> {
    // Get the "memory" export of the module.
    // If the module does not export it, just panic, since
    // we are not going to be able to copy any data.
    let memory = instance
        .get_memory(MEMORY)
        .expect("expected memory not found");

    // Get the guest's exported `alloc` function, and call
    // it with the length of the byte array.
    let alloc = instance
        .get_func(ALLOC_FN)
        .expect("expected alloc function not found");
    let alloc_result = alloc.call(
        &vec![Val::from(bytes.len() as i32)]
    )?;

    // The result is an offset relative to the module's
    // linear memory, which is used to copy the bytes into
    // the module's memory.
```

```
    let guest_ptr_offset = match alloc_result
        .get(0)
        .expect("expected the result to have one value")
    {
        Val::I32(val) => *val as isize,
        _ => return Err("guest pointer must be Val::I32"),
    };

    // Copy the desired bytes into the memory at `guest_ptr_offset`.
    unsafe {
        let raw = memory.data_ptr().offset(guest_ptr_offset);
        raw.copy_from(bytes.as_ptr(), bytes.len());
    }
    return Ok(guest_ptr_offset);
}
```

We can now use the `write_guest_memory` function to implement getting a prediction - given the contents of the model and an image, create a new WebAssembly instance of the module using the Wasmtime API, write the contents of the model and image into the guest's memory, then invoke the `invoke_from_ptrs` function from the WebAssembly module. Finally, use a simple helper function that gets the index of the predicted class and return the human-friendly label of the class:

```
/// Get a prediction given the MobileNet V2 model and an image.
fn get_prediction(
    model_bytes: Vec<u8>,
    img_bytes: Vec<u8>
) -> Result<String, anyhow::Error> {
    // Unfortunately, we have to create a new module
    // instance for every prediction, since a
    // `Wasmtime::Instance` cannot be safely sent between threads.
    // See https://github.com/bytecodealliance/wasmtime/issues/793
    let instance = create_instance(WASM.to_string())?;

    // Write the MobileNet model and the image contents to
    // the module's linear memory, and get their pointers.
    let model_bytes_ptr = write_guest_memory(
        &model_bytes,
        &instance
    )?;
    let img_bytes_ptr = write_guest_memory(
        &img_bytes,
        &instance
    )?;

    // Get the module's "infer_from_ptrs" function,
    // which is the entrypoint for our module.
```

```
    let infer = instance
        .get_func(INFER_FN)
        .expect("expected inference function not found");

    // Call the inference function with the pointer
    // and length of the model contents and image.
    let results = infer.call(&vec![
        Val::from(model_bytes_ptr as i32),
        Val::from(model_bytes.len() as i32),
        Val::from(img_bytes_ptr as i32),
        Val::from(img_bytes.len() as i32),
    ])?;

    // The inference function has one return argument,
    // the index of the predicted class.
    match results
        .get(0)
        .expect("expected the result to have one value")
    {
        Val::I32(val) => {
            let label = get_label(*val as usize)?;
            return Ok(label);
        }
        _ => return Err("cannot get prediction"),
    }
}
```

As the comment suggests, a `Wasmtime::Instance` cannot be sent across threads (not in a memory-safe manner) - which means we cannot use the same instance in a multi-threaded environment (for example when responding to HTTP requests). The impact of this is that we pay the price of instantiating the module every time we make an inference (which, for this specific WebAssembly module, and tested on my hardware, can be around 600-800 milliseconds).

The two implementations of executing predictions (in JavaScript and Rust) are very similar - we are essentially performing the same operations: create instance, write data into memory, invoke inference function.

> The complete implementation can be found on GitHub.

Creating a simple prediction API

Now that we are able to perform predictions using the module we built with Wasmtime, we can easily expose this functionality over an HTTP API, using the Rust `hyper` crate.

Running `cargo run --release` at the root of the GitHub repository of this project, an HTTP server is started on port 3000 which expects an image URL as

request body, downloads the image contents, gets a prediction using the method described above, then returns its human-friendly label:

```
$ cargo run --release
Listening on http://127.0.0.1:3000

module instantiation time: 774.715145ms
inference time: 723.531083ms
```

In another terminal instance (or from an HTTP request builder, such as Postman):

```
$ curl --request GET 'localhost:3000' \
--header 'Content-Type: text/plain' \
--data-raw 'https://<url-to-a-retriever-puppy>.jpg'
golden retriever
```

For each request, the module instantiation and inference time are printed to the console, and this is where not being able to share a `Wasmtime::Instance` across threads affects the overall response time of the API we built. But as WebAssembly and WASI mature, hardware devices (such as GPUs) will be available in WebAssembly, and as proposals such as WASI-NN are implemented, we can expect the both the instantiation and inference times to decrease.

In this article we experimented with compiling a Rust program that performs inferencing on a pre-trained neural network to WebAssembly, and executed it in NodeJS and Wasmtime, while closely exploring how the data is shared between the host and guest using WebAssembly memory.

FAQ

- Should I use this in production? Probably not, as the code presented here is experimental, using an approach mostly suited for educational purposes.
- How does this compare to the TensorFlow.js WebAssembly backend? The Wasm backend for TensorFlow.js can only be used from JavaScript environments, browsers and NodeJS. This means it can't be used with other WebAssembly runtimes. Check the official documentation for running MobileNet in the browser, using the WebAssembly backend.
- How easily can I use my own model? While the *approach* used by this project can be used to execute inferences using different neural network models, the implementation is specialized for using the MobileNet V2 model, and changing the model architecture or its inputs would require changes in the WebAssembly module and its instantiation.