McGill University

Montréal

School of Computer Science

Machine Learning in WebAssembly

Author:
Amir El Bawab

Supervisor:
Prof. Clark Verbrugge

September 16, 2019

A THESIS SUBMITTED TO MCGILL UNIVERSITY
IN PARTIAL FULFILLMENT OF THE REQUIREMENTS OF THE DEGREE OF
MASTER OF SCIENCE

Abstract

Web-based, client-side, compute-heavy applications like machine-learning and visual processing are limited by the relatively low performance offered by traditional JavaScript implementations and awkward interface to GPU acceleration. WebAssembly (Wasm) has been proposed as a low-level alternative, promising significant performance benefits and easier optimization potential. Most applications, however, are machine-ported from existing languages, such as C/C++ or Rust, limiting performance through standard library implementations and generic interfaces. In this research we explore improvements to Wasm that facilitate numeric computation. We extend the Google V8 engine with custom Wasm instructions in an attempt to speed up the baseline compiler (Liftoff), and improve built-in function call overhead for code transitioning between Wasm and JavaScript, as well as integrating Wasm more directly with native C/C++. Our design is aimed at understanding optimization opportunities, with the goal of defining high performance Machine Learning models written entirely in WebAssembly.

Résumé

Les applications côté client, basées sur le Web, lourdes en calcul telles que l'apprentissage automatique et le traitement visuel sont limitées par les performances relativement faibles offertes par les implémentations JavaScript traditionnelles et l'interface étrange d'accélération GPU. WebAssembly (Wasm) a été proposé comme alternatif de bas niveau, promettant des avantages significatifs en termes de performances et un potentiel facile d'optimisation. Cependant, la plupart des applications sont portées par des machines à partir de langages existants tels que C/C++ ou Rust, limitant les performances à travers des implémentations de bibliothèques standards et des interfaces génériques. Dans cette recherche, nous explorons les améliorations apportées au Wasm qui facilitent le calcul numérique. Nous étendons le moteur V8 de Google avec des instructions Wasm personnalisées afin d'accélérer le compilateur de base (Liftoff), et améliorer les frais généraux d'appel des fonctions intégrées pour la transition du code entre Wasm et JavaScript, ainsi que d'intégrer Wasm plus directement avec les C/C++ fonctions natives. Notre conception vise à comprendre les opportunités d'optimisation, dans le but de définir des modèles d'apprentissage automatique de haute performance entièrement écrite en WebAssembly.

Acknowledgements

I would like to express my sincere gratitude to my supervisor, Professor Clark Verbrugge. My work and contribution would not have been possible without his support, advice, lessons and time.

I would like to thank Professor Laurie Hendren, although no longer with us, for offering me her guidance and tips during our lab meetings.

I would like to thank NSERC and the COHESA research network for their funding support during my studies.

Finally, I would like to thank my friends, especially my lab mates, for the great time we spent inside and outside campus, and my family for their support and encouragement.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

In this thesis, we study and research the execution of various machine learning tasks using WebAssembly. In section 1.1 we present the motivation behind our research. In section 1.2 we list and briefly describe our contributions. In section 1.3 we present an overview of the chapters composing this thesis.

1.1 Motivation

Since its release, JavaScript has been the dominant language of the web. The language proved to be simple and convenient to use. A new set of features suddenly became available to developers from animation to dynamically populating page content. As time passed by, developers and software engineers have drawn the line as to what kind of applications should run on the web and which ones should not. Delivering an application in form of a web page makes it portable and available to a wide audience. However, a critical drawback preventing a large set of application to move from a native environment to the web is the noticeable drop in the execution speed. JavaScript code shipped in a web page is not executed right away. It first needs to be parsed, converted into an intermediate representation and then interpreted or compiled for execution. A dynamically typed language, such as JavaScript, makes this pipeline even more expensive because a web engine cannot completely optimize a function without knowing the types of its variables ahead of time. Thus, an engine might decide to apply optimizations for "hot" functions at a later stage of the code execution. This approach works well for the current usage of the language, however it limits the scope of applications that can practically run on the web. For instance, programs that are computationally heavy, such as machine learning, would suffer if a web engine fails to properly optimize the code. Researchers from the big four browser companies (Mozilla, Google, Microsoft and Apple) realized the opportunities that a typed language could bring to the web, thus in 2017 [3]

they announced their work for supporting WebAssembly as a new candidate for writing code on the web. The potential performance benefits promised by this new technology, motivated our study to explore the optimization opportunities that WebAssembly can bring to machine learning tasks on the web. Our work in this thesis aims at providing analysis and strategies for applying and optimizing WebAssembly in the context of machine learning applications. Existing machine learning libraries written in JavaScript can benefit from our study to explore some of the advantages and challenges that this language presents for training models and predicting results on the web.

Training and inference tasks in machine learning are computationally heavy and require a particular setup in order to execute efficiently. Today, a common method for using this technology is installing a machine learning library natively and interacting with it using a programming language such as Python. To extend the accessibility of machine learning and expand its use cases, web-based libraries have been offering an alternative option for users to apply this technology on their browser. However, the performance constraints of the existing technologies supported by the browser made this opportunity impractical. With the appearance of WebAssembly, we explore a new option for executing machine learning tasks more efficiently on the web.

1.2 Contribution

Our strategy for achieving our research goals begins by exploring the language capabilities and architecture. We familiarize ourselves with WebAssembly by developing an application allowing developers to inspect the low-level bytecode composing the language. This tool offers debugging and profiling features to programs written in WebAssembly. Moreover, it presents a terminal-based user interface for displaying module components such as the *stack machine* and the *linear memory*. Working on this project strengthened our understanding of WebAssembly and facilitated our later contributions.

After exploring the language at the bytecode level, we experiment with introducing custom instructions into WebAssembly in an attempt to accelerate a simple machine learning application we coded manually. In total, we present four instructions operating on three different layers of the language. First layer is the machine code generated by *Liftoff*. Second layer is the *stack machine* of a Wasm program module. Third layer is the interface for importing JavaScript functions. Following the implementation of our custom instructions, we present an analysis evaluating the performance gained by integrating the new bytecode into our example program. Overall, our experiments with custom instructions on the *Liftoff* and the *stack machine* levels did not show significant performance improvement.

On the JavaScript interface level, we noticed an enhancement in the execution time using our custom instruction approach. However, this advantage was reduced when repeating our experiments on a newer version of V8, implementing an enhanced execution procedure for certain kinds of imported JavaScript functions.

Our next contribution explores the option of performing native calls to C++ functions from WebAssembly programs. Native calls are intended to measure and study the performance gain for offloading kernel machine learning operations to C++ functions compiled and optimized *ahead-of-time*. Although they can be beneficial to accelerate the execution time of programs, native functions can possibly introduce security threats in a web engine. Our study for native calls focuses on optimizing the execution time and considers security challenges as part of our future work. To analyze the performance gain obtained by using native calls, we experiment with existing options and introduce our own syntax aiming to eliminate external calls overhead. Our experiment demonstrates the relation between a machine learning model complexity and the performance gain obtained using our approach for calling native functions.

Our most significant contribution in this thesis is WasmDNN, a library for generating deep neural network (DNN) models in WebAssembly format. With the help of Wasm++, another library we built for simplifying the task of writing in WebAssembly, WasmDNN produces bytecode snippets that are manually optimized for various model configurations. In our discussion of WasmDNN, we elaborate on the features currently supported by the library and highlight our optimizations for the various operations involved in the different machine learning tasks. Most of our optimizations benefit from the *Single Instruction Multiple Data* (SIMD) feature which is currently a work in progress in the language. Popular machine learning libraries on the web are currently written in JavaScript, for the exception of *WebDNN* which uses *Emscripten* to compile generated C++ models into WebAssembly. However, WebDNN currently only supports executing models pre-trained by other libraries. Therefore, to the best of our knowledge, WasmDNN is the first library to directly generate WebAssembly bytecode for machine learning models, while supporting training, testing and predicting on the web. Our performance analysis results shows a clear advantage in the execution time of our machine learning models over other popular libraries for both training and inference tasks.

1.3 Thesis Overview

In this thesis, we present a total of 8 chapters. Chapter 1 (current chapter) introduces our research and contribution which are elaborated in this thesis. Chapter 2 informs the reader

about necessary background knowledge required in order to better comprehend at what level our study has been done. The next four chapters present our contributions that have been briefly explained in section 1.2: Chapter 3 elaborates on our WABT Debugger tool, Chapter 4 offers a detailed discussion about our custom instructions and presents a brief explanation for modifying the language to achieve various goals, Chapter 5 presents the native call feature allowing us to make calls to C++ functions from WebAssembly using a custom Wasm syntax, and Chapter 6 covers WasmDNN library, our major contribution in this thesis. In the same chapter, we also introduce Wasm++, another helper library we developed in order to better implement WasmDNN. Chapter 7 lists several related work on the various levels of our contributions. Finally, Chapter 8 concludes our thesis by summarizing our research and experimental results, in addition to proposing certain future work which could potentially be beneficial for extending our studies.

Chapter 2

Background

Part of our contribution described in this thesis requires the reader to be familiar with certain terms and concepts. As a prerequisite, knowing the pipeline and some basic components of a compiler design will give the reader a better idea at what level our contribution has been applied. This chapter aims at providing the reader with the necessary background details referenced in later chapters. In section 2.1, we start with an overview of the WebAssembly language and how it was able to smoothly integrate into existing technologies. In section 2.2, we present a low level view of a WebAssembly module. In section 2.3, we discuss some of the tools that helped us perform our experiments. In section 2.4, we list some of the popular web engines and elaborate on Google V8, the environment where we hosted our code. Finally, in section 2.5 we discuss the potential of machine learning on the web; we also present some of the available libraries and propose how WebAssembly technology can be beneficial in speeding up those libraries.

2.1 WebAssembly (Wasm)

Historically, JavaScript has been the only candidate programming language on web browsers, and over the years it proved to be simple to learn and convenient for interacting with web pages. However, because of its performance constraints, JavaScript can be a difficult target for application requiring complex computations. This becomes even more problematic when such applications are hosted in web containers on embedded devices with limited resources. To address the latter concern, the founders of the *WebAssembly* language, with the help of the online community, focused on designing a language that is safe, fast, portable and compact [3].

WebAssemby is a low-level bytecode language that is a portable and mainly targeted at client-side computation on the web. Although WebAssembly is defined as a bytecode

language, it can also be expressed in a human readable format known as *Wat*. This representation can be written in a syntax similar to the Lisp programming language, but can also be flattened into a set of consecutive instructions similar to a stack-based assembly language. Using an existing tool (section 2.3.1), Wat code can be validated and compiled into its equivalent Wasm binary format.

In addition to directly writing WebAssembly instructions, developers have already found several methods to compile existing C, C++ and Rust projects into WebAssembly, making the whole process simple and transparent for the user. For example, *Emscripten* [4], a project that originally compiled *LLVM* intermediate representation (LLVM IR) into *asm.js*, is now capable of converting its generated code into WebAssembly. LLVM [5] is a library that optimizes and generates a target code for an input represented in its own intermediate representation. Asm.js [6] is a strict subset of JavaScript and was a previous attempt at porting native projects to the web and accelerating them using dedicated optimizations. However, asm.js was still expressed in terms of JavaScript code and a web engine still had to spend a considerable amount of time loading, parsing and compiling it. More recently, LLVM has officially released [7] a back-end for WebAssembly, cutting the necessity to go through further intermediate targets.

2.2 WebAssembly Binary Explorer

Working and optimizing WebAssembly can be done on several levels. For instance, some optimization can be done by scanning the WebAssembly bytecode and rewriting it to make it more efficient (such as by peephole or traditional dataflow optimization). Other optimizations can be done on the engine side such as enhancing generated machine code or introducing new WebAssembly instructions. The last method requires knowledge of the binary format of the language. Later in this report, we present our attempts at such optimizations where the modification of the generated binary encoding was necessary. This said, most developers who are simply using the language to write their programs do not need to know the details of the Wasm binary format.

In this section we explore the binary format for a WebAssembly program. The code presented in listing 2.1 is the source written in Wat, and in listing 2.2 is its corresponding annotated binary format. The code represents a Wasm module importing from JavaScript a print function (`print_i32`) and defining a Wasm function exported to JavaScript as "`main`" which simply prints the number `42`. The annotated binary version of this code shows the details of how a web engine would decode a Wasm module. The first 4 bytes of the binary code represent the magic number, a value identifying a Wasm file, and the next 4 bytes

represent the version of the Wasm binary format. The remaining clusters of bytes represent the different sections of the Wasm module. A web engine can recognize a section by its unique identifier which marks its beginning. A common pattern used for decoding a section, function body, parameters and returns is prefixing them with a counter or the number of expected bytes. The latter prefix values are encoded into a Little Endian Base 128 format (LEB128) [8], allowing large integer values to be represented in a few bytes.

```
(module
  (type (func (param i32)))
  (import "env" "print_i32" (func (type 0)))
  (func
    (export "main")
    (call 0
      (i32.const 42)
    )
  )
)
```

Listing 2.1: WebAssembly program in Wat format

2.3 Tools

Several WebAssembly tools have been developed in the last few years. The two main repositories hosting those tools are *"The WebAssembly Binary Toolkit" (WABT)* [9] and *"Binaryen"* [10]. Both projects are hosted under the WebAssembly organization on Github, and together provide an extensive list of features and opportunities for working in the language. In section 2.3.1, we describe certain WABT tools which were useful throughout our research. In section 2.3.2, we describe a simplified description of the Emscripten pipeline, which uses a Binaryen tool to adapt its JavaScript output format to WebAssembly.

2.3.1 The WebAssembly Binary Toolkit

The WABT repository hosts several tools for using and manipulating WebAssembly files. This section describes the most important tools that were necessary for us to develop in WebAssembly, and contribute to the language. Section 2.3.1.1 explains `wat2wasm`, a tool for converting Wat code to Wasm binary. Section 2.3.1.2 elaborates on the architecture of `wasm-interp`, a tool for interpreting WebAssembly bytecode.

```
00 61 73 6D                     ;; magic number
01 00 00 00                     ;; version field
;; (type (func (param i32)))
01                              ;; id of type section
08                              ;; size: 8 bytes
02                              ;; types count
60                              ;; function type
01                              ;; param count
7F                              ;; i32
00                              ;; return count
60                              ;; function type
00                              ;; param count
00                              ;; return count
;; (import "env" "print_i32" (func (type 0)))
02                              ;; id of import section
11                              ;; size: 17 bytes
01                              ;; imports count
03                              ;; length of "env"
65 6E 76                        ;; module name "env"
09                              ;; length of "print_i32"
70 72 69 6E 74 5F 69 33 32      ;; field name "print_i32"
00                              ;; kind: external function
00                              ;; signature index (type 0)
;; (func)
03                              ;; id of function declaration section
02                              ;; size: 2 bytes
01                              ;; functions count
01                              ;; signature index (type 1)
;; (export "main")
07                              ;; id of export section
08                              ;; size: 8 bytes
01                              ;; export count
04                              ;; length of "main"
6D 61 69 6E                     ;; export name: "main"
00                              ;; kind: external function
01                              ;; function index
;; (call 0 (i32.const 42))
0A                              ;; id of code section
08                              ;; size: 8 bytes
01                              ;; functions count
06                              ;; body size
00                              ;; locals count
41 2A                           ;; (i32.const 42)
10 00                           ;; (call 0)
0b                              ;; end
```

Listing 2.2: Annotated WebAssembly program in binary format

2.3.1.1 `wat2wasm`

WebAssembly bytecode can be represented in a human readable format (Wat). The `wat2wasm` tool converts this representation into Wasm binary format which is detailed in the

WebAssembly design repository [8]. In listing 2.3 we show a simple Wat program for adding two integer numbers of size 32-bit, with its corresponding Wasm binary generated by this tool.

Wat format

```
(module
  (func $add
    (export "add")
    (param $l i32)
    (param $r i32)
    (result i32)
    (i32.add
      (get_local $l)
      (get_local $r)
    )
  )
)
```

Wasm binary

```
0061 736d 0100 0000 0107 0160 027f 7f01
7f03 0201 0007 0701 0361 6464 0000 0a09
0107 0020 0020 016a 0b
```

Listing 2.3: Wat file compiled using `wat2wasm` into its corresponding Wasm binary representation

2.3.1.2 `wasm-interp`

As the name of this program hints, this tool provides an interpreter and an environment for executing Wasm bytecode. A simplified version of the tool architecture is presented in figure 2.1. An Environment instance contains lists of components such as module objects, tables, globals, linear memories and other Wasm components. A Thread object offers Wasm interpretation capabilities. During execution, a Thread uses its Environment in order to get information about components in the Wasm module scope. An Executor is simply a wrapper of a Thread providing an interface to control the interpretation of Wasm functions. In chapter 3, we revisit this diagram to discuss our contribution which introduces a new feature in the toolkit.

2.3.2 Emscripten

Many native projects today are being ported to the web using Emscripten [11]. By compiling LLVM IR to JavaScript, Emscripten allowed developers to create projects using various LLVM frontends, such as C, C++ and Rust. Fastcomp [12] is the compiler core for Emscripten; it is implemented as an LLVM backend, and is responsible for generating JavaScript, or more specifically asm.js.

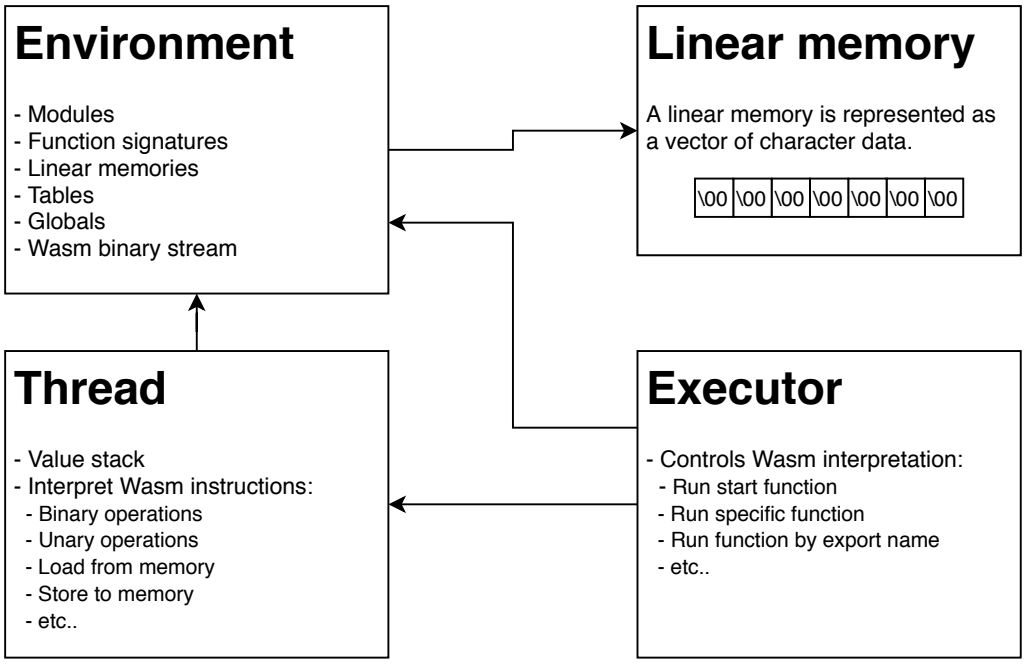

Figure 2.1: A simplified diagram of the architecture used in `wasm-interp`

When WebAssembly appeared, Emscripten was able to produce Wasm output from its generated asm.js with the help of `asm2wasm`, a tool hosted on the Binaryen repository. The latter tool reads the asm.js file generated by Emscripten, then converts it into a Wasm binary file. With this exciting technology, several companies started experimenting with the execution of their products on the web. For instance, OpenCV [13], an open source computer vision library, has already documented the process for using Emscripten to compile part of their library to WebAssembly [14]. Another example is Autodesk which ported their AutoCAD product on the web by compiling their source code to WebAssembly using Emscripten [15].

Figure 2.2 illustrates how Emscripten can be used to bridge the gap between the LLVM bytecode generated by the Clang C/C++ compiler, and Wasm bytecode produced by the `asm2wasm` tool. In March 2019, LLVM 8 [7] has officially brought WebAssembly out from its experimental state, allowing programmers to directly generate Wasm using the new LLVM backend without passing through the asm.js intermediate step.

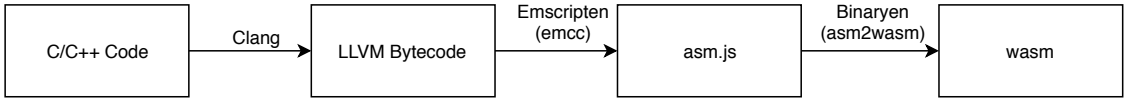

Figure 2.2: Pipeline for compiling C/C++ code to Wasm using Emscripten

2.4 JavaScript and WebAssembly Engines

WebAssembly is currently supported by four major browsers [16]: Firefox, Google Chrome, Safari and Microsoft Edge. WebAssembly is intended to allow browsers to perform fast computation when necessary, but in its current release, it is not sufficient to completely replace JavaScript which provide richer functionalities for interacting with a web page. In this section, we discuss the different environments for executing WebAssembly on the web. In sections 2.4.1 and 2.4.2 we provide a short presentation of Chakra engine used by Microsoft Edge, and SpiderMonkey engine used by Mozilla. In section 2.4.3 we go in deeper details for the selected testbed engine for our experiments, V8 by Google. Our choice for the latter, was based on the popularity of the engine. For instance, Google Chrome, an embedder of V8, leads the web browser market share with 63% (Figure 2.3). Other browsers such as Opera and Brave are also using V8 for processing JavaScript. In addition to its wide use in the web world, V8 also powers many host machine applications such as Node.js and the Electron framework. In December 2018, Microsoft Edge announced their adoption of the Chromium project into their browser [17], adding a new entry to the list of V8 embedders.

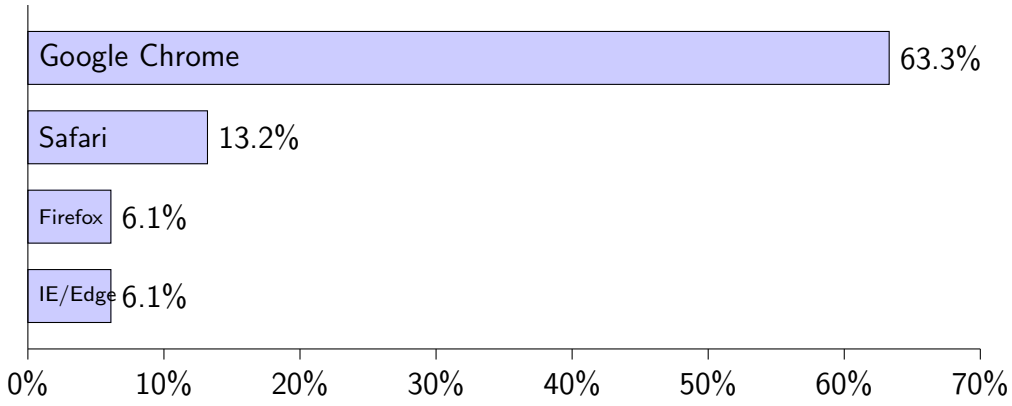

Figure 2.3: Web Browser Market Share as of March 2019 [1]

2.4.1 Chakra Engine

Chakra is the JavaScript engine used by the Microsoft browsers Internet Explorer and Edge. In their blog [17], Microsoft recently announced their shift from Chakra engine to Google V8. Even though Microsoft browser will be built on top of a different engine, other applications still depend on Chakra to execute such as Node.js on ChakraCore [18], a project allowing Node.js to utilize the core component of the Chakra engine instead of V8 in order to process JavaScript code. Chakra engine is currently one of the popular web engines supporting WebAssembly.

2.4.2 SpiderMonkey Engine

The SpiderMonkey JavaScript engine was originally written by Brendan Eich, the creator of the JavaScript programming language [19, 20]. Eich has written SpiderMonkey as the first engine that could execute his language. He later co-founded the Mozilla project which is currently maintaining his engine.

SpiderMonkey is another popular engine supporting WebAssembly. The engine has two tier compilers for the language [3]. The first tier is the baseline compiler which performs a single pass over the Wasm binary code and directly emits machine code. The baseline compiler omits any intermediate representation, but maintains the validation step which is also done in the single pass. The second tier, called IonMonkey, is a more advanced optimizing compiler using static single assignment (SSA) [21] form as an intermediate representation. The latter tier is also used by the engine to compile JavaScript.

In a blog post on the Mozilla website [22], the author mentioned that a new optimizing JIT compiler is currently under development and aims to give WebAssembly an improved performance compared to the currently used optimizing compiler.

2.4.3 V8 Engine

V8 is the JavaScript engine developed by Google and used in several browsers such as Chrome, Chromium, Opera and in the future Edge by Microsoft. In addition to web browsers, the V8 engine has been embedded into several offline projects. For example, Node.js embedder allows users to run a server with JavaScript on the backend. Another example is the Electron framework which uses V8, enabling opportunities to build applications such as Atom editor and Visual Studio Code.

Most of our contribution and proof of concepts have been implemented inside the V8 internals. We chose Node.js as a V8 embedder because it was easily accessible through a command-line interface, allowing us to remotely connect and continuously be able to implement features. V8 is a very large project and importing it into an IDE is tedious and involves significant setup complexity to resolve dependencies. Fortunately, Chrome is another embedder of V8 and has a code search tool online [23], capable of resolving most of the complex references and macros. We highly recommend using the latter for anyone interested in exploring or modifying the V8 code. The only challenge with using this search tool is alignment with the local version of V8. In our experience, this has not been difficult as while minor changes have been observed between the two versions of V8, the main impact is that an aligned source code facilitates code search.

In 2017, V8 launched an enhanced JavaScript implementation procedure including a new

optimizing JIT compiler called TurboFan [24]. The latter compiler aims to provide a better JIT optimization than its predecessor compiler Crankshaft [25]. In 2018, V8 introduced the Liftoff baseline compiler to exclusively speed up the start time for WebAssembly execution [2]. Liftoff works in a similar fashion as the tier one baseline compiler that we've described in the SpiderMonkey section (2.4.2). While Liftoff starts the non-optimized execution of the WebAssembly program, TurboFan begins to produce an optimized JIT version of the WebAssembly code in the background. In the remaining part of this section, we elaborate on the designs of TurboFan and Liftoff. We start by giving an overview of both compilers, then we highlight their differences and explain why both are necessary to achieve the best performance.

The diagram in figure 2.4 was presented in a V8 blog [2] to give a general comparison between TurboFan and Liftoff. As the diagram shows, the pipeline for TurboFan is a strict superset of the Liftoff one. The Liftoff compiler marks an important improvement in the compilation time of a WebAssembly code in V8. Because Liftoff does not focus on optimization, the process of compiling a WebAssembly program consists of a single pass over the Wasm binary code where type checking is applied and machine code is emitted. On the web, compilation time is crucial for the user experience, as any additional compilation time is perceived as additional page-loading latency or as sluggish browser rendering. In general, major browsers have been compiling JavaScript code fairly quickly for regular websites. However, WebAssembly technology aims to execute more complex programs on the browser consisting of relatively large binary files. Compiling and optimizing large programs before execution could result in a significant idle time for the user. Before creating Liftoff, the latter experience was the case because TurboFan was the only candidate for compiling WebAssembly.

Compared to Liftoff, TurboFan follows a more advanced pipeline for compiling WebAssembly to machine code. Similar to Liftoff, the compiler starts by decoding the Wasm bytecode. However, after validating the instructions TurboFan starts constructing a "Sea of Nodes" [26] intermediate representation following the SSA methodology. This graph allows TurboFan to perform sophisticated optimizations before generating the machine code. Constructing this IR and optimizing it could potentially delay the startup time of the application. Thus, Liftoff was introduced to generate temporary machine code while waiting for TurboFan to produce an optimized replacement.

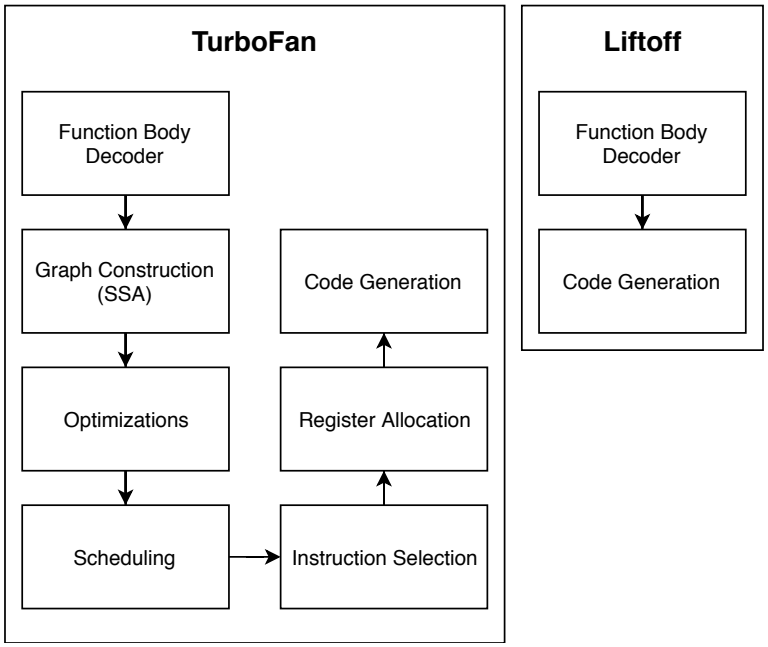

Figure 2.4: Diagram from a V8 blog [2] presenting TurboFan and Liftoff pipelines

2.4.4 Other

The first benefiters of WebAssembly are web browsers. In fact, browsers such as Chrome and Firefox simply reused their JavaScript optimizing compilers to also compile WebAssembly, thus reducing the complexity of the implementation to a simple adaptation of a new input format.

In addition to running WebAssembly on the web, developers saw a potential for the language to run outside of a browser. This is not the first time that a language is being adopted by both the web and a host system. For instance, Java for many years was capable of running applications on the web using Java Applets. However, the latter relied on a non-secure API known as Netscape Plugin Application Programming Interface (NPAPI) which became deprecated by today's popular browsers [27]. Mozilla, partnered with other interested contributors, is working on standardizing WebAssembly System Interface (WASI) [28]. The latter aims to make Wasm binaries run inside and outside a browser while maintaining their main goals: safety, fast execution, portability and compactness. Today, an implementation of WASI has been developed by Mozilla in a standalone runtime for WebAssembly called Wasmtime [29]. This implementation of WASI uses the new optimizing JIT compiler that is being developed by Mozilla (section 2.4.2). In addition to WASI, it is currently possible to run WebAssembly on the server side using Node.js. In fact, some of our experiments described in this report use this environment for executing WebAssembly.

2.5 Machine Learning on Web Engines

Our detailed study of WebAssembly is motivated by two main factors. The first factor is understanding how the language works from a compiler design perspective. The second is our interest in using the language for accelerating machine learning applications on the web. In section 2.5.1 we discuss the current state of machine learning on the web and present some of the most popular machine learning JavaScript libraries. In section 2.5.2 we explain how machine learning on the web can be beneficial to browsers and other embedders of web engines. Finally in section 2.5.3 we highlight how WebAssembly can speed up the existing machine learning libraries which are currently written in JavaScript.

2.5.1 Current State of Machine Learning on the Web

The idea of building a machine learning library on the web was already considered several years ago. In fact, many libraries already exist and cover a number of features for training and using a model on a web browser [30]. Some of the most popular machine learning libraries for the web include: ConvNetJS [31], Tensorflow.js [32], Keras.js [33], Brain.js [34] and WebDNN [35]. All those libraries support model execution, but only ConvNetJS, Tensorflow.js and Brain.js support training on the web. On the CPU backend, the libraries use JavaScript for their computation, except for WebDNN which also support WebAssembly. In addition, Tensorflow.js, Keras.js and WebDNN allow using the GPU for their computations through WebGL.

2.5.2 Benefits of Machine Learning on the Web

Today, the most common form of using machine learning is either on a local computer or by accessing an equipped virtual machine offered by one of the popular web services providers such as Google Cloud, Amazon AWS and Microsoft Azure. Furthermore, Python has been the language of choice for building machine learning models as it is simple to use and has a large online community support. In several cases, such decisions are convenient for developers and data scientists, however, offering an alternative web setup could potentially attract a larger audience interested in the field and give them a hands-on experience without the need for them to install any tools or libraries, but to simply open a web page.

Building a machine learning model currently requires the user to know the basics of programming, and to understand how to interpret ambiguous error messages reported by the compiler and the package manager. Such experience makes using machine learning for a person with superficial knowledge of computers a complicated task. One of the main

advantages for using machine learning on the web is the portability of libraries among web engines, as well as hardware devices. For instance, a model that works on Chrome, would equally work on Firefox or other browsers, and a model that works on a laptop would also equally work on a smartphone. Furthermore, because of the nature of the web, browsers are built to take care of importing all the required libraries by automatically fetching all the dependencies while keeping the entire process transparent to the user.

In addition to the accessibility offered by web browsers, running machine learning on the web has several use cases. For instance, applications and libraries that are currently available on the browser, such as OpenCV.js [13] and Tensorflow Playground [36], require machine learning on the web in order to deliver their intended goals. Another use case for machine learning on the web is augmenting audio and video streams with machine learning capabilities. Applying real time machine learning on a video or audio stream can be inconvenient if the processing is done on the server side. Such design might not provide the user with the most desirable experience and requires them to have a reliable high speed connection. Alternatively offering a client side machine learning library capable of equivalent features could prove to be a more convenient option for the user.

Protection of user privacy in machine learning has been a trending topic for last years and several studies addressed this issue and presented proposals [37, 38, 39, 40]. As machine learning is becoming more integrated in our daily life, many applications are feeding on user data to compete on offering the best results. One approach for protecting users data is federated learning [41]. The latter consist of training a model locally and simply sending the updates to the server instead of the raw data. The server in its turn aggregates the updates from several clients, adjusts a master model and redistributes it to the clients, and the same process is then repeated. Running machine learning on the web supports such design by nature, as the entire model lives on the client side.

Finally, providing machine learning for the web is not limited to browsers. As mentioned in section 2.4, a web browser is simply one embedder of a web engine. Other applications such as Atom editor and Visual Studio Code which are built on top of the V8 engine could also integrate machine learning into their applications. Moreover, smartphone applications contained in a web view, and IoT devices displaying a web-based graphical user interface could equally benefit from such an opportunity.

2.5.3 Machine Learning in WebAssembly

JavaScript is currently the only scripting language supported by web browsers. The language has been used primarily to enhance the user experience and perform simple computa-

tion on the client side. Because of the nature of the language, applications such as machine learning which are computationally heavy would perform much slower compared to native execution. To solve this problem, WebAssembly was proposed to relax such constraints and reconsider the execution of complex programs on the web.

Unlike JavaScript, WebAssembly functions can be eagerly compiled by a web engine while being streamed. When a function is called, the engine does not need to perform further optimization based on the input expressions since types of all locals, parameters and returns are known in advance. Thus, an engine simply executes the JITed machine code right away. This approach not only makes WebAssembly fast and more optimized, but also makes the execution time more predictable [3].

In addition to the performance advantage of WebAssembly over JavaScript, the former is now introducing 128-bit Single Instruction Multiple Data (SIMD) instructions. This feature is currently experimental but a large subset of it has already been implemented in the V8 engine [42]. Since a machine learning model can be implemented in a *vectorized* approach, SIMD can be extremely beneficial in accelerating its execution time. In chapter 6 we present our vectorized implementation of a deep neural network (DNN) library and we discuss how SIMD allowed us to achieve remarkable speedup for using a model.

Machine learning libraries on the web (section 2.5.1) are currently written in JavaScript and support CPU execution, but some also have a GPU backend using WebGL. WebAssembly uses the CPU and currently does not have an interface for interacting with the GPU. On the web, the choice of a machine learning backend depends on various conditions. In the rest of this section we list some of the trade-off for using CPU vs GPU.

A GPU has more computation power compared to a CPU, however in our experiments (chapter 6) with Tensorflow.js using the GPU backend, such advantage was only visible when training on complex models. For simple models, the execution on the CPU provided better overall performance compared to the GPU backend, most likely due to the overhead for accessing the GPU through the WebGL JavaScript API [35, 43]. For complex models, the overhead resulting from using the GPU becomes insignificant compared to the large amount of computations, in which using the GPU outperforms the CPU backend. On the Tensorflow.js website [44], this problem is addressed for inference and a solution is proposed by warming up the model with a dummy prediction. The latter is intended to upload the model weights to the GPU, so that next predictions would execute much faster. The same issue is not addressed for training. Our interpretation is simply that, unlike inference, during training the weights are continuously updated and cannot be uploaded a single time to the GPU, resulting in a consistent overhead. Although complex models can benefit from the GPU performance, if a model becomes too large it might not be usable on a web browser

because the amount of memory used by the GPU is controlled by the browser [30].

An additional, current disadvantage of using the GPU through WebGL is that the implementation of WebGL on different browsers does not follow an identical design. For instance, some implementations, such as on iOS devices, use 16-bit floating point instead of 32-bit [32]. Thus, machine learning models that were trained on a 32-bit system, could potentially deliver imprecise values when used on a 16-bit system [44].

Among the popular machine learning libraries on the web (section 2.5.1), training with a GPU backend is only offered by Tensorflow.js. We believe that with the current technologies on the web, and to cover a wide scope of web engine embedders with limited resources, the option of using CPU or a GPU for training a model should be a parameter that the user should control depending on the complexity of their machine learning model.

Chapter 3

WABT Debugger

Browsers supporting WebAssembly, such as Firefox and Chrome, have augmented their developer tool with a Wasm code inspector feature. The latter allows the developer to recover the Wat code from the Wasm binary and set breakpoints for debugging. This feature facilitates the investigation of issues, but currently requires the project to be deployed as a web application with JavaScript code for loading the Wasm module and calling its functions. In a development environment, especially in console mode, this setup might not be the most convenient as it adds unnecessary complications for testing a Wasm function.

Our first contribution in this thesis is the WABT Debugger. In addition to familiarizing ourselves with the WebAssembly language, we developed this tool to provide a Wasm debugging experience that requires nothing but a terminal. After exploring several execution environment for WebAssembly like Binaryen, SpiderMonkey, V8 and WABT, we decided to build the debugger as an extension to the latter. WABT's comprehensive and direct architecture facilitates the extensibility of the library to integrate a debugging feature. In section 3.1, we describe how we implemented the debugger tool as an extension to WABT. In section 3.2, we present the user interface we developed for enhancing the user experience.

3.1 Extending the Architecture

In section 2.3.1.2 we presented a simplified version of the architecture of the `wasm-interp` tool from WABT. In this section, we extend this architecture to include a debugger and an instruction profiler. Figure 3.1 highlights the additional components we implemented. The debugger executor provides functionalities allowing the developer to control the execution of the WebAssembly bytecode by setting breakpoints and stepping through function instructions. The implementation of the breakpoint system consists of a list storing the offset position of the target bytecode. Stepping through the bytecode and resuming the execution

are achieved by calling functions exposed by the Thread component, which implements the interpretation mechanism.

The profiler executor outlines statistical information about the WebAssembly module. It counts the number of times an instruction has been executed, and records the total and average time it took to execute by the WABT interpreter. Similar to the debugger executor, the bytecode interpretation is done by the Thread component. However, before calling the provided API for starting the interpretation, we implement a wrapper function allowing us to inject profiling code executed every time a Wasm instruction is processed. The profiling code consists of a timer recording the interpreter execution time of an instruction, as well as counts the number of times each instruction was executed. In the case of a call to a Wasm function, the profiler will recursively consider the execution of the instructions contained inside the body of the called function.

Both executors expose accessor functions to the Thread and Environment components in order to query information about the program module, such as the list of Wasm functions, stack value and linear memory content. This information is used to populate our user interface explained in the next section.

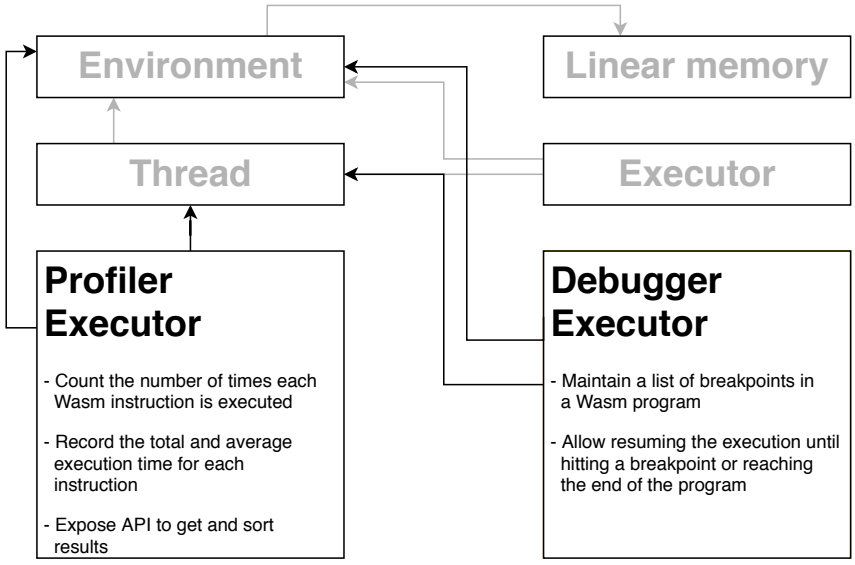

Figure 3.1: Extended architecture of figure 2.1

The source code for this WABT debugger extension can be found at:
`https://github.com/Sable/wabt-debugger`

3.2 Text-based User Interface (TUI)

To benefit from the implemented features described in section 3.1, we build a user interface enabling the user to interact with the tool. Because our goal is to reduce the requirements needed for debugging a Wasm program, we decided to implement the interface in a textual format which can be displayed by a terminal emulator. A popular option for a text-based user interface is Ncurses [45]. The latter is used by many applications such as vim, nano, htop and others [46]. Being written in C, integrating the Ncurses library into our WABT debugger extension written in C++ was a simple task.

The implemented user interface consists of three main displays in addition to a side menu and a home screen. The first display shows the disassembled Wat version of the Wasm binary. Decoding a Wasm file and generating its appropriate list of instructions in Wat format is part of the WABT library and is accessible using their `wasm2wat` tool.

The second display, presented in figure 3.2, shows a layout highlighting various components of a WebAssembly module. In fact, this setup is intended to allow the developer to utilize our debugger executor functionalities. At the top of the display, we provide a snapshot of the WebAssembly *stack* whereas at the right we show a snapshot of the *linear memory* of the module. Both components are updated continuously during execution. On the left of the display, we provide the user with a Wat version of the Wasm binary, however, unlike the first display, we also highlight the next instruction that will be interpreted and mark the lines that have a breakpoint. At the bottom of the display we implement a console allowing the user to interact with the debugger executor. Several commands are currently available for the user in order to benefit from the debugger executor as well as the Thread and the Environment components. For instance, the `main` command allows the user to set the main function of the program. A WebAssembly module does not require the user to define a main function, instead functions can be executed by exporting them to JavaScript, and calling them using their export names. To start the interpretation of the selected main function, the user can enter the `step` command to process one instruction, or `continue` to keep processing instructions until hitting a breakpoint or reaching the end of the selected main function. The full list of functionalities available in this application can be obtained by entering the `help` command in the console.

The third display in this tool, presented in figure 3.3, shows a layout for profiling Wasm functions. At the top, we provide an interactive list showing exported Wasm functions. We currently require the Wasm functions to have an export name in order to appear in that list. Using the keyboard arrows, the user can navigate through the list of functions in order to select (enter key) the one that they would like to profile. The profiling information for a

function are presented at the bottom of the display. This section shows the total and average execution time of an instruction as well as the number of times it was encountered. Depending on the feature of interest, this list can be sorted following the instructions highlighted at the bottom of the screen.

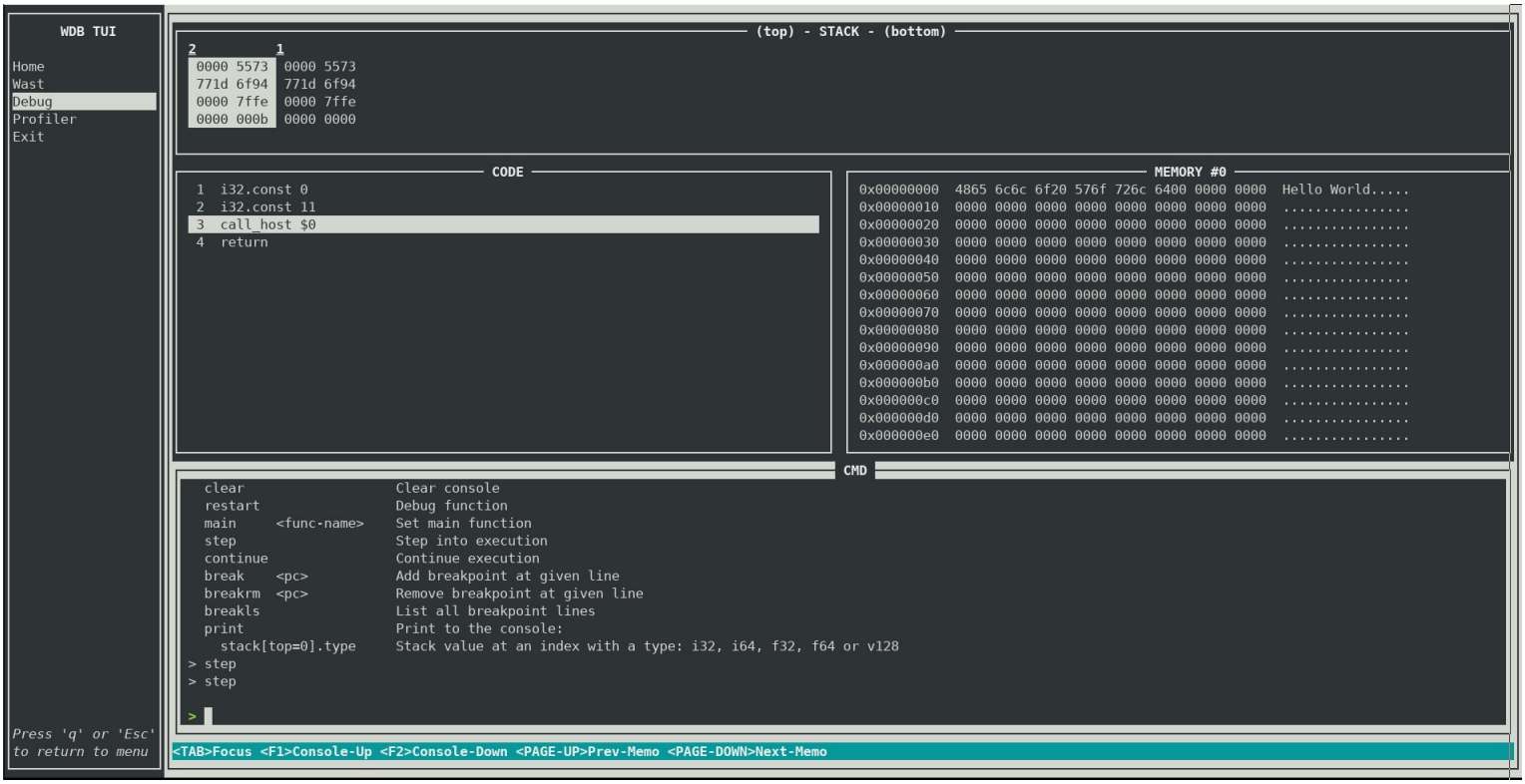

Figure 3.2: `wasm-debugger` debugger display

The source code for this WABT debugger TUI extension can be found at:
`https://github.com/Sable/wabt-debugger-tui`

3.3 Summary

WABT Debugger offers WebAssembly developers a terminal-based environment for debugging and profiling Wasm programs. The tool is built as an extension to the WABT library which decodes the Wasm program and interprets the bytecode. Furthermore, our user interface uses the Ncurses library in order to provide a text-based user interface.

The implementation of this project exposed us to the low-level bytecode of WebAssembly, and allowed us to explore the architecture and the various components composing a Wasm module. In the next two chapters (4 and 5), we utilize our knowledge gained from this

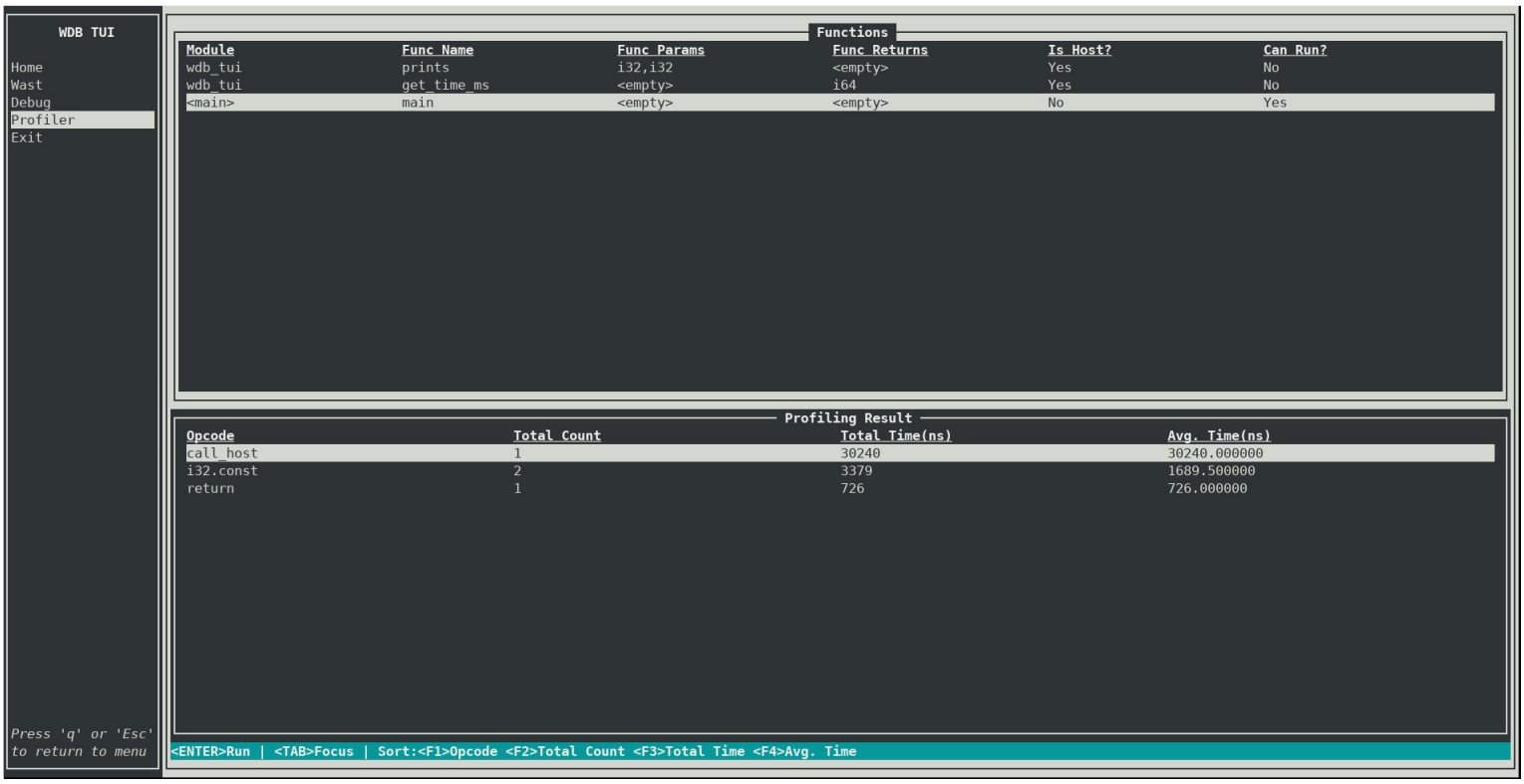

Figure 3.3: `wasm-debugger` profiler display

project in order to dig deeper into the WABT library and modify its source code in order to support our new features.

Chapter 4

Custom Instructions

After exploring WebAssembly at the bytecode level in chapter 3, we decided to dig deeper into the web engine and learn more about the generated machine code. Since V8 was our choice of web engine, each Wasm function had two sets of machine code to inspect. One from Liftoff, the quick compiler, and one from TurboFan, the optimizing compiler. Having two versions of machine code in which one of them is optimized, motivated us to try integrating some of those optimizations into the other unoptimized version. However, this is difficult since important optimizations require more than one pass over the bytecode, and the unoptimized machine code version produced by Liftoff is emitted in a single pass. In fact, performing a single pass is what makes Liftoff a complementary compiler to TurboFan as it is quick to execute. Our attempt to solve this problem was introducing new instructions which could potentially hint to the compiler about what we are trying to do, rather than simply tell it what to execute.

To learn more about some possible optimizations we can port to Liftoff through new Wasm instructions, we wrote a simple machine learning model and we manually detected some of the instruction patterns that were frequently used and inspected their code in both engine compilers. In section 4.1 we present an overview of the machine learning model that inspired the new Wasm instructions. The model aims to learn the OR logical operator using a fully-connected network. An interesting pattern of instructions we detected while writing this program was computing the address of values, such as neurons and weights, in the linear memory. In section 4.2 we introduce `offset32` instruction which attempts to simplify the machine code generated for computing an address offset. Another pattern of instructions that we found interesting appeared when computing the derivative of the sigmoid activation function during the backpropagation algorithm. In section 4.3 we elaborate on the latter and we present two other instructions `dup` and `swap`. In section 4.4, we try another approach for enhancing the execution time by implementing the `exp` instruction for internally calling

the exponential function, omitting any overhead generated by calls to functions imported from JavaScript to WebAssembly. In section 4.6 we present the performance analysis for our experiments using the new instructions and discuss the results reported by our experiments.

4.1 Learning an OR Logical Operator

Before creating custom instructions, it was necessary for us to demonstrate use cases where such instructions could potentially be beneficial. In this section, we present a simple machine learning application we manually developed in order to inspect its machine code generated by the web engine, and learn about opportunities for optimization. The machine learning application aims at learning the OR logical function. The model (figure 4.1) is composed of an input layer with 2 neurons to receive two boolean inputs, 1 hidden layer with 2 neurons and an output layer with 1 neuron. The activation function used for the hidden and output layers is the sigmoid function (section 4.3). The model we coded is not necessarily an optimal one, however in this chapter the complexity and configuration of the model is not a high priority, as long as it still represents the major features of an interesting example in which we can explore optimization opportunities.

This chapter does not describe the details of the model implementation. We postpone those details to chapter 6 which presents our machine learning library and elaborates on the implementation part.

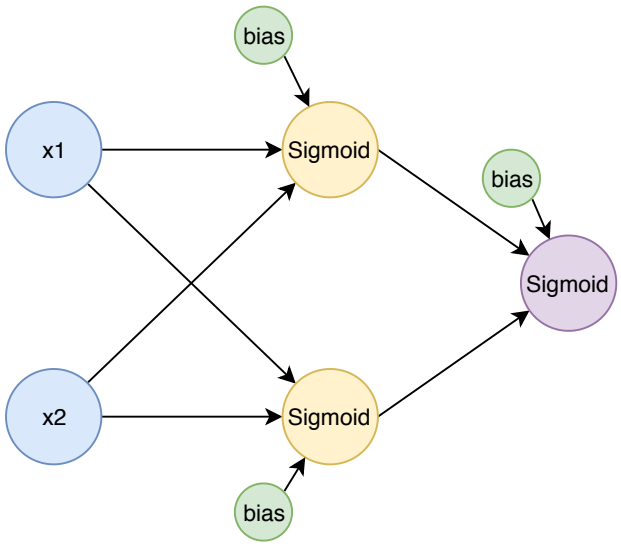

Figure 4.1: Simple model used in our application to train on the OR logical operator

4.2 offset32

WebAssembly stores and loads values from the linear memory of a program module. The latter is simply an array of bytes where integer and float numbers of size 32-bit and 64-bit can be stored. In the current release of WebAssembly, exactly one linear memory exist per module [3]. A linear memory can be exposed to JavaScript using the import and export mechanism designed as part of the language. From the JavaScript side, the linear memory buffer can be wrapped in a view of a particular type. For instance, to treat the WebAssembly linear memory as an array of unsigned 32-bit integers, one can write `let m = new Uint32Array(memory.buffer)` from JavaScript. Accessing the i^{th} element of `m` (e.g. `m[i]`) implies fetching a `uint32` element in the WebAssembly linear memory at the position `m + i * 4`. WebAssembly does not have the concept of wrapping the linear memory with a view. Therefore, addresses of elements in the linear memory need to be explicitly computed before reading or writing a value. An example of computing an address is shown in listing 4.1.

```
get_local $index    ;; index number
i32.const 4         ;; size of i32 (4 bytes)
i32.mul             ;; compute relative address
get_local $base     ;; base address
i32.add             ;; compute absolute address
i32.load            ;; load element at the computed absolute address
```

Listing 4.1: Example of computing an absolute address

In this example, we load the value at index `$index` from an integer array located at address `$base` (e.g. `((int*) base) + index`). This process requires performing 2 arithmetic operations (`i32.mul` then `i32.add`) to compute the absolute address before loading the target value from the linear memory. In our machine learning code, we used this pattern sufficiently frequently that we decided to promote it into a function. Inspecting the machine code for this function in V8 engine showed us the potential enhancement we can bring to Liftoff by introducing an instruction that can directly compute the absolute address. Listing 4.2 compares the `x86` generated for this example in Liftoff and TurboFan compilers. In the Liftoff code, we notice that the local `$index` is stored in register `rax` and `$base` is stored in register `rdx`. Computing the absolute address requires first storing the size of an `i32` (`0x4`) into register `$rcx` then performing a multiplication followed by an addition. In the TurboFan code, similar to Liftoff, the registers `rax` and `rdx` store `$index` and `$base` respectively. However, without having to use a third register it is capable of computing the absolute

value in a single load-effective-address (`lea`) instruction. Moreover, if the local `$base` had a constant value, TurboFan can even save one more register by directly injecting the constant integer into the `lea` instruction (e.g. for `$base` equal to 12, the instruction changes to `leal rax,[rax*4+0xc]`), whereas Liftoff would still have to first load the constant into a register before performing the addition.

Note that in the machine code presented in this section, we omit the instructions related to loading the value from the memory (e.g. `mov`).

```
movl rcx,0x4
imull rcx,rax
addl rcx,rdx
```

(a) Liftoff

```
leal rax,[rdx+rax*4]
```

(b) TurboFan

Listing 4.2: Machine code generated by Liftoff and TurboFan

Since this pattern of instructions has been used frequently in our code, we introduced an `offset32` Wasm instruction which aims to use TurboFan's machine code directly inside Liftoff. The `offset32` instruction takes 3 integer parameters and returns an integer absolute address. The first parameter is the base address, the second is the index, and the third is the type size. The suffix `32` of the instruction refers to the address size for the linear memory (32-bit). Listing 4.3 presents the same example explained previously but using the `offset32` instruction. The machine code generated by this example is shown in listing 4.4. Using `offset32`, we saved one register (`rcx`) and reduced the 2 arithmetic operations into a single load-effective-address instruction. Furthermore, we added the case of the local `$base` storing a constant, in which we also save one extra register by directly using the constant inside the instruction.

```
get_local $base      ;; base address
get_local $index     ;; index
i32.const 4          ;; size of i32 (4-bytes)
offset32             ;; compute final address
i32.load             ;; load element from linear memory
```

Listing 4.3: Syntax for `offset32` instruction

```
leal rcx,[rdx+rax*4]
```

(a) Liftoff

```
leal rax,[rdx+rax*4]
```

(b) TurboFan

Listing 4.4: Machine code generated for `offset32` in Liftoff and TurboFan

4.3 `dup` and `swap`

In this section, we describe two instructions `dup` which duplicates the top element of the stack, and `swap` which swaps the top two elements of the stack. Unlike `offset32`, the two instructions operate only on the stack level. Stack manipulation instructions are common in other stack-machine architectures, such as Java and Python bytecode, but not many are currently available in WebAssembly. In fact, the implementation of `dup` and `swap` are currently raised as part of the "Open Questions" under the "Multi-value" proposal for WebAssembly [47]. Because WebAssembly currently has locals which can store values to be reused, the behavior of the two instructions can already be achieved. However, those instructions can be beneficial in other ways, such as coding convenience and reducing the bytecode size. The latter is important in WebAssembly, especially as the binary files are shipped over the network.

$$\sigma(x) = \frac{1}{1 + e^{-x}} \tag{4.1}$$

$$\frac{\partial \sigma}{\partial x} = \sigma(x) \times (1 - \sigma(x)) \tag{4.2}$$

In our small machine learning program presented in section 4.1, we used the sigmoid (equation 4.1) activation function for the network neurons. As part of the backward propagation algorithm, we compute the derivative (equation 4.2) of the activation function (chapter 6).

For the purpose of this section, we do not use the optimization which reuses the sigmoid value computed during the forward propagation algorithm to compute the function derivative in the backward propagation.

Listing 4.5 shows three versions of implementing the derivative of the sigmoid function. Version (a) computes the derivative exactly as presented by the equation 4.2. This version is the simplest to write or generate, but it is inefficient because it performs the same call to the exponential function (`$exp`) twice. Version (b) shows how ideally the function should

currently be implemented in WebAssembly where the call to `$exp` is performed once, then cached and reused in two places (`tee_local` copies the value on the top of the stack to a local variable without removing it from the stack). Finally, version (c) shows our implementation which benefits from `dup` and `swap`.

We note that despite the code improvement provided by `dup`, the actual machine code generated for versions (b) and (c) are almost identical for both the Liftoff and TurboFan compilers. The only minor difference was in version (b) in which there was an additional `mov` instruction for moving the final result from register `rcx` into `rax` in Liftoff compiler. The existence of such register copy instructions in the generated code is of relatively low impact and is highly context dependent as well, but also could be an opportunity for optimization in the engine or simply a limitation of a single pass compiler, in which our version would be more optimal.

```
(func $sigmoid_prime
  (param $x f32)
  (result f32)
  get_local $x
  call $exp
  f32.const 1
  get_local $x
  call $exp
  f32.sub
  f32.mul
)
```

(a)

```
(func $sigmoid_prime
  (param $x f32)
  (result f32)
  (local $cache f32)
  get_local $x
  call $exp
  tee_local $cache
  f32.const 1
  get_local $cache
  f32.sub
  f32.mul
)
```

(b)

```
(func $sigmoid_prime
  (param $x f32)
  (result f32)
  get_local $x
  call $exp
  dup
  f32.const 1
  swap
  f32.sub
  f32.mul
)
```

(c)

Listing 4.5: Different versions of implementing the derivative of the sigmoid function

4.4 exp

In this section, we introduce the `exp` instruction which calls the V8 built-in exponential function internally instead of importing it from JavaScript. The motivation behind this approach is to accelerate function calls by omitting any overhead related to calling JavaScript from a WebAssembly module. The exponential function is a perfect candidate for such optimization, because it can be used intensively in a machine learning model and the function is already available in the engine and used in JavaScript by `Math.exp`. In our small machine

learning model (section 4.1), we used the sigmoid (section 4.3 - equation 4.1) activation function for both the hidden and output layers, and thus for each data entry the exponential function is called 3 times during the forward propagation algorithm. To train the OR logical operator we use 4 data entries (`00, 01, 10, 11`). And to learn the function with 100% percent accuracy we ran the model for more than 500 epochs. Thus the number of times the exponential function is called in the forward propagation algorithm while training is more than $3 \times 4 \times 500 = 6,000$ times. Networks that learn more complex tasks have many more hidden layers, neurons per layer and much more training data which can easily explode the number of times the function is called by multiple orders of magnitude, even when trained for a single epoch.

4.5 Implementation

Depending on the functionality, the implementation steps for a custom instruction can be different. In this section, we will explain the procedure we followed in order to integrate `offset32` into the language (section 4.2). Compared to the other custom instructions, `offset32` was the most challenging instruction as it requires the most changes in the engine. Because of the complexity of the code and the large amount of changes necessary for our custom instruction, this explanation will present only certain code snippets, and will primarily focus on describing the important concepts. For readers interested in manipulating the language at a low-level, we believe this section offers an introduction to the procedure of adding custom instructions in WebAssembly. The full code implementation can be found at our repository link posted at the end of this chapter.

Figure 4.5 presents an overview of the changes required for implementing `offset32` instruction. At the beginning, we need to determine certain specifications required for any new instruction such as the opcode, number and type of parameters, type of result and behavior of the new instruction. Once this information is defined, we start our changes in the WABT library. First, we update the library lexer in order to recognize our new instruction. Second, we modify the parser to expect and consume our instruction inside a Wasm function. Third, we modify the type checker to consume three integer parameters from the stack and push an integer result. This step prevents generating incorrect Wasm binary, which would fail when decoded by a Wasm engine. Fourth, we update the binary writer to emit our selected opcode. The details of these steps are presented in section 4.5.1. The next major step is integrating our instruction into the V8 engine. First, we update the function body decoder to read our instruction opcode. In the same modifications we also perform the stack manipulation following the instruction specification and the WABT changes. Second, we update the

TurboFan compiler in order to represent the instruction operation using a graph intermediate representation. Third, we modify the Liftoff compiler to directly emit the corresponding machine code. In section 4.5.2, we cover the details of the changes inside V8.

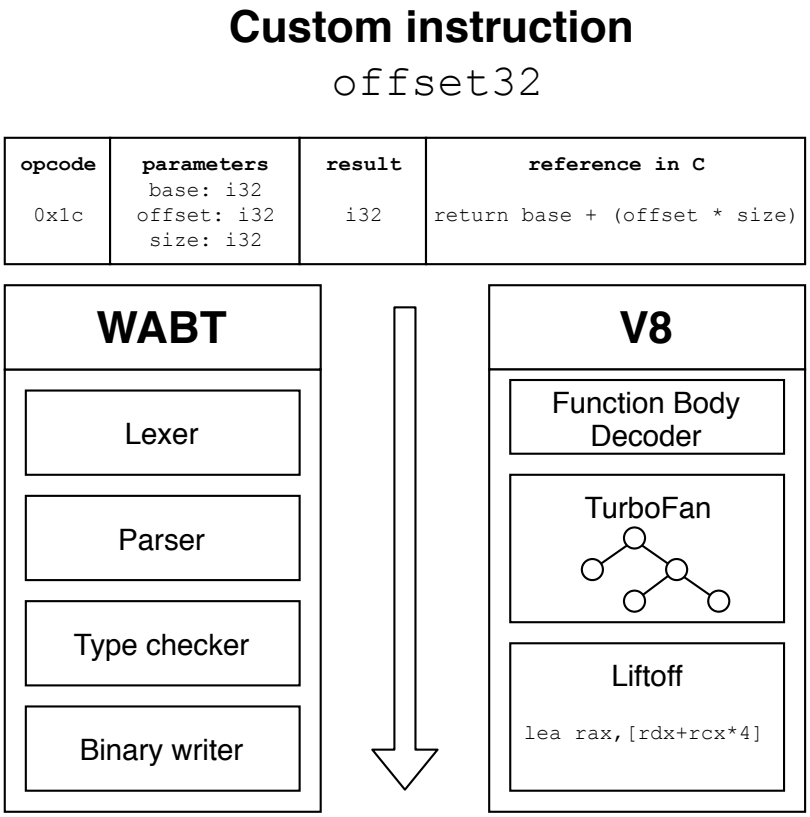

Figure 4.5: Implementation of the `offset32` instruction

4.5.1 WABT Changes

Each instruction in WebAssembly is assigned a unique opcode. Thus, before starting to write any lines of code, we need to give our new instruction an opcode which is not already in use. In WebAssembly, an opcode is simply a number that is composed of one byte with an optional prefix byte. The list of opcodes already in use can be found in the language specifications [48]. However, we recommend checking the opcodes from the source code of V8 (`wasm-opcodes.h`) since the engine contains additional experimental instructions with opcodes that could potentially conflict with the new custom instruction. For `offset32`, we selected the opcode `0x1c`, although recently this opcode has been utilized by the explicit-type `select` instruction [48].

The goal of our modifications in WABT is to enable the library to recognize `offset32` as an instruction inside a Wat file. Thus, after registering the opcode for `offset32` in WABT

(`opcode.def`), we start by adding the corresponding token into the lexical analyzer. By the time of our development, the tokens were added in `wast-lexer.cc`, however recently the library updated their analyzer and the changes need to be done inside `lexer-keywords.txt`. In addition to augmenting the lexer with the new token, the latter also needs to be inserted in the list of tokens (`token.def`). To make the implementation process much simpler, we recommend looking at the code for an existing instruction which has the same signature. In our case, we primarily inspected the code for the implicit-type `select` instruction which has an identical signature.

Updating the lexical analyzer allows the library to tokenize our instruction. However, we still cannot use it in a Wat file since the library does not know where to expect this instruction. To solve this issue, we need to update the parser (`wast-parser.cc`) in order to mark `offset32` as a function body instruction and construct the intermediate representation (IR) that correspond to it. Because `offset32` is a new instruction, we need to define an IR class describing it in `ir.h`. The changes in the parser and the IR are also inspired by the code of the implicit-type `select` instruction.

At this stage, the library can tokenize and parse the instruction, but it will completely ignore it since we have not specified how it manipulates the stack, nor how to generate Wasm. This brings us to the next important step, type checking (`type-checker.cc`). Listing 4.6 shows the code simulating the stack manipulation in order to validate the instruction signature types. The first part of the code consumes and validates the top three 32-bit integer parameters from the stack. A wrong type results in an error message. The second part of the code pushes into the stack the result type which is also a 32-bit integer. The top of the stack is then consumed by the next instruction following `offset32` and the process is repeated until the entire file is validated.

At this point, our instruction is almost part of an extension for the WebAssembly language. Depending on what the reader wants to do with instruction, this step can be different. In our case, we want to convert our instruction in Wat format into Wasm binary. WABT provides several other options for using this instruction. For instance, we can augment the library interpreter to execute our instruction, or we can regenerate the Wat output by adding the syntax for the `offset32` into the Wat writer. To generate the Wasm binary, we simply need to write the instruction opcode into the code stream (`binary-writer.cc`) as shown in listing 4.7.

In this section, we highlighted the most important steps for adding our custom instruction into WABT. In between steps, minor code snippets are required to connect the different parts but we omitted their explanation in order to avoid distracting the reader from the important points. Usually, those mini-steps are reported by the compiler in form of error or warning

```
Result TypeChecker::OnOffset32() {
  Result result = Result::Ok;
  // Consume and check top three i32 parameters from the stack
  result |= PeekAndCheckType(0, Type::I32);
  result |= PeekAndCheckType(1, Type::I32);
  result |= PeekAndCheckType(2, Type::I32);
  result |= DropTypes(3);
  // Print an error if types failed
  PrintStackIfFailed(result, "offset32", Type::I32);
  // Push the result type into the stack
  PushType(Type::I32);
  return result;
}
```

Listing 4.6: Code added to `type-checker.cc`

```
void BinaryWriter::WriteExpr(const Func* func, const Expr* expr) {
...
  switch(expr->type()) {
    ...
    case ExprType::Offset32:
      WriteOpcodes(stream_, Opcode::Offset32);
      break;
    ...
  }
...
}
```

Listing 4.7: Code added to `binary-writer.cc`

messages when compiling the library. Once changes are successfully applied to the WABT library, the `wat2wasm` (section 2.3.1.1) should be able to recognize and convert `offset32` instruction into its corresponding binary representation.

4.5.2 V8 Changes

With the WABT changes (section 4.5.1), we are now capable of writing WebAssembly in Wat format and generating the corresponding Wasm binary. By default, trying to run the Wasm binary, containing our new instruction, in V8 will fail with an error message reporting an invalid opcode. Obviously, this is the behavior we are expecting and it is a positive sign

that our WABT modifications work. Because our `offset32` is a function body instruction, we need to start our modifications in V8 in the function body decoder. But before we explain the changes, we need to revise the V8 diagram in figure 2.4 (page 14). V8 has two compilers, TurboFan which creates an intermediate representation for each function, and Liftoff the single-pass compiler and direct machine code generator. As shown in the diagram, the function body decoder step is common for both compilers. Thus, the changes we present for this step will be executed twice, but the steps that follow are specific for each compiler.

We initiate our changes in V8 by registering the opcode of our `offset32` instruction. This is simply done by adding an entry for `offset32` in `wasm-opcodes.h` and aligning the opcode value with the one chosen in the WABT library (section 4.5.1). Now that the code can resolve our opcode, we start by defining the decoder behavior when encountering the instruction opcode. Listing 4.8 presents the code added to `function-body-decoder-impl.h` in order to decode the `offset32` instruction. Similar to listing 4.6, the first part of the code consumes and checks the top three 32-bit integer parameters from the stack. Because of the nature of the data structure, the parameters stored in the stack are popped in reverse order. The second part of the code pushes the result type of the instruction into the stack. The third part of the code, calls a function `Offset32(...)` (abstracted by the C++ macro) that is defined in both Liftoff and TurboFan compilers, and passes the instruction parameters and result as arguments. In the next paragraph, we will explain how we defined the code in TurboFan, then later we will describe the changes in Liftoff.

```
switch(opcode) {
  ...
  case kExprOffset32: {
    auto size    = Pop(2, kWasmI32);
    auto offset  = Pop(1, kWasmI32);
    auto base    = Pop(0, kWasmI32);
    auto* result = Push(kWasmI32);
    CALL_INTERFACE_IF_REACHABLE(Offset32, base, offset, size, result);
    break;
  }
  ...
}
```

Listing 4.8: Code added to `function-body-decoder-impl.h`

As we explained in section 4.2, the TurboFan compiler can already optimize address computation (section 4.2). Thus, our changes in TurboFan are simply injecting those com-

putation operations into the graph intermediate representation, which are later taken care of during the optimization step. Figure 4.9 presents the code for constructing the address computation subgraph. The first expression in the `Offset32` function corresponds to building the multiplication node of the `offset` and `size` parameters. The second expression builds an addition node from the `base` and the multiplication node. The last expression of the code assigns the constructed subgraph to the `result` pushed into the stack, which is later picked up by another instruction in order to build a larger subgraph. Figure 4.6 provides a graph representation of the code.

```
void Offset32(FullDecoder* decoder, const Value& base,
              const Value& offset, const Value& size, Value* result) {
  auto mul_node = BUILD(Binop, kExprI32Mul, offset.node, size.node,
                        decoder->position());
  auto add_node = BUILD(Binop, kExprI32Add, base.node, mul_node,
                        decoder->position());
  result->node = add_node;
}
```

Listing 4.9: Code added to `graph-builder-interface.cc`

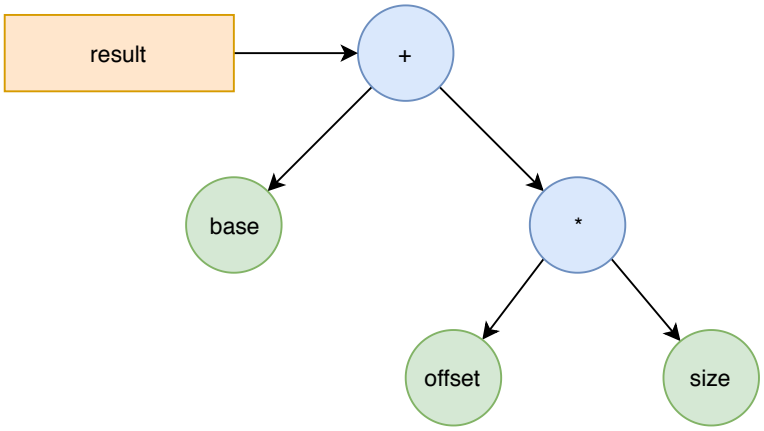

Figure 4.6: TurboFan graph representation of `offset32`

The modifications in Liftoff are more complicated compared to TurboFan since the former skips any intermediate representation and directly generates machine code. Because V8 supports multiple architectures, the machine code needs to work on various platforms. However, for the purpose of our experiments, we only apply the changes for `x86` architecture, and fallback to two arithmetic operations (multiplication followed by addition) on other architectures. Our implementation for `offset32` emits different machine code depending on

the input location (e.g. register, stack or constant). Listing 4.10 presents part of the code which handles the case of a constant `size` operand with a value 1, 2, 4 or 8 (e.g. `i32.const 4`). The four value options are the ones supported by the load-effective-address instruction which is targeted by our implementation as a replacement for the two separate binary operations. In the first part of the code, we pop the `offset` operand from the stack into a general purpose register. The stack used in the Liftoff compiler is different than the stack we mentioned in the function body decoder. In the latter, the stack is used by TurboFan for validating types and storing graph nodes, and by Liftoff for validating types only. The stack in Liftoff compiler allows the latter to keep track of the values, such as operands of a binary operation, which are then popped to emit machine code. In the second part of the code, we implement two blocks of code—the first executes if the base address is a constant, otherwise the second block is activated. In the first block, we remove the `base` address from the stack, and we use it directly as a constant saving a register slot. In the second block, we have to load the `base` address into a general purpose register before using it. In both blocks, the function responsible for generating the machine is `emit_offset32` which has two implementations depending on the branching condition. The two implementations are presented in listing 4.11. The first function, generates an instruction where the base address is a constant, the second generates an instruction where the base address is stored in a register. Finally after generating the machine code, we store the destination register into the stack which is then consumed by later instruction. In the case where the `size` input is not a constant or not one of the four values supported by load-effective-address, then we again fallback to two arithmetic operations as shown in figure 4.12. The code simply pops `size` and `offset` operands into registers, then emits the multiplication code using `emit_i32_mul` function and stores the result into a general purpose register reference by `dst`. The next part of the code pops the `base` operand into a general purpose register and emits the addition code using `emit_i32_add` function. At the end, the destination register is pushed into the stack which is later consumed by another instruction.

By updating the TurboFan and Liftoff compilers to support our new instruction, the V8 engine should now be able to emit the corresponding machine code in both modes. The generated machine code can be inspected in V8 by passing the flag `--print-wasm-code` to a V8 embedder such as Node or Chromium (example in section 4.2).

4.6 Performance Analysis

After implementing the four WebAssembly instructions, it was time to do some performance analysis. We divide our analysis into two experiments, both of which are based on

```
void Offset32(FullDecoder* decoder, const Value& base,
              const Value& offset, Const Value& size, Value* result) {
  ...
  // Pop offset into a register
  LiftoffRegister offset_reg = __ PopToRegister();

  // Check if the base address is a constant
  // or is stored into a register or stack
  LiftoffAssembler::VarState base_slot = __ cache_state()->stack_state.back();
  if(base_slot.loc() == KIntConst) {

    // If it is a constant
    // then we can save one more register
    __ cache_state()->stack_state.pop_back();
    int32_t immBase = base_slot.i32_const();
    LiftoffRegister dst = __ GetUnusedRegister(result_rc, {offset_reg});

    // Generate machine code
    __ emit_offset32(dst.gp(), immBase, offset_reg.gp(), size_val);

    // Push destination register into stack
    __ PushRegister(kWasmI32, dst);
  } else {

    // Pop base address into a register
    LiftoffRegister base_reg = __ PopToRegister(LiftoffRegList::ForRegs(offset_reg));
    LiftoffRegister dst = __GetUnusedRegister(result_rc, {offset_reg, base_reg});

    // Generate machine code
    __ emit_offset32(dst.gp(), base_reg.gp(), offset_reg.gp(), size_val);

    // Push destination register into stack
    __ PushRegister(kWasmI32, dst);
  }
  ...
}
```

Listing 4.10: Code added to `liftoff-compiler.cc`

the model described in section 4.1. In section 4.6.1 we analyze the performance of the model using the custom instructions `offset32`, `dup` and `swap`. In section 4.6.2 we compare the execution time using the imported exponential function with our custom instruction `exp`.

```
// Case of base address is a constant (1, 2, 4 or 8)
void LiftoffAssembler::emit_offset32(Register dst, int32_t base,
                                     Register offset, ScaleFactor size) {
  leal(dst, Operand(offset, size, base));
}

// Case of base address is not a constant
void LiftoffAssembler::emit_offset32(Register dst, Register base,
                                     Register offset, ScaleFactor size) {
  leal(dst, Operand(base, offset, size, 0));
}
```

Listing 4.11: Code added to `liftoff-assembler-x64.h`

```
void Offset32(FullDecoder* decoder, const Value& base,
              const Value& offset, Const Value& size, Value* result) {
  ...
  LiffoffRegister size_reg = __ PopToRegister();
  LiftoffRegister offset_reg = __ PopToRegister(LiftoffRegList::ForRegs(size_reg));
  LiftoffRegister dst = __ GetUnusedRegister(result_rec, {size_reg, offset_reg});
  __ emit_i32_mul(dst.gp(), offset_reg.gp(), size_reg.gp());
  LiftoffRegister base_reg = __ PopToRegister(LiftoffRegList::ForRegs(dst));
  __ emit_i32_add(dst.gp(), dst.gp(), base_reg.gp());
  __ PushRegister(kWasmI32, dst);
  ...
}
```

Listing 4.12: Code added to `liftoff-compiler.h`

4.6.1 offset32, dup and swap

In this experiment, we use the model from section 4.1 and vary the number of hidden layers to range between 1 and 8. Moreover, we integrate three custom instructions into our WebAssembly program. That is, we replace manual address computation with `offset32` and we use the `dup` and `swap` instructions to implement the derivative of the sigmoid function as described in section 4.3. Because those instructions only affect the Liftoff compiler, we also pass a flag to the V8 engine `--no-wasm-tier-up --liftoff` to disable TurboFan for our Wasm functions. This experiment was executed on Node v12.2.0 running on a desktop computer with an Intel i7-8700K. Figure 4.7 shows the training time per epoch for the 8 different models, with (dup, swap, offset32) and without (None) using the three instructions.

In general we notice that as the model depth increases, the execution time also increases as a result of more computations. Furthermore, comparing the execution time of the two versions of each model, we notice that the performance gain from using the three custom instructions was not enough to make a significant improvement. Our results reported an average of 1.02× speedup. The latter value reflects the updates to the machine code that our instructions applied, however, the contribution of those modifications compared to the overall program task were minor. Enabling TurboFan results in identical execution time for the two versions. This is expected because our modifications to the Liftoff version of the model are quickly overwritten by the optimizing compiler, especially since our Wasm functions for those models are fairly small.

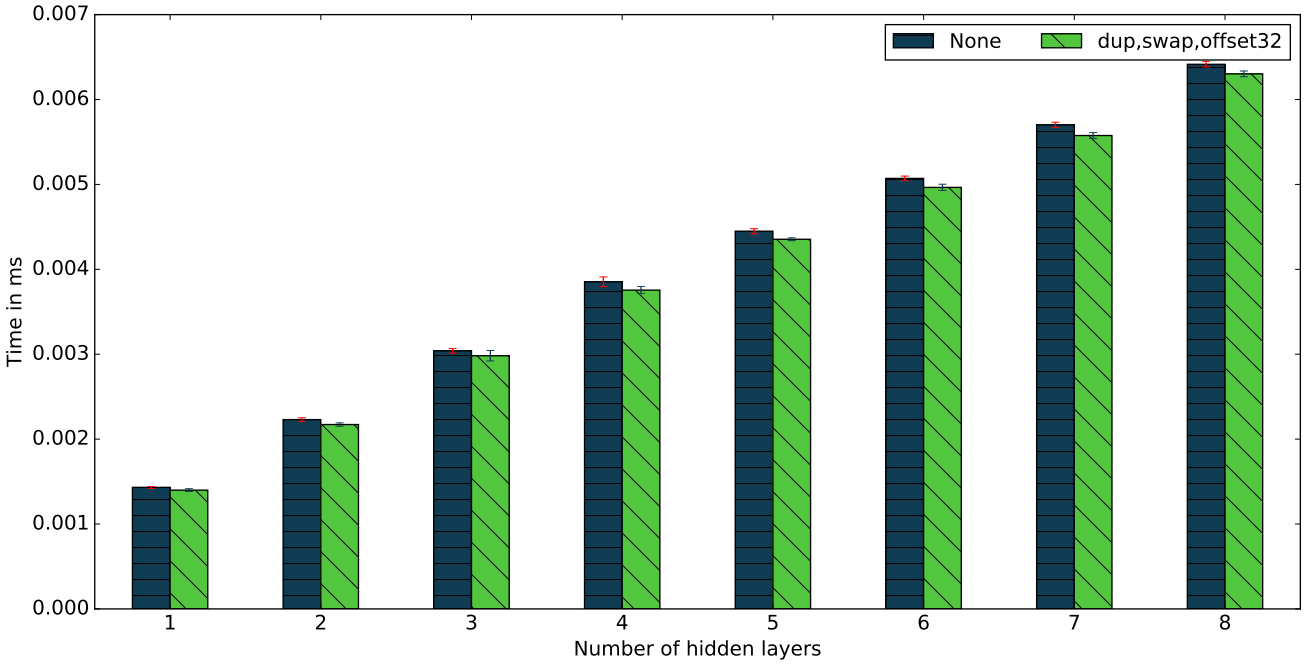

Figure 4.7: Training time per epoch for 8 models, with and without using `offset32`, `dup` and `swap`

4.6.2 exp

In this experiment, we use a similar setup as the one presented in section 4.6.1, but instead we measure the performance gain obtained by eliminating the call overhead to the exponential function. To achieve that, we compare the training time per epoch for 8 models with hidden layers ranging between 1 and 8, using the regular `call` instruction, which calls the imported JavaScript version, and using our custom `exp` instruction. Moreover, we do

not restrict the V8 engine to only use the Liftoff compiler. The Node version used in this experiment is v11.9.0, the latest one available during our development, however we report the average speedup on a newer version of Node at the end of this section. Figure 4.8 shows the results of our experiment. In general, we notice that using our instruction, the execution time is consistently faster. In fact, the average speedup when using `exp` is 1.28×. The advantage of our `exp` implementation over calling an imported JavaScript exponential function into WebAssembly, is omitting an expensive overhead resulting from the communication between the two languages.

Node v12.0.0, released on the 23rd of April 2019, uses a new version of V8 which reduces this overhead. The exponential function as well as other built-in math functions have been manually coded to follow a faster execution approach than a regular imported JavaScript function [49]. Repeating this experiment with Node v12.2.0, we obtained an average speedup of 1.04× *using the* ***`exp`*** *instruction.*

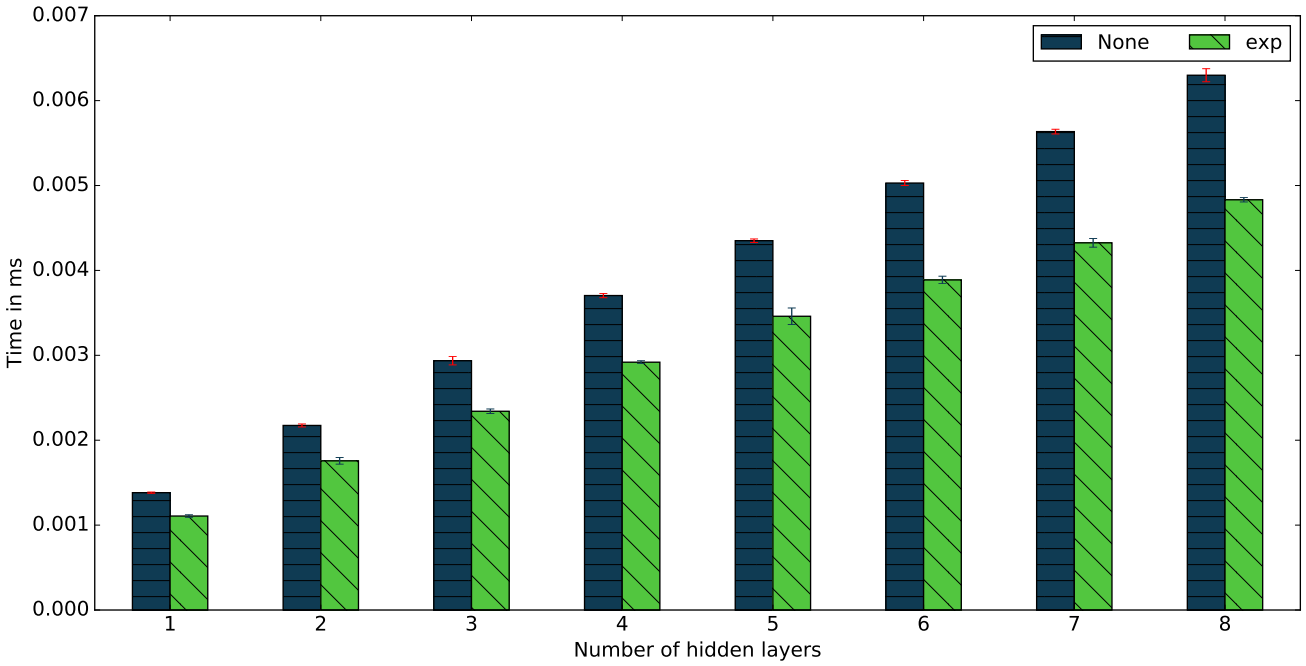

Figure 4.8: Training time per epoch for 8 models, with and without using `exp`

4.6.3 Summary

In this chapter, we introduced in total of four custom instructions. `offset32` aimed at optimizing the machine code generated by Liftoff for address computations. Instructions `dup` and `swap` implemented stack manipulation operations that are common in other stack

machine architectures. Finally, the `exp` aimed at omitting the call overhead generated by making calls to the exponential function imported from JavaScript. In this section, we discuss our observations and propose future considerations for adding custom instructions.

In our experiments, we noticed that optimizing the Liftoff compiler for address computation did not improve the overall program execution time when both engine compilers were enabled. Because the machine code generated by Liftoff is temporary and will be replaced by an optimized version, targeting Liftoff requires more considerations in order to study the impact a new instruction will have on the performance of a program. Our experiments with `offset32` did not show significant improvement when disabling TurboFan, however, our programs were based on a small machine learning application that we wrote manually. On larger programs, especially projects compiled from C/C++, the JITter might spend more time executing the machine code generated by the Liftoff compiler, therefore new instructions might prove to be more beneficial.

Unlike other bytecode languages, WebAssembly currently does not have many instructions for manipulating the stack. In fact, the language currently uses locals in order to duplicate the top of the stack and swap the top two elements in the stack. In our usage for `dup` and `swap`, we confirmed that the instructions will eventually generate machine code similar to a Wasm program that uses locals instead. Thus, the primary benefit of such instruction is the convenience of using them and reducing the number of bytecode in a Wasm binary file.

Calling imported JavaScript functions from WebAssembly generates an overhead which becomes noticeable when used frequently. Our implementation for `exp` aimed at adding the exponential function as part of the language instruction set, instead of a feature available externally. In our experiments, we noticed that using `exp` optimized the total execution time. However, repeating our experiment on a newer version of V8, did not maintain the same improvement since the engine implemented a solution that enhances the performance of imported math functions.

In addition to our strategies for using custom instructions, the latter can be utilized in the future for various other reasons. For instance, new instructions can implement an interface for interacting with external accelerators which are beneficial for compute heavy applications such as machine learning. Moreover, in chapter 5 we implement another instruction allowing us to call native C++ functions directly from WebAssembly, without the need to pass by JavaScript, thus omitting any overhead resulting from the communication between two languages. Our contribution in this chapter helped us build a solid knowledge about the low-level details of the WebAssembly implementation. Furthermore, it allowed us to better use the language in our machine learning library presented in chapter 6.

The source code for our modified version of the WABT library can be found at:
`https://github.com/Sable/wabt-experimental`

The source code for our modified version of Node.js can be found at:
`https://github.com/Sable/node`

Chapter 5

Native calls

Unlike JITed JavaScript and WebAssembly code, native functions imported by WASM but implemented in C++ are compiled ahead-of-time (AOT). This characteristic has an advantage in reducing the startup time of a program since the host environment has already compiled and optimized the code in advance. In fact, AOT compilers can spend a significant amount of time optimizing static code, whereas a JIT compiler should find the right balance between minimizing the time spent on compilation and applying expensive optimizations [50]. Although native functions can be more optimized and as a result improve the execution time of a program, they could also introduce vulnerabilities to web engines. For instance, a library that performs highly optimized linear algebra operations might not necessarily put security as a top priority. On the web, this can be problematic, especially when strange input is not handled or memory access is unchecked. In our study, we use this feature to highlight the performance gain offered by native functions (chapter 6), and although it limits our design to known, safe contexts, we leave the security challenges to future work.

In this chapter, we explore the possibility to perform native calls to functions written in C++ from WebAssembly. The idea of implementing certain functions natively is not new in V8 and SpiderMonkey. For example, the latter two engines implement the JavaScript math functions natively into their web engine [51, 52]. Native implementations are not limited to JavaScript functions; in fact, V8 also implements certain WebAssembly instructions by calling external references [53]. To find the best approach for implementing and experimenting with native functions, we present three options. In section 5.1 we discuss the option of using custom instructions to make function calls to external references. In section 5.2 we discuss an option available in Node which allows us to write native functions and use them in WebAssembly by importing them first from JavaScript. In section 5.3 we introduce a `call_native` instruction for calling native functions, eliminating the overhead present in the Node option. Finally, in section 5.4 we measure the call overhead generated by going

through JavaScript.

5.1 Custom Instructions Option

One solution for calling a natively implemented function is using custom instructions. In fact, the `exp` instruction we discussed in section 4.4 does exactly that. Internally, the exponential function is implemented as a C++ function, and the `exp` instruction simply makes a call to the appropriate external reference. In terms of functionality, this option works perfectly, however, as we described in chapter 4, implementing a custom instruction is a complex task, and repeating the process for each native function would be inconvenient.

5.2 Node API Option

Node is a popular V8 embedder which allows developers to use JavaScript and WebAssembly on the server side. Node currently allows the user to implement native functions in C++ and makes them available in JavaScript by simply importing them (e.g. `let lib = require(addon);`). This option works great since we can simply import the native functions from JavaScript then import them again from WebAssembly. However, as we discussed in section 4.4, intensive use of JavaScript functions from WebAssembly generates a noticeable overhead. In the new V8, this issue is no longer applicable for math functions, but remains for other built-ins.

To understand the origin of this overhead, we take a step backward and dig deep again into the V8 code. Calls from WebAssembly to imported JavaScript functions are classified into several kinds. A kind depends on two primary conditions. First, check if the number of arguments for the imported function matches in both JavaScript and WebAssembly. Second, check if the JavaScript function is marked as strict. A strict function is identified by the expression `"use strict";` inserted at the beginning of the function body, and allows the engine to be more restrictive about the usage of JavaScript. While debugging V8, we realized that native functions written using Node API and imported from JavaScript into WebAssembly appear to have parameter count mismatches and are marked as non-strict. This kind of imported function is in fact expensive. Before executing the machine code of the target function, the call has to visit a wrapper function and then an arguments adaptor frame. The wrapper function is mandatory for all JavaScript import kinds, but the frame is an extra overhead resulting from not detecting the number of arguments of the native function, and thus the import is considered to have an argument mismatch. The machine code for the wrapper and the frame can be inspected by passing `--print-all-code` flag

to the V8 engine. In section 5.4 we measure the time spent executing this overhead code. In the next section, we implement another approach that allows us to call native functions without passing through JavaScript.

5.3 `call_native` Option

In section 5.1, we explained the possibility of performing native calls using custom instructions, and we also highlighted the inconvenience of repeating this process for every function we would like to experiment with. To solve this problem, we use an approach similar to how WebAssembly currently imports JavaScript functions. Figure 5.1 shows an example of a Wat code importing a JavaScript function into WebAssembly. This function simply adds two integer numbers. The first part of the code imports the function from JavaScript, assigns to it the alias `$add_i32` and declares its signature. The second part of the code shows a Wasm function calling the imported JavaScript function.

```
(import "math" "add_i32" (func $add_i32 (param i32 i32) (result i32)))

(func
  (result i32)
  i32.const 1
  i32.const 2
  call $add_i32
)
```

Listing 5.1: Example of importing JavaScript function in WebAssembly

Instead of importing JavaScript functions, our implementation targets C++ functions. Listing 5.2 shows our new syntax for calling a native function. Similar to the `import` section from figure 5.1, we create a `native` section which imports the C++ function, assigns to it the alias `$add_i32` and declare its signature. The second part of the code simply calls the imported native function using our new `call_native` instruction. Using this strategy, we can now import as many C++ functions as we want, and execute any of them using the `call_native` instruction. In the rest of this section, we will elaborate on the functionality and usage of this feature.

```
(native "add_i32" (func $add_i32 (param i32 i32) (result i32)))

(func
  (result i32)
  i32.const 1
  i32.const 2
  call_native $add_i32
)
```

Listing 5.2: Example of using `call_native` instruction

Each `native` declaration in a Wasm module corresponds to an object in V8 which carries an external reference for a native function. For simplicity, we require native functions to be implemented in an extension file we created in the V8 engine (`wasm-native-external-refs.cc`). Listing 5.3 shows an example code for a native function inside the extension file. The first part of the code is responsible for declaring the native function. Using C++ macros, we define the function name (`"add_i32"`) and signature (Parameters: `P(kWasmI32, kWasmI32)`, return: `R(kWasmI32)`). Our macros take care of creating the external reference for the native function inside V8, and generate linker code that is executed when a Wasm module is initialized. The linker code allows the Wasm module to populate the `native` object with its corresponding external reference, if and only if the registered name and signature match the ones defined in the Wasm module. For example, in listings 5.2 and 5.3 we can see that declarations of the function name and signature match from both sides, therefore the linker will successfully be able to reference the native function from the WebAssembly module. In case of a declaration mismatch, the engine will report to the user a descriptive message informing them that the native function was not found. The second part of the code of listing 5.3, implements the executable native function. Each native function implemented inside our extension file takes two arguments, a pointer to the WebAssembly linear memory (`Address linear_memory`) and a pointer to a data memory (`Address data`) containing slots for storing the function arguments and return value. For example, our defined native function expects two integer values and returns their sum. Thus, from the `data` pointer, we read two 32-bit integers stored 4 bytes apart, then we sum their values and store the result 4 bytes after the second operand. Reading operands and writing return values from and to a data memory is in fact a common strategy used in V8 to communicate with external references, thus we use this concept and extend it in order to directly specify and execute custom native calls from a WebAssembly module.

```
// Function declaration
// - Create an external reference in V8
// - Name and signature are used by the linker code
#define NATIVE_FUNCTIONS(F, V, P, R) \
  V(F, add_i32, "add_i32", P(kWasmI32, kWasmI32), R(kWasmI32))

// Function implementation
// - Param 1: Linear memory pointer
// - Param 2: Data pointer (operands + return)
void add_i32(Address linear_memory, Address data) {
  int32_t left_op  = ReadI32(data, 0);   // Read integer at offset 0 bytes
  int32_t right_op = ReadI32(data, 4);   // Read integer at offset 4 bytes
  WriteI32(data, 8, left_op + right_op); // Write integer at offset 8 bytes
}
```

Listing 5.3: Example of a native function implementation

5.4 Overhead Comparison

In this section we compare the execution time of a C++ native function implemented using our approach and using Node API. The target function simply performs a matrix multiplication operation on data located inside the Wasm linear memory. Because our interest is measuring the call overhead, we simply consider that both matrix operands are of size 10×10. We run in total of 7 experiments simulating our plan in using this feature for offloading *matrix multiplication operations when training a machine learning model* (chapter 6).

We start measuring the overhead for calling the native function 37,520 times, and we increment this value to the power of 2 for the next 6 experiments. The initial value selected represents a close approximation of the number of matrix multiplication operations performed in a *fully-connected vectorized* machine learning model composed of an input layer, two hidden layers and an output layer, and is trained on 60,000 input data (e.g. MNIST dataset [54]) for 5 epochs. *The reason the number is a close approximation and not the exact value is because of a limitation in our WasmDNN library (chapter 6) requiring the number of input to be a multiple of the batch size, but we ignore this fact in this chapter.* Figure 5.1 illustrates our model and marks the number of matrix multiplications performed in each layer. *The details of the matrix multiplication operations are presented in chapter 6.* In our model, training on each input data *batch* requires a total of 8 matrix multiplications $(3 + 3 + 2)$. Thus, the number of native function calls (num_native_calls) is computed by

multiplying the total number of batches ($\lceil \frac{num\_images}{batch\_size} \rceil$) by the total number of matrix multiplications required for training on one data batch ($(((num\_layers - 2) \times 3) + 2)$) and finally by the number of epochs ($epochs$).

$$num\_native\_calls = \lceil \frac{num\_images}{batch\_size} \rceil \times (((num\_layers - 2) \times 3_{mat.mul.}) + 2_{mat.mul.}) \times epochs$$

Using the above equation, setting $num\_images = 60,000$, $num\_layers = 4$, $epochs = 5$ and $batch\_size = 64$ we obtain $num\_native\_calls = 37,520$. For the 6 other experiments, we compute the number of function calls by updating the batch size to 32, 16, 8, 4, 2 and 1.

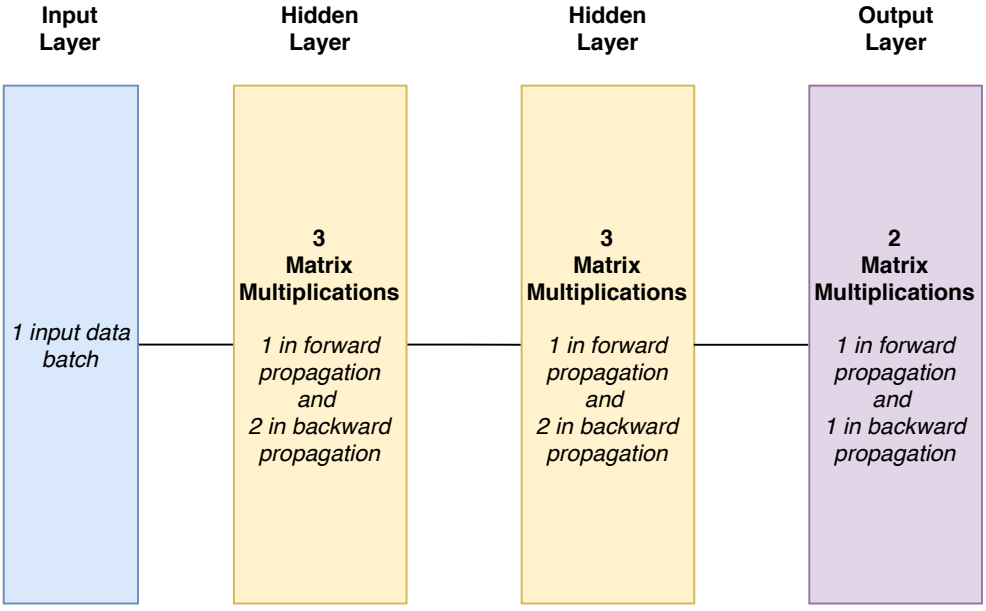

Figure 5.1: Machine learning model composed of 4 layers in total

Figure 5.2 shows the results of our 7 experiments executed on our extended version of Node v12.2.0 (link at the end of chapter 4) running on a desktop computer using an Intel i7-8700K. On average, using `call_native` we obtain a speedup of 1.81× for our matrix multiplication program. We notice that for a model with 2 hidden layers, the call overhead varies from 14.44 ms to 849.626 ms depending on the batch size. In general, as the batch size value decreases, the call overhead increases as more native calls are performed. In addition to the batch size, the depth of the model and the number of epochs also play an important role in affecting those results, as presented in our equation above. Therefore, the benefit obtained from using `call_native` depends entirely on the complexity of the machine learning model. In chapter 6 we use this feature to offload matrix multiplication operations from WebAssembly to a highly optimized C++ linear algebra library.

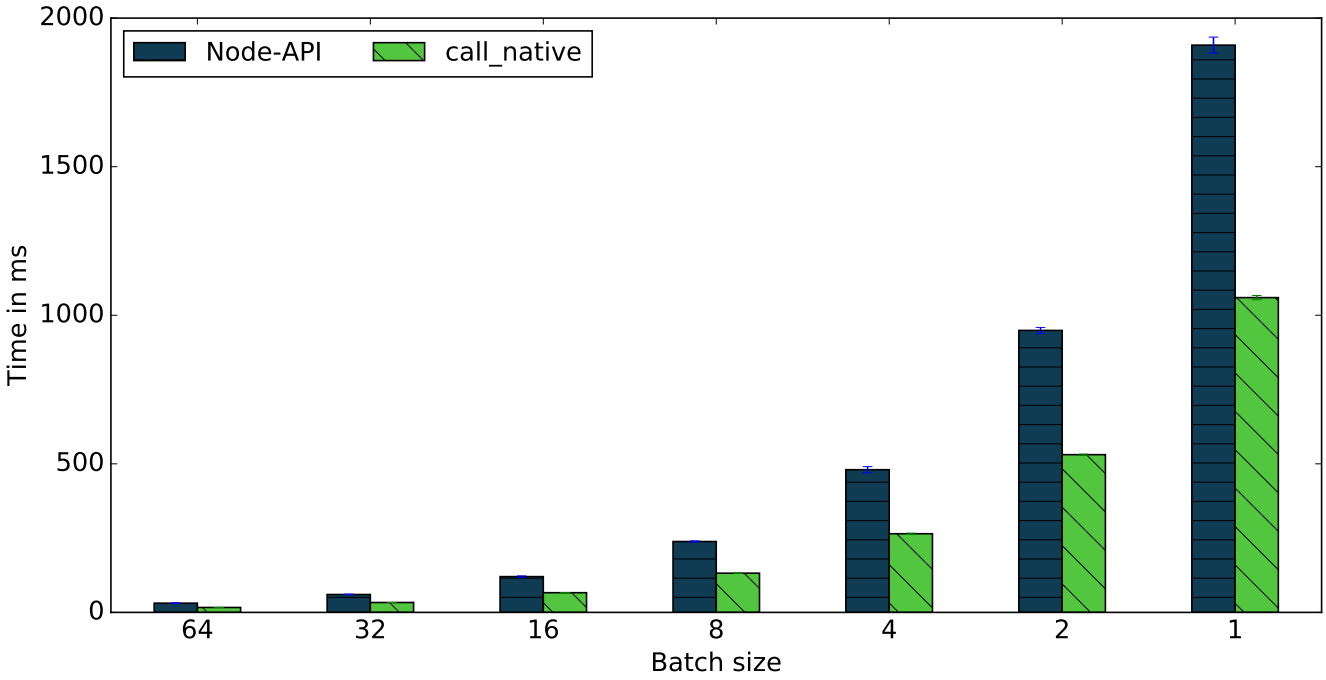

Figure 5.2: Comparison of the total time to execute a native function with respect to the batch size, using `call_native` and Node-API

5.5 Summary

In this chapter we introduced our approach for performing native calls from WebAssembly, allowing developers to directly make calls to C++ functions implemented into the web engine. Compared to code compiled just-in-time, native functions compiled ahead-of-time have the advantage of being more optimized since the compiler can spend a significant amount of time processing the code before generating the final corresponding binary. Although native functions can possibly improve the execution time of a Wasm program, this feature comes at the cost of potentially introducing security threats which is particularly crucial in web engines. As security is not a priority in our study, we carry this concern as part of our future work.

Performing native calls can be achieved in various methods. In this chapter we presented 3 possible options. First we consider using custom instruction for each native function. However, this approach is inconvenient given the amount of non-trivial effort required for implementing each instruction. Second, we study the option of using Node API which is capable of calling native functions in WebAssembly by importing them from JavaScript. Although this option works, calls to native functions generate an overhead imposed by the interface bridging the two languages. In the third option, we integrate our own syntax into

the WebAssembly language in order to provide a more natural method for performing native calls, while also benefiting from the external reference feature in V8 allowing directly calling C++ functions without passing by the JavaScript interface.

Optimizing calls to native functions is motivated by our attempt to accelerate kernel operations for machine learning applications by offloading them to highly optimized functions. Our experiments simulate our usage for native calls in chapter 6, measure the overhead saved by avoiding the JavaScript interface and present the relation between the performance gain and the various parameters of a neural network model.

Chapter 6

WasmDNN

Our final contribution in this thesis is WasmDNN, a deep neural network (DNN) library for generating machine learning models in WebAssembly. The library aims to benefit from the newest technology available in the web in order to accelerate machine learning computations. In addition to developing in WebAssembly, we study optimization opportunities which can further speedup the training and inference process on the browser.

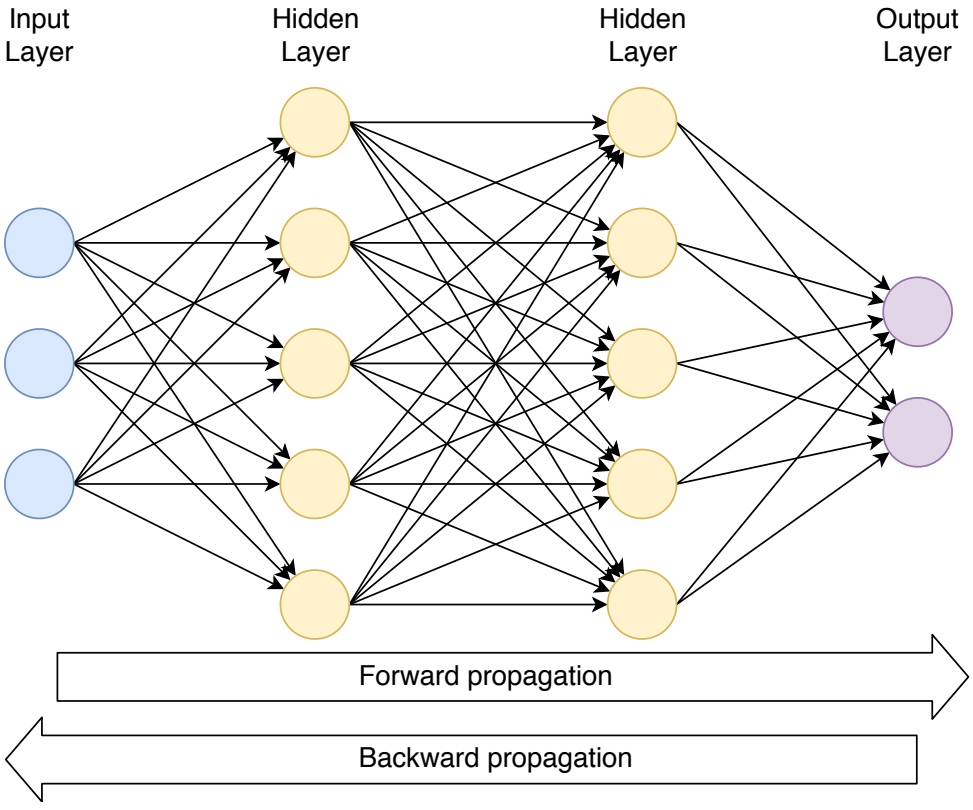

Figure 6.1: Fully connected neural network

In our library, we focus on fully connected models which are powered by the forward and backward propagation algorithms. Figure 6.1 visualizes an example of a fully connected model with an input layer with 3 neurons, 2 hidden layers with 5 neurons each, and an output layer with 2 neurons. The forward propagation algorithm is responsible for flowing the input features into the network layers in order to obtain the output result in the last layer. At the beginning, model predictions might not produce the expected output. To minimize the error of the predictions compared with the expected output, we backpropagate from the output layer in an attempt to adjust the weights connecting consecutive layers. This process is repeated until the model is considered trained and the error value has reached an acceptable rate.

This chapter describes how we implemented neural network models in WebAssembly. In section 6.1 we start by describing the vectorized approach for efficiently applying the forward and backward propagation algorithms. In section 6.2, we take a closer look at the architecture of the network and we briefly discuss *Wasm++*, another library we built in order to facilitate the implementation of this project. In section 6.3, we present the various features supported by our library. In section 6.4 we discuss the limitations of our library and present certain potential solutions. In section 6.5, we demonstrate how we tested the correctness of the Wasm models generated by our library. In section 6.6 we explain some of the optimization we used in order to speedup the execution of our models. Finally, in section 6.7, we perform several experiments comparing our library with other existing ones on the web. In addition, we profile model operations and measure the performance gain obtained by offloading expensive computations to a C++ library using our native calls feature presented in chapter 5.

6.1 Vectorized Implementation

In this section we present the implementation steps of the forward and backward propagation algorithms, without elaborating on their theoretical details. Several resources [55, 56, 57] describe how to implement neural networks from scratch. Proposed implementations can be classified into two categories: iterative approaches treating each model component as an object and using for-loops for traversing the network, and vectorized approaches storing weights and neurons in matrices and using linear algebra operations to propagate the values into the network. From an engineering approach, the former method is more flexible as it is capable of expressing a more descriptive code structure. From a computer science approach, a vectorized implementation opens more doors for optimization opportunities. For the latter reason, we decide to represent the algorithms steps as a set of matrix operations. Our imple-

mentation follows exactly the equations presented in the Neural Network and Deep Learning course [57] offered by Andrew Ng, an adjunct professor at Stanford University.

$$A^{[0]} = Input \tag{6.1}$$

$$Z^{[k]} = W^{[k]} \cdot A^{[k-1]} + b^{[k]} \tag{6.2}$$

$$A^{[k]} = g^{[k]}(Z^{[k]}) \tag{6.3}$$

$$Output = A^{[K]} \tag{6.4}$$

Figure 6.2: Forward propagation equations

Figure 6.2 shows the equations for the forward propagation algorithm. Table 6.1 shows a brief definition of each symbol and explains the kind of operation performed between different elements. The input features are first stored in $A^{[0]}$ where index 0 indicates the input layer (equation 6.1). The next layers will compute its values based on the input from the previous layer (equation 6.2) then apply an *activation function* on the result (equation 6.3). At the end, the output values will be available in $A^{[K]}$ where K is the index of the output layer.

$$dA^{[K]} = \frac{\partial Loss(\hat{y}, y)}{\partial \hat{y}} \tag{6.5}$$

$$dZ^{[k]} = dA^{[k]} * g'^{[k]}(Z^{[k]}) \tag{6.6}$$

$$dW^{[k]} = \frac{1}{m} dZ^{[k]} \cdot A^{[k-1]T} \tag{6.7}$$

$$db^{[k]} = \frac{1}{m} dZ^{[k]} \tag{6.8}$$

$$dA^{[k-1]} = W^{[k]T} \cdot dZ^{[k]} \tag{6.9}$$

Figure 6.3: Backward propagation equations

Figure 6.3 shows the equations for the backward propagation algorithm going in the reverse direction. Table 6.2 defines the new symbols and operations present in the equations. In this algorithm we compute the derivatives starting with $dA^{[K]}$ from the output layer (equation 6.5). Computing dA for the hidden layers depends on the layer that comes after it (equation 6.9). Using the result, and the derivative of the *activation function*, each layer, except the input, computes $dZ^{[k]}$ (equation 6.6) followed by $dW^{[k]}$ (equation 6.7) and $db^{[k]}$

(equation 6.8). To simplify the equations, we omit certain terms and computations such as applying *dropout*, *L1/L2 regularization* and *weight update*.

Expression	Kind	Definition
$A^{[0]}$	Matrix	Matrix storing the user input. Height=Number of input features, Width=Batch size.
$Z^{[k]}$	Matrix	Input matrix for the layer at index k. Height=Number of nodes at layer k, Width=Batch size.
$A^{[k]}$	Matrix	Output matrix for the layer at index k. Height=Number of nodes at layer k, Width=Batch size.
$A^{[K]}$	Matrix	Output matrix for the last layer at index K. Height=Number of output classes, Width=Batch size.
$W^{[k]}$	Matrix	Weight matrix for the layer at index k. Height=Number of nodes at layer k, Width=Number of nodes at layer $k-1$.
$b^{[k]}$	Vector	Bias vector for the layer at index k. Height=Number of nodes at layer k.
$g^{[k]}$	Function	*Activation function* used by the layer at index k.
$Matrix \cdot Matrix$	Operation	Matrix multiplication of the two matrix operands.
$Matrix + Vector$	Operation	Broadcast vector horizontally to match the matrix shape then perform a regular matrix addition.
$f(Matrix)$	Operation	Execute function on every cell of the matrix.

Table 6.1: Forward propagation algorithm symbols

Expression	Kind	Definition
$d\alpha$	Matrix	Derivative of α.
m	Value	Batch size.
$Loss(\hat{y}, y)$	Function	Apply the *loss function* on every input in the mini batch. $\hat{y}$ is the predicted output and y is the true label.
$Matrix * Matrix$	Operation	Element wise multiplication.
$Vector = Matrix$	Operation	Horizontal sum of the matrix. In our example, we multiply the result by $\frac{1}{m}$ to get the average of each summation.

Table 6.2: Backward propagation algorithm symbols

6.2 Architecture

Before starting the implementation of WasmDNN, we focused on building an architecture which could benefit from the WebAssembly system design. In section 6.2.1, we discuss our implementation strategy and choice of intermediate representation used to build our library. In section 6.2.2, we introduce *Wasm++*, a library we developed to facilitate building the intermediate representation. In section 6.2.4, we present a design challenge we encountered when supporting different batch sizes for training, testing and predicting. In section 6.2.5 we introduce *batches-in-memory*, a new feature for managing the amount of data stored inside the WebAssembly linear memory. Finally in section 6.2.6, we discuss how we compiled WasmDNN for the web.

6.2.1 Implementation Strategy and Choice of IR

The objective of WasmDNN is to generate machine learning models in WebAssembly. Each Wasm model is composed of bytecode specific to its parameters configuration. For instance, if the model does not use certain parameters, such as *regularization*, we do not want to generate their corresponding code. This approach allows us to omit several unnecessary checks over model configuration during runtime. Furthermore, knowing the parameter values and input shape in advance allows the generated Wasm to apply certain optimization which we will discuss later in section 6.6.

Targeting WebAssembly directly can be difficult and error prone because of the nature of low-level languages. To solve this problem, we decided to develop WasmDNN such that it generates an intermediate representation (IR) instead of bytecode. Several libraries provide an IR capable of generating WebAssembly including WABT, Binaryen and more recently LLVM. Because we already experimented with WABT and implemented our native calls feature in it, we decided to target its IR.

6.2.2 Wasm++

Building the library on an IR level can still be complicated. Functionalities such as creating a Wasm function, a loop or an if statement are composed of several steps which when used frequently make the code unreadable and hard to maintain. To solve this problem, we built Wasm++, a library to facilitate building WABT IR objects. In section, 6.2.3 we introduce makers and generators, which are functions that simplify the construction of the IR. In section 6.2.3.1, we discuss the memory manager for the WebAssembly linear memory implemented as part of Wasm++. Finally in section 6.2.3.2, we discuss the Wasm++ module

manager which feeds the constructed IR to WABT functions in order to validate it and generate the final Wasm bytecode.

6.2.3 Makers and Generators

The most used functionalities of Wasm++ in WasmDNN are makers and generators. A maker simply creates an atomic functionality such as binary operation, unary operation, load from memory, store to memory and many others. Maker functions are prefixed with the word "Make". Here is an example of a maker function for adding the integers 1 and 2:

```
MakeBinary(Opcode::I32Add, MakeI32Const(1), MakeI32Const(2));
```

The C++ code above is translated to the following WebAssembly (expressed in Wat):

```
i32.const 1
i32.const 2
i32.add
```

Generators are a little more complicated as they build on top of makers to create more complex semantic. Generator functions are prefixed with the word "Generate". Below we present an example of a generator for building a for loop ranging from `i` = 0 to 100 while incrementing by 1:

```
// param 1: Label manager pointer
// param 2: Local variable
// param 3: Range from
// param 4: Range to
// param 5: Increment value
// param 6: Loop return value
// param 7: Body code of the loop
GenerateRangeLoop(label_manager, i, 0, 100, 1, {}, [&](BlockBody* body) {
    // Loop body: e.g. body->Insert(MakeCall(...));
});
```

The code above translates to the following WebAssembly:

```
i32.const 0
set_local $i
loop $loop_1
  // Loop body ...
```

```
get_local $i
i32.const 1
i32.add
tee_local $i
i32.const 100
i32.ne
br_if $loop_1
end
```

The labels assigned to different instructions are generated by our label manager which simply creates a unique identifier on each request. In the example above, we modified the actual labels generated in order to allow the reader to compare between the two versions. The diagram in figure 6.4 summarizes the connection between WABT IR, Maker and Generator discussed in this section.

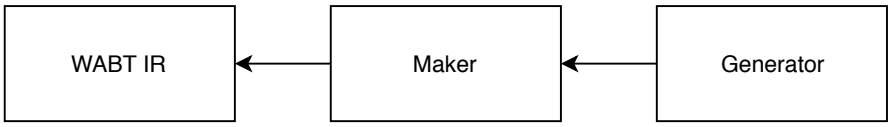

Figure 6.4: Connection between WABT IR, Maker and Generator

6.2.3.1 Memory Manager

Managing the linear memory efficiently could be extremely challenging if done manually. To solve this problem, we build a memory manager inside Wasm++ which uses a first fit approach to allocate linear memory space. In WasmDNN, we use this manager to allocate memory for components inside each layer, such as weight matrices and bias vectors. The memory manager assigns addresses before generating the machine learning Wasm models. Therefore in the latter, matrices and other data structures residing in the linear memory are referenced by constant addresses injected directly inside the different Wasm functions. Because memory is allocated before the WebAssembly is generated, the manager can automatically compute the amount of linear memory pages, of size 64KB each, required by a program. The total number of pages is a mandatory parameters for initializing a WebAssembly module.

6.2.3.2 Module Manager

The module manager is simply the container of the memory and label managers, and hosts all the Wasm functions and other sections living on the global scope of a module. This

manager also uses functionalities provided by the WABT library such as compiling the IR to WebAssembly bytecode or to Wat format, and validating the correctness of the generated WebAssembly.

6.2.4 Batches for Training, Testing and Prediction

Considering that the data can fit in the linear memory, increasing the batch size for testing and predicting can be beneficial since the result computation can be done in parallel (section 6.6). In addition to the execution speed [58], the value of the training batch size can also affect the model accuracy [59]. One of the main challenges we encountered while developing WasmDNN, using a vectorized implementation, is supporting different batch sizes for training, testing and prediction. Different batch sizes require different matrix shapes, and thus matrices used in training, testing and prediction cannot share the same memory slots. To solve this problem, we created three versions of the forward propagation algorithm, each traversing the layers through their corresponding batch size. This decision affects only the forward propagation algorithm architecture, but does not require any changes for the backward propagation algorithm which is only used during the training phase. Figure 6.5 shows matrices for three versions of the forward propagation algorithm (training, testing, and predicting) corresponding to a model composed of an input layer, a hidden layer and an output layer. Matrices Z and A of the same color have the same width, which is equal to the batch size (table 6.1), and they are used in the same version of the forward propagation function. Matrices W and b are independent of the batch size, therefore they are shared among all versions of the function.

6.2.5 Batches in Memory

Before applying the machine learning algorithms on the data, the input needs to be copied into the WebAssembly linear memory. If the batch size is small, then storing a single batch at a time into the linear memory could potentially affect the speed of using the model. On the other hand, fitting the entire data into the linear memory could require a large amount of memory and possibly exceed the maximum memory the module can offer. To solve this problem, we introduce batches-in-memory, a new parameter allowing the user to control the number of batches copied into the linear memory while maintaining a fast overall performance. Since input matrices store entries in a vertical fashion, we need to transpose the data for every batch before copying it. However, interrupting the model frequently to transpose before copying data could become expensive. In section 6.2.5.1, we solve this problem by introducing our data encoder.

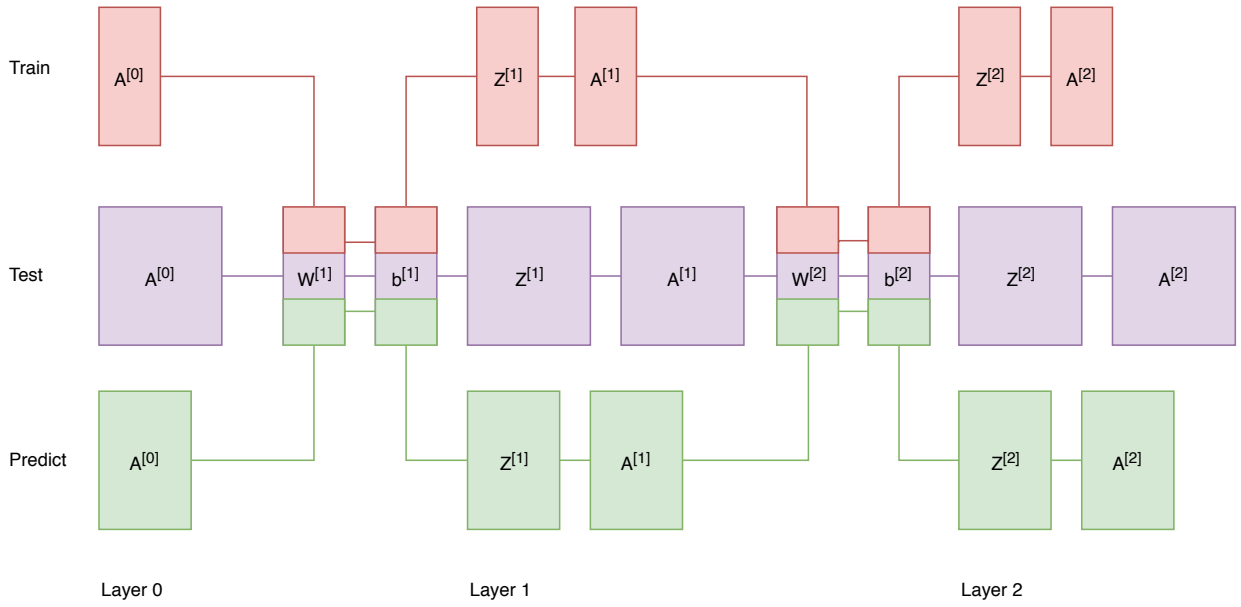

Figure 6.5: Matrices for three versions (Training, Testing and Prediction) of the forward propagation algorithm in a model with a total of three layers

6.2.5.1 Data Encoding

To speed up the copy operation of the batches-in-memory, we first ask the user to encode their input. Our encoder aims at reformatting the input into its expected shape in the linear memory. This process consists of transposing the input by batch size, flattening it and storing it into a typed array (`Float32Array`). Figure 6.6 shows an example of this process. Consider the batch size equal to 3, and the input dataset composed of 6 entries with 2 features each. The first step is transposing the first 3 (batch size value) entries, then the second 3 entries. *Note that we currently require the data to be a multiple of the batch size (section 6.4).* The second step is flattening the 2 dimensional data into a linear array. This approach enables us to quickly copy the batches into the linear memory using the `Float32Array.prototype.set(...)` function which is implemented natively in the web engine.

6.2.6 Project pipeline

WasmDNN is written in C++, but using Emscripten, we were able to provide an alternative web version of the library. Figure 6.7 presents an overview diagram of the WasmDNN library architecture using an example model designed to operate on the MNIST dataset [54]. To configure this model, we can either do this in C++ or in WebAssembly using JavaScript bindings. In the C++ option, the model configuration is done using the WasmDNN library.

```
Batch size: 3
Input:
[[0, 1],
 [2, 3],
 [4, 5],
 [6, 7],
 [8, 9],
 [10, 11]]

Transpose input by batch size:
[[0, 2, 4],
 [1, 3, 5],
 [6, 8, 10],
 [7, 9, 11]]

Flatten result:
data = [0, 2, 4, 1, 3, 5, 6, 8, 10, 7, 9, 11]

Store data in a typed array:
new Float32Array(data)
```

Figure 6.6: Data encoder steps by example

In the JavaScript bindings, the model configuration is done using the WebAssembly version of the library compiled, together with Wasm++ and WABT, using Emscripten. Both model configuration options are equivalent and should be capable of generating identical machine learning Wasm models to train, test and predict on the MNIST dataset. In the rest of this section, we present more details about the various components of this architecture.

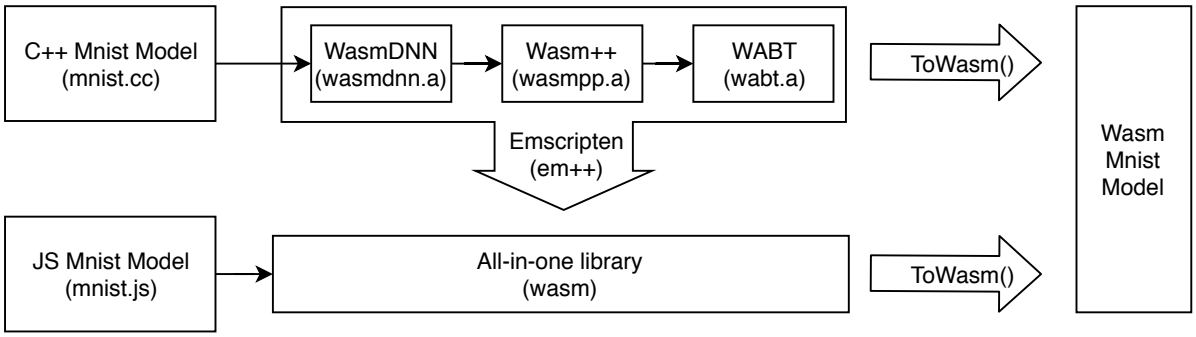

Figure 6.7: Library architecture

Our initial milestone of WasmDNN required the user to configure their model in C++, the language of choice for our Wasm model generator and the WABT library. Listing 6.1 shows an example of using our C++ API to build a model for the MNIST dataset. We

first specify certain options allowing us to configure the bytecode of the generated Wasm model. In this example, we allow the library to generate Single Instruction Multiple Data instructions and we prevent it from injecting the set of instructions used for profiling the forward and backward propagation algorithms (section 6.7.3). Second we configure the network layers with their corresponding number of neurons, *activation function* and *weight initializer*. Third we build the model by passing the remaining model parameters such as the different batch sizes, batches-in-memory and the *loss function*. Finally, we generate the Wasm binary and write it to a file on the system.

```
// Configure the bytecode, activation functions parameters, etc..
ModelOptions options;
options.bytecode_options.use_simd                  = true;
options.bytecode_options.gen_forward_profiling     = false;
options.bytecode_options.gen_backward_profiling    = false;

// Start building the model
Model model(options);
model.SetLayers({
  NewLayer<DenseInputLayer>(28*28)
    ->WeightType(XavierUniform),
  NewLayer<DenseHiddenLayer>(64, model.Builtins().activation.Sigmoid())
    ->WeightType(XavierUniform),
  NewLayer<DenseOutputLayer>(10, model.Builtins().activation.Softmax())
    ->WeightType(LeCunUniform)
});
model.Build(training_batch_size, training_batches_in_memory,
            testing_batch_size, testing_batches_in_memory,
            prediction_batch_size, loss);

// Write Wasm to file
std::ofstream file;
file.open(output_file);
auto data = model.ModuleManager().ToWasm().data;
file << std::string(data.begin(), data.end());
file.close();
```

Listing 6.1: Configuring Wasm model from the C++ API

After the Wasm binary is created, the next step is to load the generated WebAssembly file into the browser, copy the data into the linear memory and finally start interacting with it. However, this can be a complicated process since we don't know at which memory offset

the model is expecting the data, and which functions to execute in order to start training or performing any other action. For this reason, we create a JavaScript wrapper class called `CompiledModel`, which simply calls certain setup functions defined in every model generated by WasmDNN, in order to transfer all the meta information to the JavaScript side. Listing 6.2 shows an example of how this wrapper facilitates interacting with the Wasm model. First, we load the wasm binary (e.g. from a URL address) into a `Uint8Array` and instantiate the WebAssembly module. Once the module is ready, we obtain the `wasm` object and place it into an instance of the `CompiledModel` class. Using the latter, we encode the training data and labels. Finally, we start training by passing the encoded data to the `Train` function which expects certain configuration as a second parameter, including the number of epoch and the learning rate.

```
// Load the generated wasm file from the C++ program
const instance = WebAssembly.instantiate(new Uint8Array(wasm_binary),
                                         CompiledModel.Imports());
instance.then( wasm => {
  // Wrap wasm model
  const compiled_model = new CompiledModel(wasm);

  // Encode training data
  let training = compiled_model.EncodeTrainingData(train_data, train_labels);

  // Start training
  compiled_model.Train(training, {
    epochs: 5,
    learning_rate: 0.02
  });
});
```

Listing 6.2: Example of using CompiledModel to use the generate Wasm model

In the setup described above, the user has to be familiar with C++ and JavaScript. However, constantly switching between the two languages to modify the model in C++ and then use it from JavaScript is inconvenient. To solve this problem, we try compiling the library with its dependencies (Wasm++ and WABT) to WebAssembly using Emscripten. Luckily, the compilation worked without any modification to the project pipeline. With this option, we provided bindings to our C++ API in JavaScript, which resulted in generating the Wasm model as well as executing it possible on the browser. Listing 6.3 shows an example of generating the model and executing it in the same web page. The model configuration part of the code is equivalent to the one described for listing 6.1 but written using the JavaScript

bindings, and the second part of the code is identical to the one explained for listing 6.2.

6.3 Features

In this section we elaborate on the features supported in WasmDNN. In section 6.3.1, we start by listing the activation functions that can be used by the fully connected layers. In total, we implement 6 activation functions operated by the forward propagation algorithm, and their derivatives utilized during the backward propagation (section 6.1). In section 6.3.2, we present the loss functions used to compute the error. The choice of a loss function depends on the activation function used in the output layer, thus we also highlight this dependency. In section 6.3.3, we discuss the different types of weight initializers which play an important role in training the model [60]. Finally in section 6.3.4, we mention the method that we currently use to update the network weights.

6.3.1 Activation Functions

Given an input, an activation function simply transforms this value into an output which is consumed by the next layer or, in the case of the last layer, the output is used as a prediction result. Table 6.3 presents the activation functions [61] supported in WasmDNN. A hidden layer can use one out of 5 available functions, and an output layer can use a sigmoid or a softmax function.

Activation Function	**Layer usage**
Sigmoid	Hidden and Output
Softmax	Output
ReLU	Hidden
LeakyReLU	Hidden
ELU	Hidden
Tanh	Hidden

Table 6.3: Activation functions

6.3.2 Loss Functions

A loss function is used in machine learning models in order to compute the error rate of predictions compared to the real output. In the current version of WasmDNN, we support 3 loss functions [62] presented in table 6.4. The usage of a loss function depends on the activation function of the output layer. Thus, an incompatible combination will result in an error message reported by the library.

```
// Configure the bytecode, activation functions parameters, etc..
let bytecode_options = new ModelBytecodeOptions();
bytecode_options.use_simd                  = true;
bytecode_options.gen_forward_profiling     = false;
bytecode_options.gen_backward_profiling    = false;
let options = new ModelOptions();
options.bytecode_options = bytecode_options;

// Start building the model
let model = new Model(options);
let l0 = new DenseInputLayerDescriptor(28*28);
let l1 = new DenseHiddenLayerDescriptor(config.n, "sigmoid");
l1.SetWeightType("xavier_uniform");
let l2 = new DenseOutputLayerDescriptor(10, "softmax");
l2.SetWeightType("lecun_uniform");

model.AddDenseInputLayer(l0);
model.AddDenseHiddenLayer(l1);
model.AddDenseOutputLayer(l2);


model.Build(training_batch_size, training_batches_in_memory,
            testing_batch_size, testing_batches_in_memory,
            prediction_batch_size, loss);

let buffer = ToUint8Array(model.ToWasm());
let instance = WebAssembly.instantiate(buffer, CompiledModel.Imports());
instance.then( wasm => {
  // Wrap wasm model
  const compiled_model = new CompiledModel(wasm);

  // Encode training data
  let training = compiled_model.EncodeTrainingData(train_data, train_labels);

  // Start training
  compiled_model.Train(training, {
    epochs: 5,
    learning_rate: 0.02
  });
});
```

Listing 6.3: Configuring Wasm model in WebAssembly using the JavaScript API

Loss function	Output Layer Activation Function
Mean Squared Error	Sigmoid
Sigmoid Cross Entropy	Sigmoid
Softmax Cross Entropy	Softmax

Table 6.4: Loss functions

6.3.3 Weight Initialization

In neural networks, weights between layers are initialized randomly following certain conditions and parameters. Table 6.5 shows the weight initialization options, present in WasmDNN, for the hidden and output layers. All our experiments use a Xavier initializer [63], which depends on the incoming and outgoing connections of a layer, and/or LeCun initializer [64], which depends only on the incoming connections of a layer.

Weight initializer	Layer Position
Xavier Uniform	Hidden
Xavier Normal	Hidden
LeCun Uniform	Hidden and Output
LeCun Normal	Hidden and Output
Gaussian	Hidden and Output
Uniform	Hidden and Output
Constant	Hidden and Output

Table 6.5: Weight Initializers

6.3.4 Weights Optimizer

In our current release we support the Stochastic Gradient Descent (SGD) [65] for updating the weights. The latter method is expressed in our library using equation 6.10. The weight matrix $W^{[k]}$ is updated by subtracting from it the weight gradients multiplied by the learning rate α.

$$W^{[k]} = W^{[k]} - \alpha \; dW^{[k]} \tag{6.10}$$

6.3.5 Regularization

Regularization is a common method used in neural networks in order to prevent the model from overfitting to the training data. Table 6.6 lists three regularization techniques [66, 67] supported in WasmDNN. In this thesis, we mainly focus on the *L1* and *L2* regularization as we optimize their implementation using SIMD. Dropout regularization technique consists of

randomly dropping neurons from a network layers. This method is more complex to properly optimize using SIMD as it requires calling the JavaScript random function for each neuron in a dropout layer.

Regularization	**Scope**
Dropout	Input layer and Hidden layers
L1	All weights and bias values
L2	All weights and bias values

Table 6.6: Regularization techniques

6.4 Limitations

A WasmDNN user should be able to benefit from all the features implemented in the library (section 6.3). One known limitation in the current version of the library is the direct relation between the batch size and the input. In our vectorized implementation, we require the data size to be a multiple of the batch size. The reason for the latter is that the model expects the input matrix $A^{[0]}$ to be filled with a number of entries equal to the batch size. However, in a case where the data size is not a multiple of the batch size, the last batch will contain less entries, resulting in a partially filled matrix. A possible future solution for this problem is repeating the training data in order to fill the rest of the matrix.

WasmDNN focuses purely on fully connected neural networks. However, other types of networks such as convolutional neural network (CNN) and recurrent neural network (RNN) can perform better in tasks such as object detection [68] and document classification [69]. It would be interesting to extend this study to account for other types of networks which could equally benefit from the WebAssembly technology.

6.5 Implementation Correctness

A crucial part of developing a library is making sure it works correctly. Implementing the theory of a deep neural network requires simply translating the equations presented in section 6.1 into code. However, an environment where the developer controls every instruction utilized and designs an architecture specific to the platform can introduce many challenges and can be error prone. In section 6.5.1, we discuss our approach to verify the correctness of our matrix operations. In section 6.5.2, we compare the training and testing results of a model generated by WasmDNN and two other models from different machine learning libraries.

6.5.1 Unit Test

Unit test is a common strategy used in software development in order to test program functions. This approach works perfectly for our library since we implement existing algorithms consisting of sets of matrix operations. Thus, if the operations pass the tests, then the algorithm should work in theory. The latter hypothesis is elaborated in the section 6.5.2. In the current release of the library, we implement 32 unit tests covering different matrix operations. Some of the functions are tested more than once since the bytecode generated could differ depending on the shape of the matrix input, for optimization purposes (section 6.4).

6.5.2 Comparison with Other Libraries

Performing unit test was important for testing atomic program functions. But to test the overall correctness of the model and make sure it is capable of learning, we construct and compare three almost identical models for training on a large subset of the MNIST dataset in WasmDNN, ConvNetJS and Tensorflow.js. Our choice of libraries used for this experiment was based on two main factors. First, the libraries should provide training capability. Second, the model must be highly configurable in order for the comparison to be fair. Table 6.7 describes the model configuration. The only option not offered by ConvNetJS is specifying the type of the weight initializer.

Feature	**WasmDNN**	**ConvNetJS**	**Tensorflow.js**
Input features	784	784	784
Output classes	10	10	10
Hidden layer activation function	Sigmoid	Sigmoid	Sigmoid
Output layer activation function	Softmax	Softmax	Softmax
Loss function	Softmax Cross Entropy	Softmax Cross Entropy (from source code)	Softmax Cross Entropy
Weight initializer	LeCun Uniform	N/A	LeCun Uniform
Optimizer	Stochastic Gradient Descent	Stochastic Gradient Descent	Stochastic Gradient Descent
Learning rate	0.02	0.02	0.02
Training batch size	64	64	64
Number of epoch	1	1	1

Table 6.7: Model configuration

After aligning the three models with similar configurations, we run several experiments with different model complexities and report the training loss and the testing accuracy. Experiments described in this chapter follow the same setup used in a paper [30], submitted to the International World Wide Web Conference 2019 [70], comparing several machine learning libraries on the web. For our comparisons, we run 9 experiments with different model structures generated from the combination of 3 variants in the numbers of hidden layers (1, 2 and 4) and 3 variants in the numbers of neurons per layer (64, 128 and 256). For all models, the data consist of the same 49,984 training and 9,984 testing MNIST images fed into the network in the same order. The size of the training and testing data are rounded down to the closest multiple of the batch size value 64 (as per section 6.4).

Figure 6.8 shows the training loss for the 9 experiments. On the x-axis, the number on the left represents the number of hidden layers and the number on the right represents the number of neurons per layer. For 1 hidden layer, we can see that the loss value is at its lowest for all three models, and for 4 hidden layers, the loss value is at its highest. The gradual increase in the training error as the number of layers increases, is most likely caused by the complexity of the network for this task requiring a longer training time for larger models. Overall, the three models have very close training loss values. The highest difference is 0.01 between WasmDNN and Tensorflow.js (1-128) and 0.008 between WasmDNN and ConvNetJS (1-64).

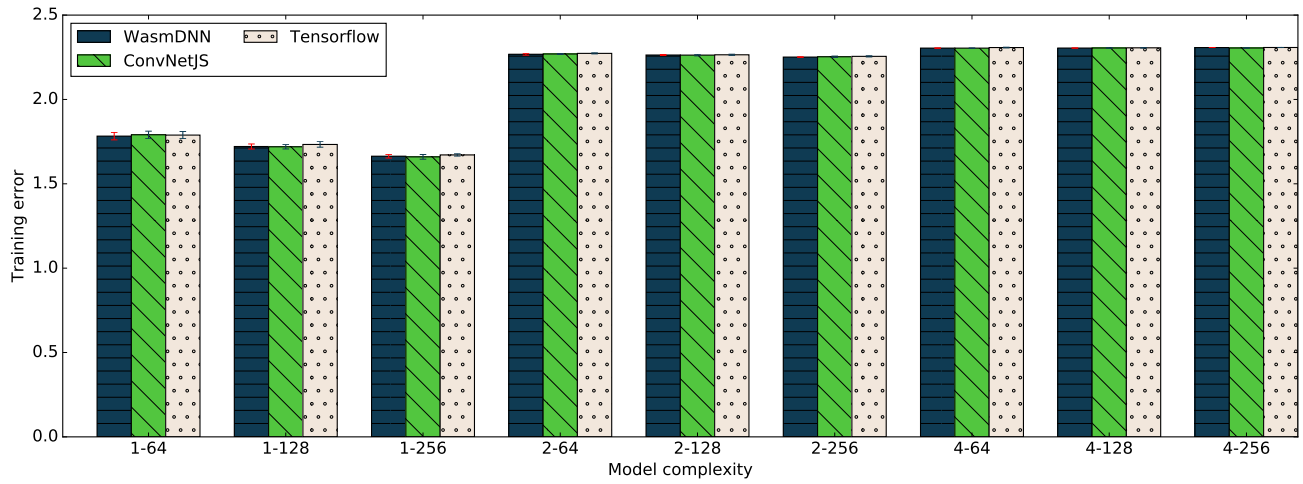

Figure 6.8: Training loss for different model complexities

Figure 6.9 presents the testing accuracy result after the training is complete. We can see that the highest accuracy achieved with our configuration is when using 1 hidden layer. The accuracy drops almost by half, when using 2 hidden layers. For the 4 hidden layers,

all accuracies drop to 0.1. For the three different model depths, we notice an opposite trend for the testing accuracy compared to the training error discussed previously. The testing accuracy gradually decreases as we add more layers, aligning with the fact that training incrementally becomes insufficient to correctly predict on new data entries. The highest testing accuracy difference between WasmDNN and Tensorflow.js is 0.01 (2-256) and between WasmDNN and ConvNetJS is 0.01 (1-64).

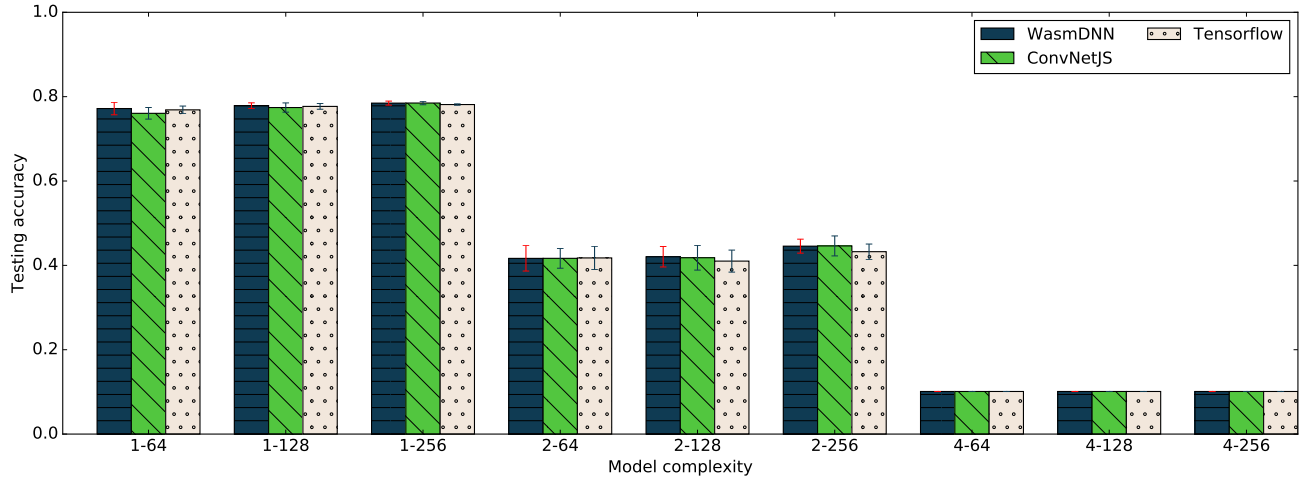

Figure 6.9: Testing accuracy after training on different model complexities

In general, we can see that the training error as well and testing accuracy values are very close between all three models in the 9 experiments. The small differences between the results is expected since each model starts with different weight values.

6.6 Optimization

WasmDNN produces WebAssembly bytecode specific for each model configuration. For instance, instead of generating if-statements that check certain configuration values at run-time, we directly generate the appropriate code depending on the parameter values. For example, if the user sets the training batch size to 1, then in the backward propagation algorithm we do not generate the scalar multiplications (section 6.1 equations 6.7 and 6.8). Another example is L1 and L2 regularization: if their value is set to 0, then we omit their corresponding operations. Applying this strategy for the various parameters contributes to enhancing the overall performance of the model, but is not enough to make a drastic speedup for the execution time. Most of our important optimizations are SIMD related which allow us to perform operations on several data at the same time. In section 6.6.1, we present 5

different SIMD matrix multiplication optimizations used in WasmDNN. In section 6.6.2, we explain SIMD optimizations used for other matrix operations. In section 6.6.3, we highlight and discuss certain differences in the model results observed when we enable our SIMD optimizations.

6.6.1 Matrix Multiplication

Matrix multiplication in different forms occurs in both the forward and backward propagation algorithms. In this section we cover 5 different SIMD optimizations for various matrix conditions and input shapes. Matrix multiplication has been researched for a long time [71, 72] and several highly optimized libraries have already been implemented in various programming languages, such as Eigen in C++ [73] and BLAS in Fortran [74]. For the purpose of experimenting with WebAssembly SIMD and to study its current capabilities, we use the standard algorithms for matrix operations. For instance, matrix multiplication of $n \times n$ is performed in $O(n^3)$, and its WebAssembly SIMD version aims at speeding up the execution time of the same implementation. Before explaining the optimizations, it is important to mention that our matrices are stored row-based in the linear memory. Thus, traversing the matrix left-to-right top-down is simply iterating over the linear memory from the matrix beginning address till the end.

The first matrix multiplication optimization is for non-transposed matrix operands $A \cdot B$. Figure 6.10 shows the steps of this optimization. Each color represents the cells that are involved in the same iteration. The ultimate idea of this SIMD implementation is computing 4 cells in the destination matrix at time. To achieve that, we load (`f32.load`) from the linear memory one value from A and duplicate it to a total of 4 (`f32x4.splat`). From B, we simply load (`v128.load`) 4 consecutive values from the linear memory. We then store (`set_local`) their multiplication (`f32x4.mul`) result into a local accumulator of size 128 bits. Next, we repeat the process for A on the next consecutive memory address with an offset of 4 bytes. For B, we load the 4 consecutive values located in the next row. The address offset of those values can be simply computed by multiplying the width of the matrix by 4 bytes. Again, we multiply the 4 values from both matrices, and add their results to the local accumulator. Finally, we store (`v128.store`) the local accumulator into the corresponding destination matrix in the linear memory. To compute the next 4 destination cells, we repeat the same process again on the same cell from A, but in B we move to the next 4 consecutive values located in the next 16 bytes. In the case of a matrix B that is not a multiple of 4, we fall-back to computing one value at time for the remaining matrix columns.

Matrix multiplications where an operand is transposed are optimized differently. Instead

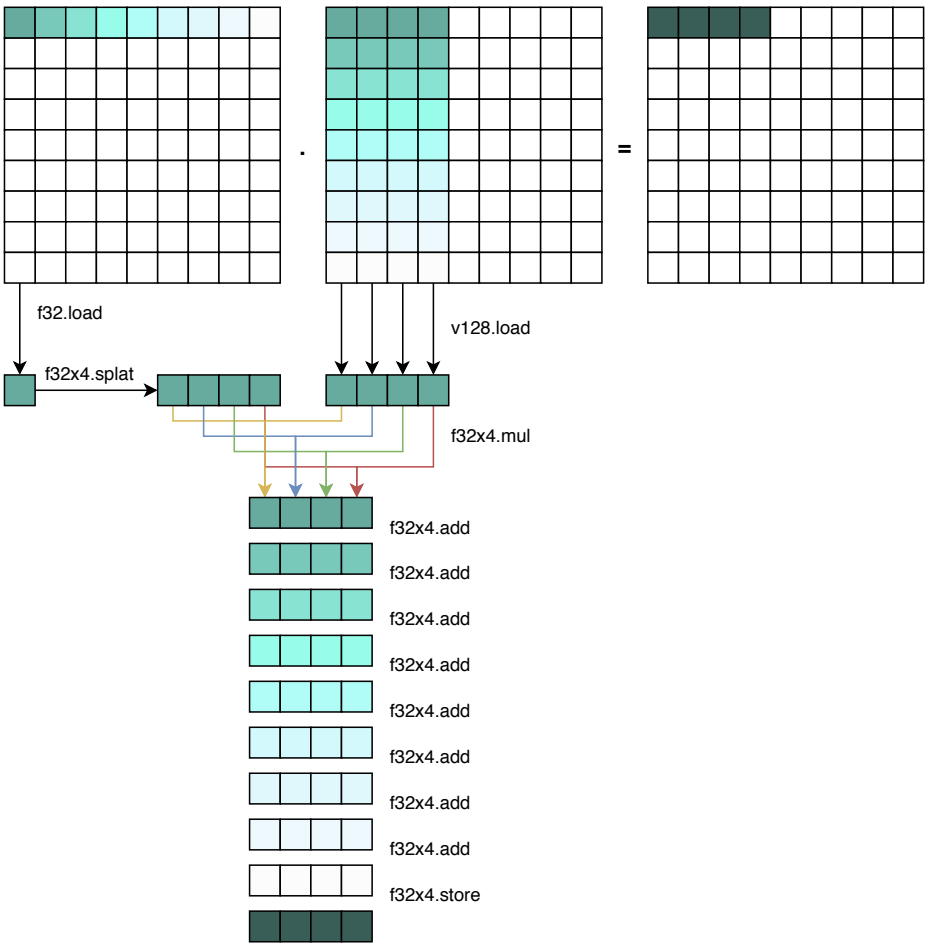

Figure 6.10: Matrix multiplication $A \cdot B$ using WebAssembly SIMD

of transposing a matrix, we implement another multiplication algorithm which adapts to how the matrix is stored in the linear memory. The first example is when the matrix multiplication expects the left operand to be transposed $A^T \cdot B$. Figure 6.11 shows the exact same steps followed in figure 6.10, but for the left matrix operand, we iterate column wise instead of row wise. Thus, after the first iteration, we load from A the value located at an address offset equal to the matrix width multiplied by 4 bytes.

Note that all figures presented in this section show the matrices as stored in the linear memory. Thus, aligning cells left-to-right top-down simply represents the block of consecutive bytes in the linear memory where the matrix values are stored. As a result, figures for matrix multiplication expecting transposed operands do not visualize this transposition.

The algorithm for matrix multiplication expecting a transposed right operand $A \cdot B^T$ is shown in figure 6.12. This algorithm applies the SIMD optimization in a different approach compared to the two previous examples. Because of the nature of a matrix multiplication with a transposed right operand, all the matrix cells involved for computing one destination

value are stored consecutively in the linear memory. This approach allows us to compute one destination cell quickly instead of computing 4 slowly. To apply this method, we load (`v128.load`) 4 values from A from the linear memory. Similarly, we load (`v128.load`) 4 values from B from the linear memory. Then, we multiply (`f32x4.mul`) the 4 values from both matrices and store (`set_local`) their result into a local field of size 128 bits. In the next iteration, we load the next 4 values at an offset address of 16 (4×4 bytes) from A and B. Then, we multiply both values and add (`f32x4.add`) their result to the local field. At the end, using three float addition operations (`f32.add`), we horizontally sum the four 32-bit lanes (`f32x4.extract_lane i` where `i` is 0, 1, 2 or 3) of the 128-bit local field and store (`f32.store`) the result into the destination matrix cell. In the case of an operand matrix with a width size that is not a multiple of 4, we fall-back to multiplying one cell from each matrix at a time, for the remaining columns.

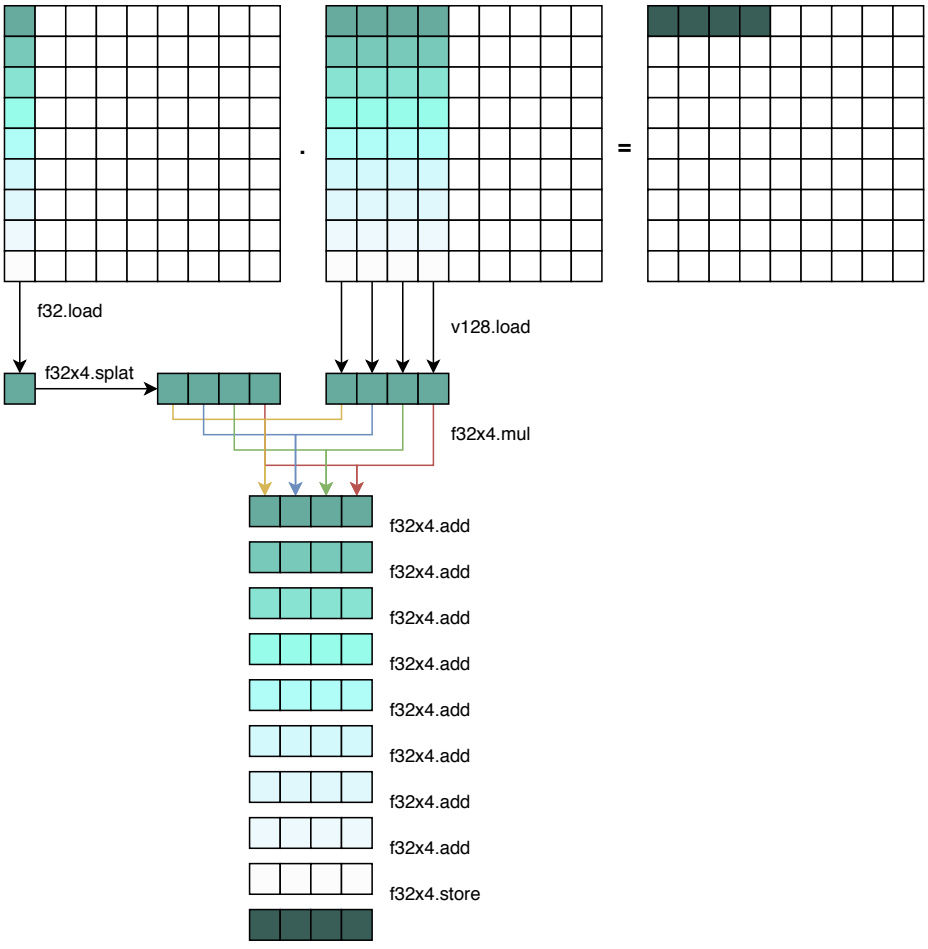

Figure 6.11: Matrix multiplication $A^T \cdot B$ using WebAssembly SIMD

The three matrix multiplications presented above make the computation of large matrices faster. In a deep neural network, the dimensions of some of those matrices depend on the

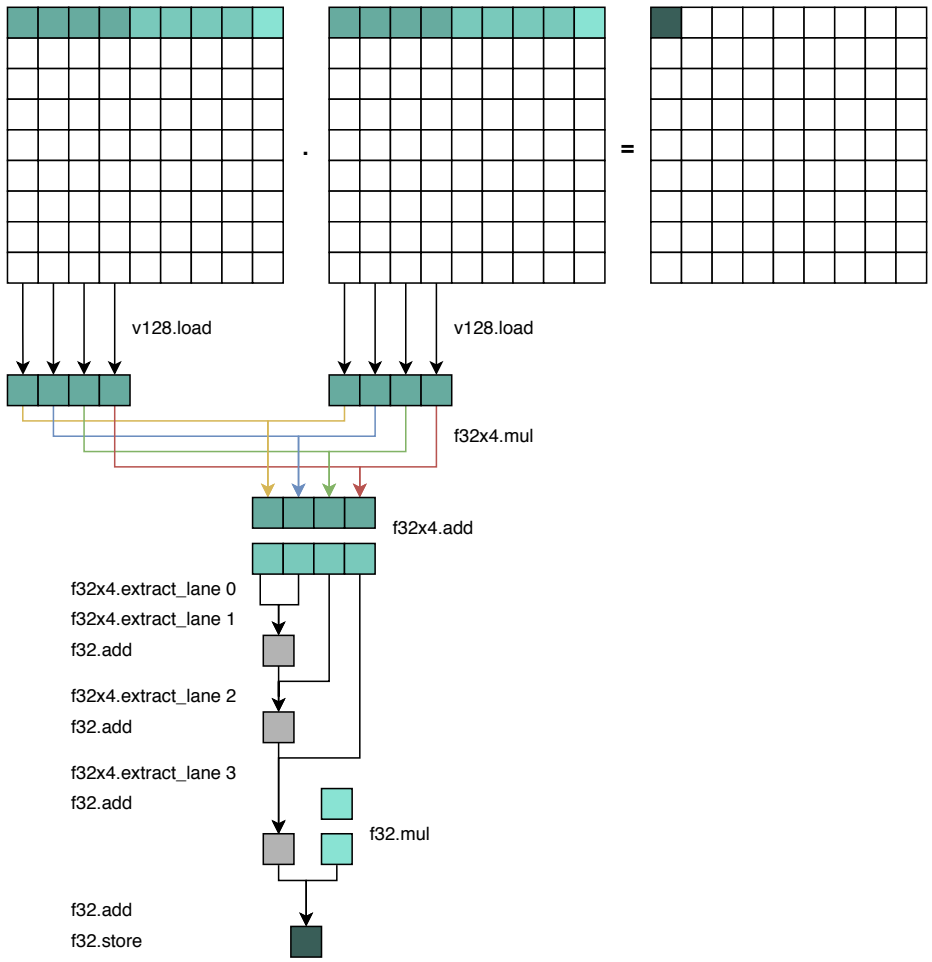

Figure 6.12: Matrix multiplication $A \cdot B^T$ using WebAssembly SIMD

batch size. Thus if a batch size is less than 4, the model would not be able to benefit from those SIMD optimizations. To solve this problem for a batch size equal to 1, we develop two separate optimizations.

In the backward algorithm when multiplying $dZ^{[k]} \cdot A^{[k-1]T}$ (equation 6.7) with a batch size equal to 1, both operands will be vectors (tables 6.1 and 6.2) Thus, we add a special optimization for the case of matrix multiplication where both operands are vectors and the right operand is transposed $A \cdot B^T$. Figure 6.13 shows the SIMD steps used to accelerate the computation time. Because each destination cell is computed with a single multiplication operation, we can improve this computation with a SIMD operation to compute 4 destination cells at a time. To achieve this, we load (`f32.load`) one value from A then copy it into a total of 4 (`f32x4.splat`). From B, we load (`v128.load`) 4 consecutive values from the linear memory. Since the width of B is 1, rows are located 4 bytes apart, allowing us to load multiple row cells with a single instruction. Then, we multiply (`f32x4.mul`) the 4 values from both operands and store (`v128.store`) the result directly into the destination matrix.

In the following iteration, we compute the next 4 destination values by repeating the process on the same cell in A, but in B we load the next 4 consecutive values. Again, we multiply the values from both operands and store them into the destination matrix.

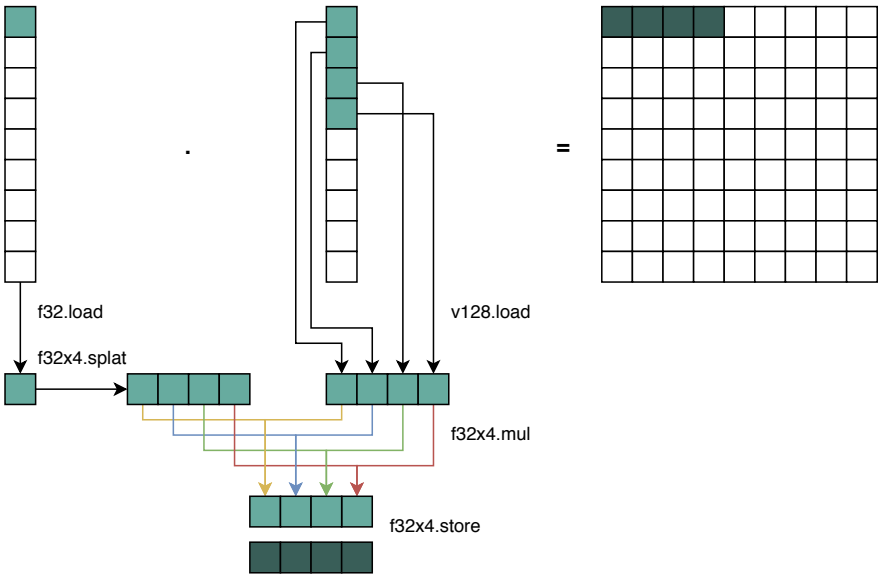

Figure 6.13: Matrix multiplication $A \cdot B^T$ for batch size equal to 1 using WebAssembly SIMD

The second matrix multiplication for which the case of batch size equal to 1 can be improved is in the forward algorithm $W^{[k]} \cdot A^{[k-1]}$ (equation 6.2). Similar to how we optimized the case of matrix multiplication with the right operand transposed (figure 6.12), we compute one destination cell quickly. Figure 6.14 shows the SIMD steps for this operation $A \cdot B$. We first load (`v128.load`) 4 consecutive values from A. Second, we load (`v128.load`) 4 consecutive values from B. Similar to the previous example, rows in vector B are stored 4 bytes apart, allowing us to load multiple row cells in one instruction. Then we multiply (`f32x4.mul`) values from both operands and store (`set_local`) them into a local field of size 128-bits. In the next iteration, we load the next consecutive value from A and B. Again, we multiply them and accumulate the result in the local field. Finally, using 3 float addition operations (`f32.add`), we horizontally sum the 32-bit lanes (`f32x4.extract_lane i` where `i` is 0, 1, 2 or 3) of the 128-bit field and store the result into the destination matrix. In the case of a vector of size that is not a multiple of 4, we fall-back to multiplying 1 cell from each matrix at a time, for the remaining columns.

In the case of a batch size equal to 1, values in the vector are stored consecutively in the linear memory. For batch size equal to 2 or 3, optimization becomes more difficult to apply since values of the same entries are no longer consecutive in memory. Therefore, we recommend the user to use batch size equal to 1 or greater than 3 in order to benefit from our SIMD optimizations.

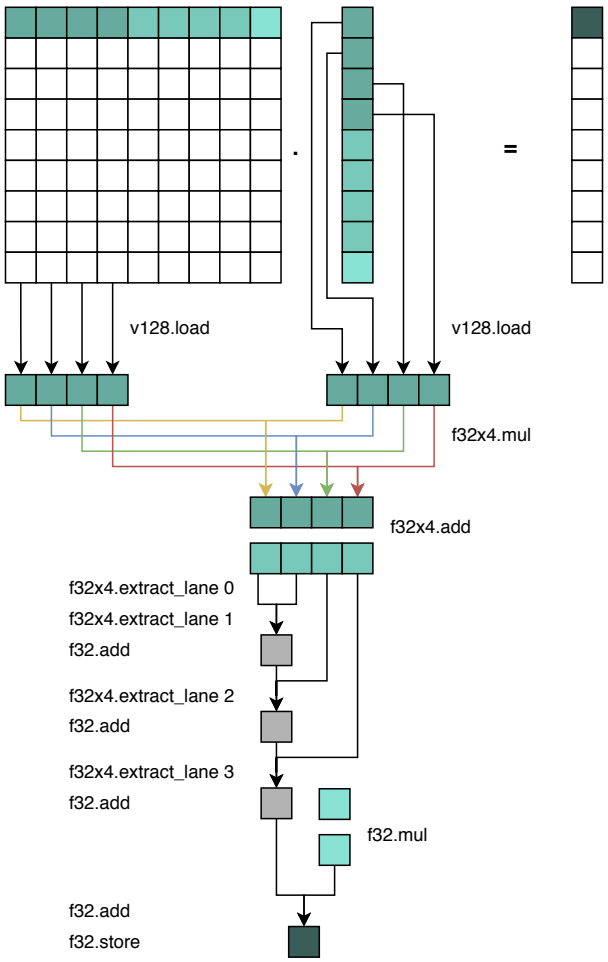

Figure 6.14: Matrix multiplication $A \cdot B$ for batch size equal to 1 using WebAssembly SIMD

6.6.2 Other Matrix Operations

The usage of SIMD instructions proved to be beneficial for matrix multiplication. In a later section (6.7), we present the speed up gained after applying those optimizations. In addition to matrix multiplication, we also used SIMD instructions to perform other matrix related operations such as element wise addition, subtraction and multiplication. The latter operations can be simply applied by iterating over linear memory from each matrix operand's beginning address till the end while performing an `f32x4.add`, `f32x4.sub` or `f32x4.mul` operation respectively. Multiplying all elements in a matrix by a scalar can also be optimized in a similar fashion. Instead of multiplying elements between two matrices, we use `f32x4.splat` to make a total of 4 copies of the scalar and then scale 4 elements at a time.

Optimization opportunities is not limited to binary operations, in fact an interesting case we explored was trying to use SIMD for speeding up L1 regularization computation. Before explaining this method, we highlight how L1 contributes to computing the value of dW in

the backward propagation algorithm. *Note that this equation is a modified version of the backward propagation equation 6.7 from section 6.1 (page 53).*

$$dW^{[k]} = \frac{1}{m}(dZ^{[k]} \cdot A^{[k-1]T} + L1 \times Sign(W^{[k]}))$$

$$Sign(x) = \begin{cases} 1 & \text{for } x \geq 0 \\ -1 & \text{otherwise} \end{cases}$$

The highlighted part simply means that for each value in the weight matrix, if it is negative then the multiplication result is $-L1$, otherwise it is simply $L1$. WebAssembly has an instruction which can easily perform this using `f32.copysign`. The instruction works as follows:

```
get_local $L1              ;; load L1 value into stack
get_local $weight_addr     ;; load weight address into stack
f32.load                   ;; consume top of the stack and
                           ;; load actual weight value
f32.copysign               ;; push L1 into the stack for positive weight value
                           ;; or push -L1 otherwise
```

Unfortunately, a 128-bit version of the `copysign` instruction does not exist and we had to find an alternative method to achieve the same functionality. One approach we considered is reinterpreting `f32` as an `i32`, and querying the sign bit to determine if the number is positive or negative, however the reinterpret instruction is also not available in SIMD. To solve this problem, we used a sequence of four SIMD instructions illustrated in table 6.8 with an example of a 128-bit vector composed of 4 floating numbers [-0.1 0.2 -0.3 0.4]. After applying the four steps, we expect the result to be [`-L1 L1 -L1 L1`]. First, we compare (`f32x4.ge`) the floating numbers with a vector of 4 zeros. If a floating number is greater than or equal to zero, then the instruction would fill the corresponding 32-bits slot in the returned 128-bit vector with ones, otherwise it will set the bits to zero. Thus, a `true` boolean value, represented by all-ones, resolves to -1 when interpreted as a signed integer. Second, we convert (`f32x4.convert_i32x4_s`) the returned boolean vector (values -1 or 0) to a floating vector (values -1.0 or 0.0) in order to match the types of the binary operation performed in the next step. Third, we multiply (`f32x4.mul`) the converted vector by 4 copies of `-2 L1`. Finally, we subtract 4 copies of `L1` from the result of the third step. Following those steps on the initial input, we obtain the correct expected result.

SIMD Instruction	Left operand	Right operand	Return
`f32x4.ge`	[-0.1 0.2 -0.3 0.4]	[0.0 0.0 0.0 0.0]	[0 -1 0 -1]
`f32x4.convert_i32x4_s`	[0 -1 0 -1]		[0.0 -1.0 0.0 -1.0]
`f32x4.mul`	[0.0 -1.0 0.0 -1.0]	[-2L1 -2L1 -2L1 -2L1]	[0.0 2L1 0.0 2L1]
`f32x4.sub`	[0.0 2L1 0.0 2L1]	[L1 L1 L1 L1]	[-L1 L1 -L1 L1]

Table 6.8: Copy-sign operation using SIMD instructions

6.6.3 Optimizations Produce Different Values

During certain experiments where the random seed used for generating weights was a constant and the input was not shuffled, we noticed a small difference between the values produced by a model using SIMD and another identical one that is not using SIMD. After investigating the problem, we realized that the order of adding floating numbers is responsible for producing slightly different results [75]. An example where this case occurs is the SIMD version of the matrix multiplication with a transposed right operand (figure 6.12). In that example, when computing one destination cell, we use a local accumulator which adds the multiplication values of the cells that are 16 (4×4) bytes apart. In a non-SIMD version, the accumulator simply adds the multiplication values of the cells that are 4 bytes apart. To demonstrate how the order of float addition makes a difference, consider the 3 floating number 0.1, 0.3 and 0.5. Adding those values in two different order gives to two different results: $0.1 + 0.3 + 0.5 = 0.8999999761581421$ but $0.5 + 0.3 + 0.1 = 0.9000000357627869$. Optimization of floating point operations can result in subtle errors by naively assuming they respect basic mathematical properties, like associativity. For addition of non-extremal values this introduces a potential error close to round-off, and is thus not a major concern for problems like machine learning, but it can be a more important issue if weights are extremely large or small, where the presence of overflow or underflow at different points in a computation can greatly change results.

6.7 Performance Analysis

The primary motivation of using WebAssembly over JavaScript for our models is the performance that the former has over the latter. In section 6.7.1, we compare the training time between models generated by WasmDNN and other machine learning libraries on the web. In section 6.7.2, similarly we compare the inference time. In section 6.7.3, we study the time distribution in the forward and backward propagation algorithms, and we show how using native calls to highly optimized linear algebra libraries can benefit the execution

speed. Finally in section 6.7.4, we present the speedup obtained by using SIMD for the L1 and L2 regularization.

6.7.1 Training Time

To compare the performance of our library with other existing ones, we reproduce and extend experiments done by the same paper [30] mentioned earlier in section 6.5.2. The experiments compares the training time per batch on the MNIST dataset between Brain.js, Tensorflow.js and ConvNetJS for 12 model complexities. The latter are constructed from a combination of 4 variants in the numbers of hidden layers (1, 2, 4 and 8) and 3 variants in the numbers of neurons per layer (64, 128 and 256). In addition to those three libraries, we add WasmDNN and WasmDNN-SIMD to the chart. The original comparison did not mention the details of the network layers parameters, so we provide our own in table 6.7, except for Brain.js. Configuring the latter was more challenging as, to the best of our knowledge, there is no documentation provided for configuring the loss function, weight initializer and weight optimizer.

Our experiments were performed on Google Chrome version 74 running on a desktop computer using an Intel i7-8700K with an integrated UHD Graphics 630. Figure 6.15 shows the training time per batch for the 12 different model complexities. In general, we were able to reproduce a similar overall trend compared to the results reported by the original paper. One minor difference can be observed for the "8-128" experiment, where our model showed much higher training time per batch for Tensorflow.js (CPU). We believe that the value reported by the original experiment contains a typo since it shows an extremely fast training time for a model that is computationally more complex than the "8-64" model which trained much slower.

Using WebAssembly, we are capable of running the computation consistently faster than all JavaScript libraries with a CPU backend. Using our SIMD optimizations, we observe an impressive speedup. In fact, for the 12 model complexities, the speedup of WasmDNN compared to the fastest JavaScript library [30] (ConvNetJS) ranges between 1.65× to 2.15×, when using SIMD the speedup increases to range between 4.95× to 6.45×. In general, we notice that as the network requires exponentially more computations, the speedup slowly decreases. This behavior can simply be due to the fact that for JavaScript, the optimized machine code is only generated after a function is intensively used (becomes "hot" [2]), and in the long term both JavaScript and WebAssembly will be optimized. Although both machine codes become optimized, WebAssembly allows more optimization opportunities and JavaScript will always suffer from an expensive warmup phase [3]. When comparing

the training time of WasmDNN-SIMD and Tensorflow.js (GPU), we notice that 10 out of 12 times, our library performs better. In fact, the performance advantage of Tensorflow.js (GPU) becomes visible when the model complexity increases. Thus, we suspect that for large models, Tensorflow.js (GPU) will consistently outperform WasmDNN-SIMD. However, as we mentioned in section 2.5.3, if a model becomes very large, it might not be usable on a web browser, since the latter imposes certain restrictions on the host resources. Furthermore, the reason the CPU was capable of training faster in several of our experiments can be caused by the overhead for interacting with the GPU using WebGL.

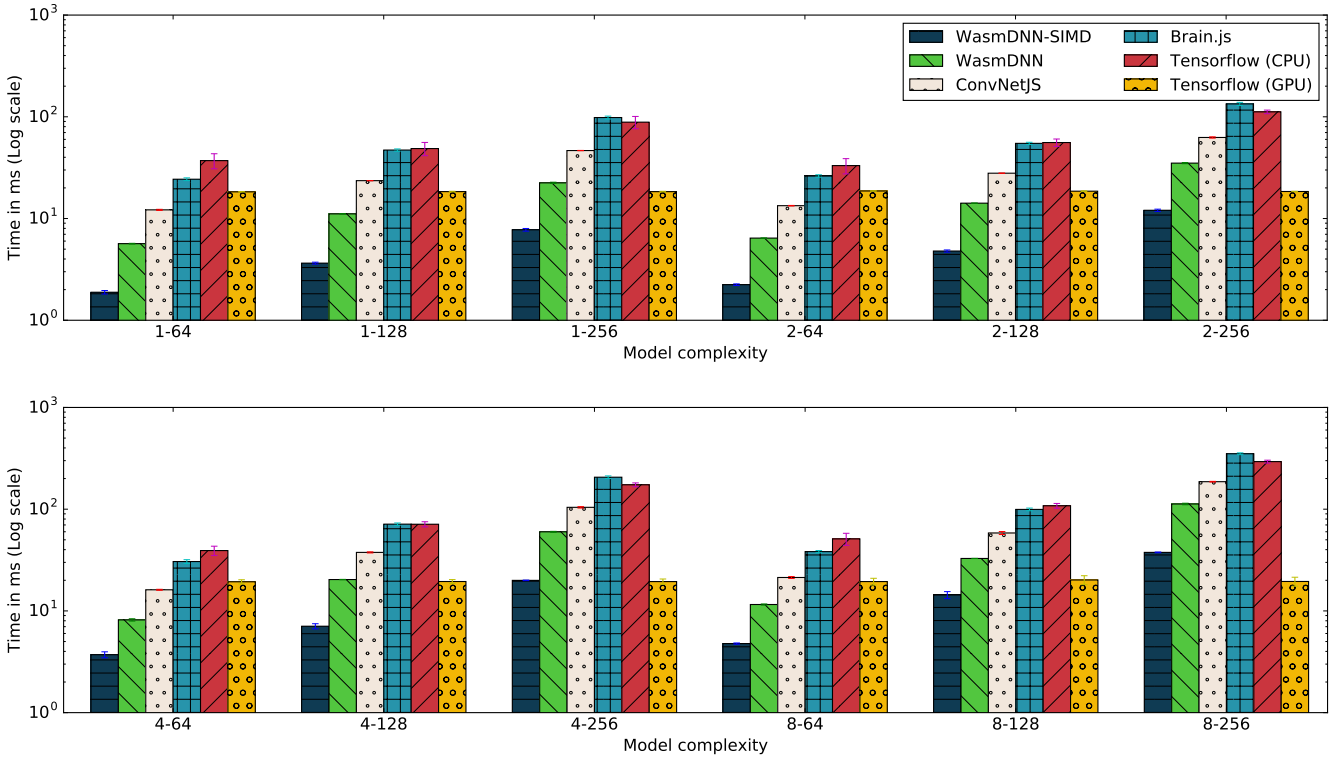

Figure 6.15: Training time per batch in different libraries for different model complexities

6.7.2 Inference Time

In this section, we reproduce the inference experiments from the same paper [30]. However, in order to reduce the complexity of this experiment, we compare WasmDNN only against the fastest JavaScript library (ConvNetJS) and the only other WebAssembly candidate (WebDNN). Readers interested in the inference time for other libraries can refer to the original paper. Furthermore, we add 3 additional experiments for a model with 8 hidden layers and 3 variable number of neurons (64, 128 and 256). We use an identical setup as

the one described for the training experiments (section 6.7.1). WebDNN converts models pre-trained by various popular libraries into executables on the web. For our experiments, we chose Keras for configuring and training the models before converting them into WebDNN format. Moreover, since WebDNN uses Emscripten to generate their Wasm models, we modified the library configuration to enable the experimental SIMD feature using the LLVM backend, by passing `-msimd128` to the compiler. For all the libraries, we measure the average inference time per image by predicting on a total of 9,984 MNIST images. Figure 6.16 shows the results for the 12 experiments. We notice that WasmDNN inference time is faster than the other candidates with and without SIMD. Compared to ConvNetJS, WasmDNN speedup ranges between 1.16× and 1.39×. Using our SIMD optimizations, the speedup increases to range between 3.18× and 4.13×. Compared to the training time experiments, the performance between WasmDNN and ConvNetJS is closer. The latter behavior could be a result of only applying the forward propagation algorithm, during inference, which is not as computationally heavy compared to the backward propagation, hence the benefit is smaller. The more interesting result is WebDNN which using its WebAssembly backend, executes slower than ConvNetJS, even after we modified the build command to enable auto-vectorization (SIMD feature) in the compiler, which gave the library a speedup that ranges between 1.22× and 1.44×. In fact, WebDNN uses Eigen [73], a highly optimized C++ linear algebra library, for performing their matrix multiplication [35], but it is possible that the Wasm generated from the compiler is not making the most out of that library, especially when enabling SIMD.

6.7.3 Profiling the Forward and Backward Propagation Algorithms

In this section, we setup 7 experiments in order to profile the different steps of the forward and backward propagation algorithms with a batch size ranging in 1,2 ,4, 8, 16, 32 and 64. Each of the experiments consists of training on 9,984 images of the MNIST dataset. The full configuration of the used model is presented in table 6.9. Furthermore, for each step we compare the execution time without using SIMD (WasmDNN), with SIMD (WasmDNN-SIMD) and with SIMD and native calls (WasmDNN-SIMD-NC). The latter was used only for accelerating matrix multiplication operations. The C++ functions selected for our native calls are provided by the Eigen library [73], which happens to also be used by Tensorflow for implementing many of their operation kernels [76] and by WebDNN [35] for performing matrix multiplications in their C++ models which are then compiled to WebAssembly using Emscripten.

Figure 6.17 shows the execution time per batch for the 7 experiments. In general, we

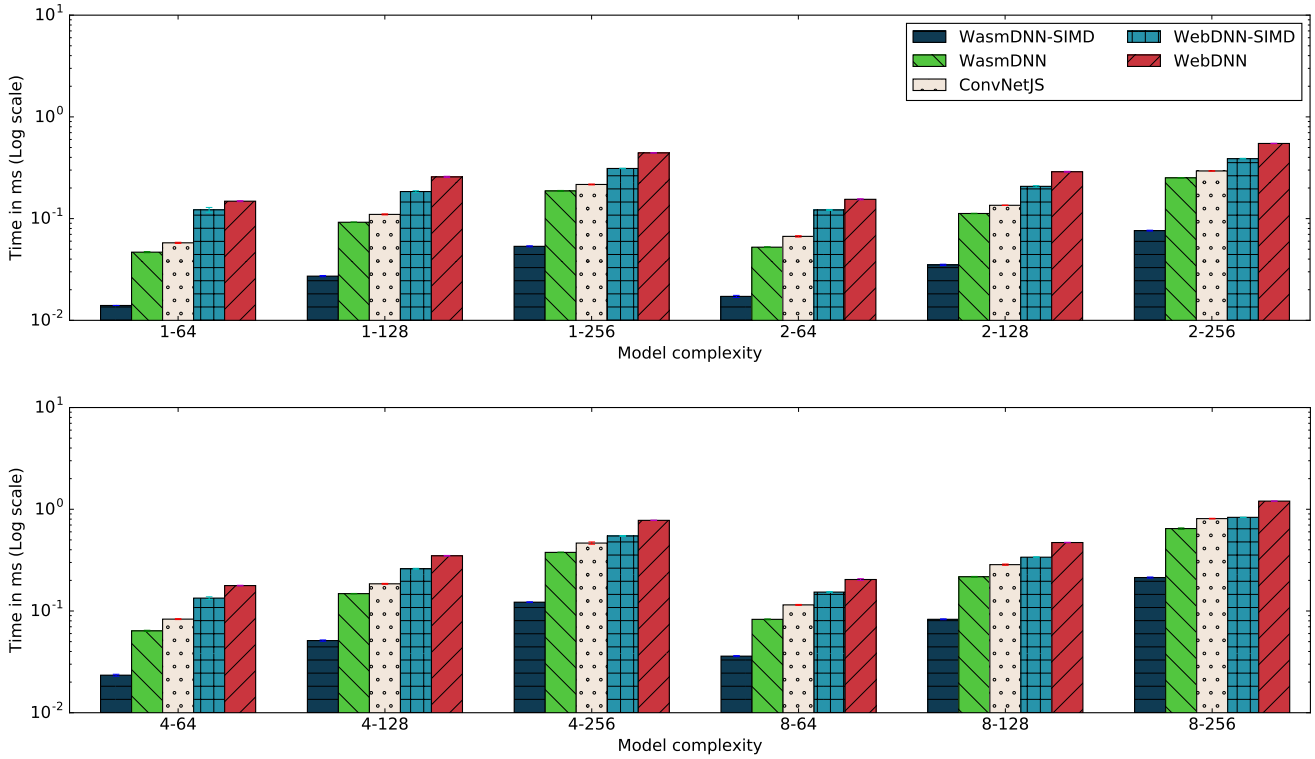

Figure 6.16: Inference time per image in different libraries for various model complexities

notice that as the batch size increases, the execution time per batch also increases as a result of operating on larger matrices. Furthermore, we can see that in most cases, the 3 matrix multiplication operations constitute the largest part of the execution time. As the batch size increases, we notice that our SIMD optimizations enhance the performance for most steps in both the forward and backward propagation algorithms. The only exception where SIMD for the matrix multiplication was not beneficial is when the batch size is equal to 2, which is one of the two batch sizes for which we did not optimize (section 6.6). Out of the 7 experiments, 6 of them benefit from calling native functions for the matrix multiplication operations. Batch size equal to 1 is the only exception where our SIMD optimization for one of the 3 matrix multiplications executes faster. Each of the 9 equations in each experiment contributes to the total execution time per batch. However, the process for computing $dA^{[K]}$ and $b^{[k]}$ have the least performance impact because their operations are simple and the matrix operands involved have relatively small shapes. While repeating each experiment 10 times, several of those iterations reported a value of 0 ms for those two equations, which explains the large error bars shown in the graphs.

The average speedup of the total execution time (sum of all steps) of the 7 experiments using WasmDNN-SIMD over WasmDNN ($Batch\ size : Speedup$) are $(1 : 3.04\times)$, $(2 : 1.186\times)$,

$(4 : 2.25\times)$, $(8 : 2.66\times)$, $(16 : 2.9\times)$, $(32 : 3.11\times)$, $(64 : 3.28\times)$. Using native calls to Eigen library, the speedup, including all the steps, over WasmDNN-SIMD are $(1 : 1.04\times)$, $(2 : 2.54\times)$, $(4 : 1.94\times)$, $(8 : 2.12\times)$, $(16 : 2.31\times)$, $(32 : 2.47\times)$, $(64 : 2.66\times)$. The performance improvement obtained by our SIMD optimizations is expected as a single instruction can process 4 32-bit data units at a time. The further speedup obtained by using native calls is also expected because our matrix multiplication implementation follows the standard $O(n^3)$ algorithm, whereas Eigen has much more optimized algorithms [73]. In addition, the Eigen library was compiled using `g++` with the optimization flag `-O3` enabled, which automatically turns on auto-vectorization [77]. Furthermore, the computer used to perform this experiment has an Intel i7-8700K which supports Advanced Vector Extensions 2 (AVX2) upgrading most 128-bit instructions to 256-bit equivalents [78]. Thus, instead of processing 4 32-bit floating numbers, one instruction is capable of processing 8 32-bit floating numbers at a time.

Feature	Value
Hidden layers	2
Neurons per layer	500
Input features	784
Output classes	10
Hidden layer activation function	Sigmoid
Output layer activation function	Sigmoid
Loss function	Mean Squared Error
Number of epoch	1

Table 6.9: Model configuration

6.7.4 L1/L2 Regularization

In this section we compare the execution time for using L1, L2 and L1/L2 at the same time [79] with and without SIMD. For this experiment, we use an identical setup from section 6.7.3. We presented in section 6.6.2 the equation for $L1$ regularization. The $L2$ regularization is highlighted in this modified version of the equation 6.7 from section 6.1 (page 53).

$$dW^{[k]} = \frac{1}{m}(dZ^{[k]} \cdot A^{[k-1]T} + L2 \times W^{[k]})$$

Figure 6.18 shows the total time spent on L1, L2 and L1/L2 regularization while training on 9,984 MNIST images and varying the batch sizes of the model. In general, we notice

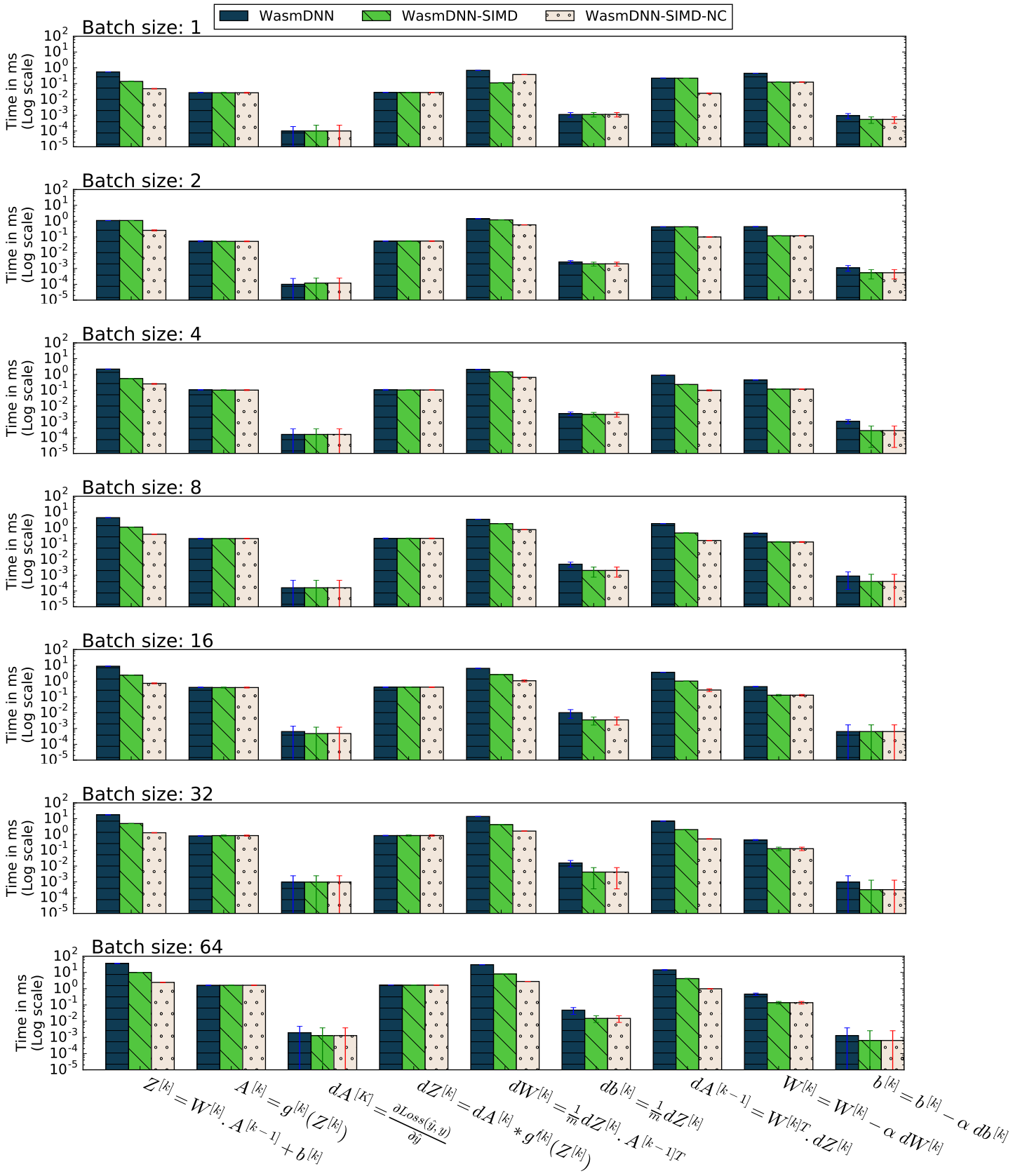

Figure 6.17: Execution time per batch, with 7 different batch sizes, for the various steps of the forward and backward propagation algorithms

that as the batch size increases exponentially, the time spent on regularization decreases exponentially. This trend is expected since a larger batch size means fewer iterations for updating the weights, and thus faster regularization execution time. Moreover, we can see that using our SIMD optimizations, we get consistent performance improvement for all the different experiments. In fact, using SIMD for L1 and L1/L2 we obtain an average speedup of 3.7×, and for L2 we get an average speedup of 3.6×. The reason the execution time spent on L1/L2 is not equal to the sum of the execution time for both L1 and L2 individually, is because instead of performing one regularization after the other, we combine them into the same matrix operation to reduce the number of memory loads and stores.

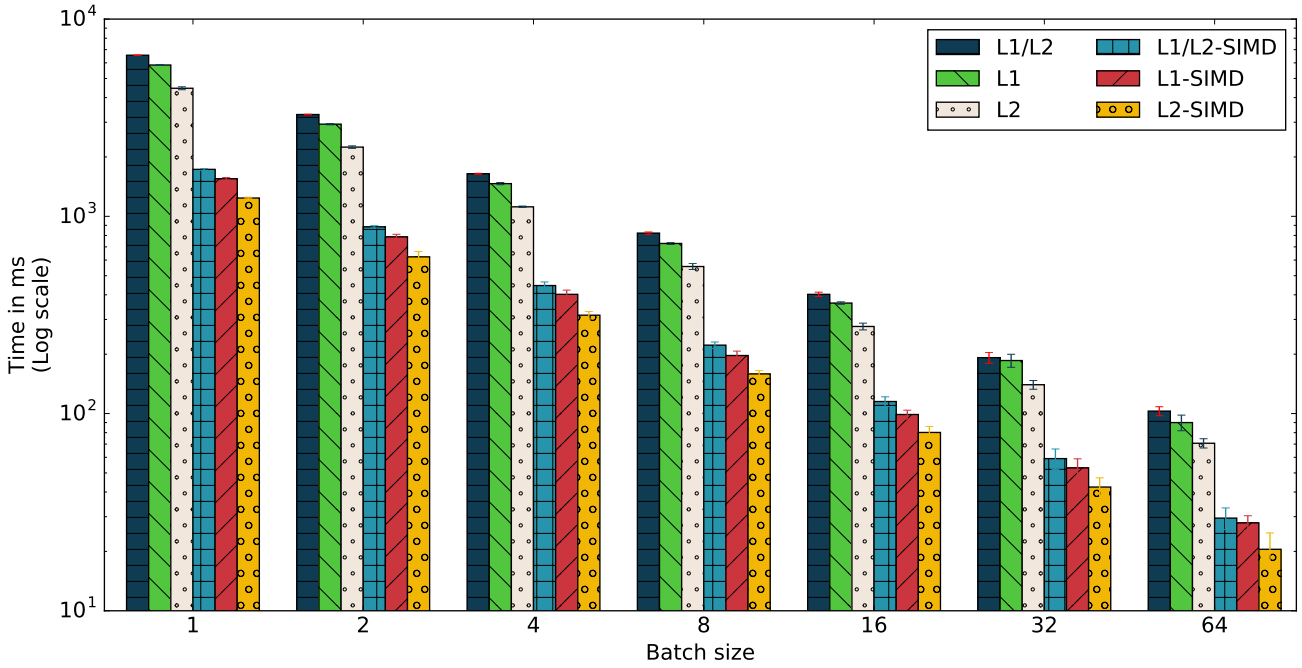

Figure 6.18: Regularization time with and without using SIMD for different batch sizes

The source code for Wasmpp and WasmDNN can be found at:
`https://github.com/Sable/wasmpp`

Demo for using WasmDNN can be found at:
`https://sable.github.io/wasmpp/demo/nnb.html`

6.8 Summary

In this chapter, we presented WasmDNN, a library written in C++ for generating machine learning model in WebAssembly. With the help of Emscripten, we compiled WasmDNN into a web version allowing models to be configured using a JavaScript interface. Our current release supports fully-connected layers for neural network models, and implements a number of features including activation functions, loss functions, weights initializers, weights optimizer and regularization techniques. In order to facilitate our WebAssembly development in WasmDNN, we designed Wasm++ library. The latter aims at providing a simplified interface for constructing WebAssembly modules. Furthermore, Wasm++ offers a set of manager classes reducing the complexity of coding at a low level. On the backend, Wasm++ targets WABT IR allowing modules to benefit from the various tools offered by the WABT library, such as generating Wasm binary files and converting the IR into readable Wat format.

Wasm machine learning models generated by WasmDNN implement a vectorized approach for the forward and backward propagation algorithms powering the various neural network tasks. Because such design translates machine learning algorithms into a set of linear algebra operations, we experimented with the WebAssembly SIMD feature in order to accelerate the execution time of WasmDNN models.

Our performance analysis, comparing WasmDNN with other existing machine learning libraries, showed a clear advantage for using WebAssembly when training and testing models, especially after enabling the SIMD feature. In addition, our experiments with native calls to certain kernel operations, such as matrix multiplications, proved to be beneficial to further improve the execution time of machine learning tasks on the web.

Chapter 7

Related Work

In this section we present related work on the different levels of our contributions which aim to implement and optimize machine learning in WebAssembly. In section 7.1 we discuss certain projects that are currently using and developing tools for WebAssembly. In section 7.2 we list some projects that inspect and extend WebAssembly on the bytecode level in order to achieve various goals. In section 7.3 we present some machine learning libraries that can currently be used on the web. Finally, in section 7.4 we present some work for which SIMD on the web can be beneficial.

7.1 WebAssembly Development

In our research, we chose WebAssembly as a target language for web platforms because of the performance advantage it has over JavaScript when executing computation heavy applications. Today, the most common method for utilizing WebAssembly is compiling C/C++ projects into Wasm binary files using Emscripten. For instance, OpenCV [80], an open source computer vision library, offers a WebAssembly version of their project (OpenCV.js) [13, 81] by compiling their C++ source code using Emscripten. WebDNN, a machine learning library, allows users to execute models on the web using WebAssembly. Before obtaining the Wasm binary, the library first generates a C++ file implementing the model execution functions, and second uses Emscripten to compile the produced code into Wasm output. In our thesis, we use Emscripten to compile WasmDNN library from C++ to WebAssembly. However, we also generate WebAssembly by writing at the bytecode and IR levels and then converting them to Wasm using WABT tools.

In addition to building and utilizing WebAssembly for our programs, we also developed a tool capable of profiling Wasm instructions. Profiling bytecode is an important feature for analyzing Wasm programs, and it is currently also offered by other libraries. Wasabi

[82], a dynamic analysis framework for WebAssembly, provides developers with the option of implementing hooks executed whenever Wasm instructions are processed. This system works by injecting WebAssembly bytecode in between the original program instructions in order to call the analysis framework functions. Using this strategy, the authors provide an example demonstrating how their technology can be used to profile instruction of a Wasm program. Wasabi framework takes a Wasm file as input and outputs an instrumented version of it as well as complementary JavaScript file required as part of the setup. To benefit from the analysis, the user has to load the generated files into a web engine which would then execute the modified Wasm program and make the appropriate calls to the analysis framework functions written in JavaScript. The analysis provided by our tool is limiting to profiling Wasm programs. However, because we implemented our tool as an extension to the WABT library, we benefit from the built-in interpreter, cutting the need for a web setup in order to execute the analysis of the Wasm program.

SEISMIC [83], a detector for unauthorized WebAssembly cryptomining, uses a profiling strategy to interrupt and warn the user about suspicious mining activities in the browser. The system works by injecting WebAssembly bytecode responsible for updating a counter that tracks the usage of instructions of interest. Unlike Wasabi, the modified Wasm does not make calls to JavaScript functions, instead the counter logic is inlined directly after each profiled instructions. The counting results can then be accessed externally through exported accessors. Using our profiler, we can count the occurrence of Wasm instruction, and by sorting the results, we can monitor the most used Wasm instructions. However, because our feature is not browser-based, detecting and interrupting cryptomining cannot be achieved with our current implementation.

7.2 Low-level WebAssembly Manipulation

Part of our contribution described in this thesis is aimed at the low-level bytecode of WebAssembly as well as the machine code generated by the V8 web engine compilers Liftoff and TurboFan. This part of our research aimed at finding optimization opportunities for WebAssembly in the context of machine learning. Working on the low-level layer of WebAssembly has also been done by research projects for various other goals. Constant-Time WebAssembly (CT-Wasm) [84], a strict extension to WebAssembly, aims at making the language secure for implementing cryptography algorithms. This project augments the WebAssembly type system with secret 32-bit and 64-bit integers (`s32` and `s64`) and implements arithmetic instructions that can operate on those types. Furthermore, they add secret linear memories which can contain secret values. The features implemented by this project are

done on the V8 engine. Similar to how we implemented our custom instructions, CT-Wasm project modified the engine frontend for decoding the bytecode and perform type checking, and on the backend for emitting the corresponding machine code.

Memory Safe WebAssembly (MS-Wasm) [85] is a proposal for extending WebAssembly to provide memory safety feature as part of the language. This feature aims at preventing security threats that could result from various exploits such as buffer overflow and use-after-free. To achieve this, the authors introduced a segmented memory which itself is composed of segments. The latter can be utilized using handles which are pointers composed of a `base` address, `offset` value relative to the segment space, `bound` of the segment and `isCorrupted` flag marking a corrupted handle. Handles can ensure memory safety by verifying that data is not overflowing outside its bound and potentially overwriting some adjacent cells. For instance, in our machine learning library we omit such bound checks because first we assume that our memory manager will not overlap two chunks of memory (or segments), and we also assume that our operations such as matrix multiplication do not write outside the matrix bounds. However, we do complement our assumptions with model correctness study including unit tests. The latter strategy works for our usage of the memory manager, however it might not be sufficient for a more complex system where security is a priority. Thus, using a feature such as the one proposed by MS-Wasm could potentially protect the user from various vulnerability threats. In our machine learning application which learns the OR logical operator (chapter 4), we did not have a memory manager and thus it was more prone to buffer overflow. However, we manually segmented the linear memory into smaller segments and used `offset32` instruction to easily iterate over the values within the bounds by fixing the `base` address and incrementing the `offset` value.

In a work studying the performance difference between WebAssembly and native code [86], the authors investigate the performance advantage of native code compiled with `clang`, a C/C++ compiler, over JITed WebAssembly bytecode. The evaluation detailed in their work compares the `x86` machine code for a matrix multiplication in both environments. The algorithm used for the matrix multiplication is the same one used by our machine learning library running with a time complexity of $O(n^3)$. The analysis shows several optimization opportunities that can be applied to V8 and highlights some limitations that are more related to the language design and thus more challenging to solve. In our study, we compare `x86` machine code, at the instruction level, generated by the V8 optimizing compiler TurboFan and the V8 single-pass compiler Liftoff. We also utilize our implementation for the `call_native` instruction to analyze the performance gain by offloading the matrix multiplication computation inside models generated by WasmDNN, to an external implementation provided by a highly optimized linear algebra library.

7.3 Machine Learning Libraries on the Web

In this thesis, we discuss our implementation of WasmDNN a library for generating machine learning models in WebAssembly bytecode. Our library is targeted toward platforms running web engines and benefits from the latest technology and features for accelerating and optimizing machine learning computations. Several other machine learning libraries for the web currently exist including Tensorflow.js [32], ConvNetJS [31], Keras.js [33], Brain.js [34] and WebDNN [35]. All those libraries only support JavaScript for their CPU backend, except for WebDNN which also supports WebAssembly. Keras.js and WebDNN cannot be used for training models on the web, instead they load pre-trained ones and use them for inference. Our library can be used for training as well as inference but currently supports only fully-connected layers. The existing machine learning libraries support more complex network models which can be used for solving a wider scope of tasks. An important feature offered by our library is SIMD optimizations, which are only possible on the web using WebAssembly, making the language even more qualified for being used for machine learning.

7.4 SIMD on the Web

In our machine learning library (chapter 6), we explained how Wasm SIMD feature allowed us to speedup our matrix computations. SIMD in WebAssembly is currently a work in progress and to the best of our knowledge there has not been work reported on the performance gain this experimental feature brought to other WebAssembly libraries. Similar to our results, we expect this library to equally offer acceleration to other projects. For instance, OpenCV.js [81] has already experimented with SIMD on the web for their computer vision algorithms, however their work was based on SIMD.js [87] which today has been taken out in favor of making this feature available in WebAssembly. Work on sparse matrices on the web [88] has also been studied and compared to a native C implementation. The study describes some of the advantages native implementation has over the web and identifies SIMD to be one of the optimization opportunities that could prove beneficial once available in WebAssembly.

Chapter 8

Conclusion and Future work

The most common application of machine learning today are performed offline on local computers or on remote servers. This setup has enabled many developers to use this technology for their various tasks. However, an alternative web-based framework can potentially attract a wider audience and offer new opportunities in enhancing client-side based applications. Many machine learning libraries are today developed for the web. Some of those libraries even support a GPU backend. However, unlike native execution, the GPU interface on the browser has various challenges imposed by its implementation on the browser. Therefore, a CPU option is required for a consistent experience among platforms and to cover a wider scope of web engine embedders with limited resources. For their CPU backend, most of the popular machine learning libraries use JavaScript for executing their models. For a long time, JavaScript has been the only candidate supported by web engines. Being a dynamic language, JavaScript lacked the performance required for training models and predicting on data. To address limitations in computational performance and extend web engines capabilities, researchers from the top browser companies introduced WebAssembly, a low-level bytecode language for the web promising significant performance improvement and more optimization opportunities for web-based applications. In this thesis, we studied the potential of WebAssembly for improving the performance of machine learning tasks on the web. We explored the language on different levels and experimented with its features in an attempt to optimize its usage in the context of machine learning applications.

Our first step in studying the potential of WebAssembly for optimizing and accelerating machine learning tasks on the web was to familiarize ourselves with the language architecture and set of instructions. To acquire this knowledge, we developed an application which required us to inspect the language at the low-level. Our application aimed at providing a terminal based debugging and profiling experience for the developer programming directly in the language. The backend of our application was built on top of the WABT library

which offers a set of tools for dealing with WebAssembly bytecode. Working on this project prepared us to better utilize the language and introduced us to the details of the different components existing in a Wasm program module.

In our second contribution, we experimented with extending the WebAssembly language by introducing our own custom instructions inside the V8 web engine. Implementing custom instructions aimed at benefiting the general use of the language but primarily focused on using machine learning examples as a reference for finding optimization opportunities. Our choice of custom instruction was motivated by our study to the machine code generated by TurboFan, the optimized compiler, and Liftoff, the single-pass compiler. Instructions introduced were developed on different levels of the WebAssembly system. For instance, `offset32` aimed at enhancing the machine code generated by the baseline compiler (Liftoff) whereas `dup` and `swap` manipulated the stack machine. To reduce the overhead generated from calling the imported JavaScript exponential function, which is used intensively in certain machine learning models, we implemented the `exp` instruction and configured it to operate internally. Implementing custom instructions is a challenging task since changes were required in both the WABT library and the V8 engine. Moreover, for certain instructions, the V8 engine required two different implementations, one for its baseline compiler and another for its optimized one. In our experiments, custom instructions did not offer a significant improvement for our programs, however they provided us with a solid grasp on how a web engine decodes and executes WebAssembly instructions. For our future usage of the language, we carried and applied this knowledge to expand the benefits of the language.

Our third contribution explored the possibility of offloading kernel machine learning operations outside of WebAssembly. A main advantage of executing native code, compiled ahead-of-time as opposed to JITted code, is performance. Because the compiler can spend a significant amount of time compiling and optimizing static code on the host hardware, native code can potentially benefit from more optimization opportunities. In fact, certain browsers currently implement functionalities, such as the JavaScript math library, natively into the web engine. In our research, we extended the WebAssembly system capabilities to allow executing C++ functions using a dedicated syntax. Similar to how WebAssembly imports JavaScript functions, our feature allows developers to register their C++ functions and reference them from their Wasm modules. The primary motivation of our design was to eliminate the overhead existing today when calling native functions using alternative options, which require passing first by JavaScript. Our approach of performing native calls allowed us to obtain more precise performance analysis results, when experimenting with this feature, eliminating the overhead caused by the interface utilized for interacting with external functions.

Our final contribution in this thesis is WasmDNN, a library for generating machine learning models in WebAssembly. WasmDNN is developed in C++, but using Emscripten we were able to compile the project to the web, allowing us to generate Wasm machine learning models using WebAssembly. WasmDNN is developed on top of Wasm++, another library we built in order to target the intermediate representation of the WABT library and orchestrate the Wasm memory and module management. Our library currently support fully-connected layers with a number of configuration covering activation functions, loss functions, weight initializers and regularization techniques. The forward and backward propagation algorithms powering our models are implemented using a vectorized approach. This decision aimed at benefiting from the experimental SIMD feature that is currently being developed in WebAssembly. In fact, most of our important optimizations take advantage of this feature in order to accelerate the execution time of matrix based operations. Our results comparing the training time and inference time of machine learning models using WasmDNN and other libraries showed a clear benefit of using WebAssembly for numeric computations, and an impressive speedup when using our SIMD optimizations. Using the native calls feature, we offloaded matrix multiplication operations to a highly optimized C++ linear algebra library, and achieve further speedup. This native call experiment estimated the advantage offered by adopting a linear algebra library into a web engine. In conclusion, we believe that WebAssembly is a promising candidate for optimizing machine learning applications on the web. Compared to JavaScript implementations, our WebAssembly models presented an obvious performance advantages, especially for training tasks. Moreover, using SIMD instructions, neural network algorithms can benefit from a considerable number of optimization opportunities.

Future Work

In the future, we would like to extend this research in order to study the advantages of using WebAssembly for other types of neural networks such as CNN and RNN. The latter can be more beneficial for tasks such as object detection and document classifications. A vectorized implementation of such network types could potentially adopt SIMD strategies in order to accelerate the execution of machine learning tasks.

One of the WebAssembly features that can further improve the performance of WasmDNN is threads. Similar to the SIMD feature, threads are currently a work in progress in the language. In our first implementation, we focused on comparing and analyzing the performance advantages of WebAssembly over JavaScript for machine learning, but in the future adding threads can potentially speedup our linear algebra operations and model execution

process in general.

In later versions of WasmDNN, we plan on supporting more features such as activation functions, loss functions, weight initializers, regularization techniques and weight optimizers. Currently, the library support only the important features which were enough to perform a fair comparison with other existing libraries, in order to measure the execution time for the training and inference tasks, and to evaluate the training error and testing accuracy.

Beside WasmDNN, we would like to experiment with new custom instructions in WebAssembly. For machine learning applications, instructions that can interact with external accelerator could extremely improve the execution time of machine learning applications on web engines. Thus, studying such opportunities allows us to widen our research and cover more environments for running machine learning tasks on the web.

Bibliography

[1] W3Counter: Global Web Stats. https://www.w3counter.com/globalstats.php?year=2019&month=3, 03 2019. (visited on 2019-08-02).

[2] Clemens Hammacher. Liftoff: a new baseline compiler for WebAssembly in V8. https://v8.dev/blog/liftoff, 08 2018. (visited on 2019-08-02).

[3] Andreas Haas, Andreas Rossberg, Derek L. Schuff, Ben L. Titzer, Michael Holman, Dan Gohman, Luke Wagner, Alon Zakai, and JF Bastien. Bringing the Web Up to Speed with WebAssembly. In Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2017, pages 185–200, New York, NY, USA, 2017. ACM.

[4] Alon Zakai. Emscripten: an LLVM-to-JavaScript compiler. In Proceedings of the ACM international conference companion on Object oriented programming systems languages and applications companion, pages 301–312. ACM, 2011.

[5] Chris Lattner and Vikram Adve. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In Proceedings of the 2004 International Symposium on Code Generation and Optimization (CGO'04), Palo Alto, California, Mar 2004.

[6] Mozilla. asm.js. http://asmjs.org/, 03 2013. (visited on 2019-07-20).

[7] LLVM 8.0.0 Release Notes. https://releases.llvm.org/8.0.0/docs/ReleaseNotes.html, 03 2019. (visited on 2019-07-13).

[8] WebAssembly Design Documents. https://github.com/WebAssembly/design/blob/master/BinaryEncoding.md, 04 2015. (visited on 2019-08-02).

[9] The WebAssembly Binary Toolkit. https://github.com/WebAssembly/wabt, 09 2015.

[10] Binaryen. https://github.com/WebAssembly/binaryen, 10 2015. (visited on 2019-08-02).

[11] Emscripten Wiki: Porting Examples and Demos. https://github.com/emscripten-core/emscripten/wiki/Porting-Examples-and-Demos. (visited on 2019-08-02).

[12] Emscripten Fastcomp. https://github.com/emscripten-core/emscripten-fastcomp, 11 2013. (visited on 2019-08-02).

[13] Sajjad Taheri, Alexander Veidenbaum, Alexandru Nicolau, and Mohammad R Haghighat. Opencv. js: Computer vision processing for the web. Univ. California, Irvine, Irvine, CA, USA, Tech. Rep, 2017.

[14] Build OpenCV.js. https://docs.opencv.org/3.4/d4/da1/tutorial_js_setup.html. (visited on 2019-08-02).

[15] Emscripten — Wikipedia, the free encyclopedia. https://en.wikipedia.org/w/index.php?title=Emscripten&oldid=903152560, 2019. (visited on 2019-08-02).

[16] WebAssembly. https://webassembly.org/. (visited on 2019-07-21).

[17] Microsoft edge: Making the web better through more open source collaboration. https://blogs.windows.com/windowsexperience/2018/12/06/microsoft-edge-making-the-web-better-through-more-open-source-collaboration/, 12 2018. (visited on 2019-08-02).

[18] Node.js on ChakraCore. https://github.com/nodejs/node-chakracore, 11 2015. (visited on 2019-07-21).

[19] Spidermonkey — Wikipedia, the free encyclopedia. https://en.wikipedia.org/w/index.php?title=SpiderMonkey&oldid=905970092, 2019. (visited on 2019-07-21).

[20] Brendan eich — Wikipedia, the free encyclopedia. https://en.wikipedia.org/w/index.php?title=Brendan_Eich&oldid=906434172, 2019. (visited on 2019-07-21).

[21] Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. Efficiently computing static single assignment form and the control dependence graph. ACM Trans. Program. Lang. Syst., 13(4):451–490, October 1991.

[22] Lin Clark, Till Schneidereit, and Luke Wagner. WebAssembly's post-MVP future: A cartoon skill tree. https://hacks.mozilla.org/2018/10/webassemblys-post-mvp-future/, 10 2018. (visited on 2019-08-02).

[23] Chromium Code Search. https://cs.chromium.org/. (visited on 2019-08-02).

[24] V8 team. Launching Ignition and TurboFan. https://v8.dev/blog/launching-ignition-and-turbofan, 05 2017. (visited on 2019-08-02).

[25] Ben L. Titzer. Digging into the TurboFan JIT. https://v8.dev/blog/turbofan-jit, 07 2015. (visited on 2019-08-02).

[26] Cliff Click and Keith D. Cooper. Combining analyses, combining optimizations. ACM Trans. Program. Lang. Syst., 17(2):181–196, March 1995.

[27] Java applet — Wikipedia, the free encyclopedia. https://en.wikipedia.org/w/index.php?title=Java_applet&oldid=907848597, 2019. (visited on 2019-08-02).

[28] Lin Clark. Standardizing WASI: A system interface to run WebAssembly outside the web. https://hacks.mozilla.org/2019/03/standardizing-wasi-a-webassembly-system-interface/, 03 2019.

[29] Wasmtime: a WebAssembly Runtime. https://github.com/CraneStation/wasmtime, 08 2017. (visited on 2019-08-02).

[30] Yun Ma, Dongwei Xiang, Shuyu Zheng, Deyu Tian, and Xuanzhe Liu. Moving Deep Learning into Web Browser: How Far Can We Go? CoRR, abs/1901.09388, 2019.

[31] Andrej Karpathy. ConvNetJS. https://cs.stanford.edu/people/karpathy/convnetjs/. (visited on 2019-07-13).

[32] Daniel Smilkov, Nikhil Thorat, Yannick Assogba, Ann Yuan, Nick Kreeger, Ping Yu, Kangyi Zhang, Shanqing Cai, Eric Nielsen, David Soergel, Stan Bileschi, Michael Terry, Charles Nicholson, Sandeep N. Gupta, Sarah Sirajuddin, D. Sculley, Rajat Monga, Greg Corrado, Fernanda B. Viégas, and Martin Wattenberg. Tensorflow.js: Machine learning for the web and beyond. CoRR, abs/1901.05350, 2019.

[33] Leon Chen. Keras.js. https://transcranial.github.io/keras-js. (visited on 2019-07-13).

[34] Robert Plummer. Brain.js. https://github.com/BrainJS/brain.js. (visited on 2019-07-13).

[35] Masatoshi Hidaka, Yuichiro Kikura, Yoshitaka Ushiku, and Tatsuya Harada. Webdnn: Fastest DNN execution framework on web browser. In Proceedings of the 25th ACM International Conference on Multimedia, MM '17, pages 1213–1216, New York, NY, USA, 2017. ACM.

[36] Daniel Smilkov, Shan Carter, D. Sculley, Fernanda B. Viégas, and Martin Wattenberg. Direct-manipulation visualization of deep networks. CoRR, abs/1708.03788, 2017.

[37] Reza Shokri and Vitaly Shmatikov. Privacy-preserving deep learning. In Proceedings of the 22Nd ACM SIGSAC Conference on Computer and Communications Security, CCS '15, pages 1310–1321, New York, NY, USA, 2015. ACM.

[38] Iraklis Leontiadis, Kaoutar Elkhiyaoui, Melek Önen, and Refik Molva. Puda–privacy and unforgeability for data aggregation. In International Conference on Cryptology and Network Security, pages 3–18. Springer, 2015.

[39] Aaron Segal, Antonio Marcedone, Benjamin Kreuter, Daniel Ramage, H. Brendan McMahan, Karn Seth, Keith Bonawitz, Sarvar Patel, and Vladimir Ivanov. Practical secure aggregation for privacy-preserving machine learning. In CCS, 2017.

[40] L. T. Phong, Y. Aono, T. Hayashi, L. Wang, and S. Moriai. Privacy-preserving deep learning via additively homomorphic encryption. IEEE Transactions on Information Forensics and Security, 13(5):1333–1345, May 2018.

[41] H. Brendan McMahan, Eider Moore, Daniel Ramage, and Blaise Agüera y Arcas. Federated learning of deep networks using model averaging. CoRR, abs/1602.05629, 2016.

[42] SIMD: Implementation status. `https://github.com/WebAssembly/simd/blob/master/proposals/simd/ImplementationStatus.md`, 04 2017. (visited on 2019-08-06).

[43] Sapuan Fazli, Saw Matthew, and Cheah Eugene. General-purpose computation on gpus in the browser using gpu.js. Computing in Science & Engineering, 20(1):33, 2018.

[44] Tensorflow for JavaScript: Platform and environment. `https://www.tensorflow.org/js/guide/platform_environment`. (visited on 2019-07-21).

[45] Ncurses. `https://www.gnu.org/software/ncurses/ncurses.html`. (visited on 2019-08-02).

[46] Best ncurses linux console programs. `http://www.etcwiki.org/wiki/Best_ncurses_linux_console_programs`, 04 2014. (visited on 2019-08-02).

[47] Andreas Rossberg. Multi-value Extension. `https://github.com/WebAssembly/multi-value/blob/master/proposals/multi-value/Overview.md#open-questions`, 07 2017. (visited on 2019-07-14).

[48] WebAssembly Specification. http://webassembly.github.io/spec/core/_download/WebAssembly.pdf, 05 2019. (visited on 2019-08-02).

[49] Google. Node: wasm-compiler.cc. https://github.com/nodejs/node/blob/v12.0.0/deps/v8/src/compiler/wasm-compiler.cc#L5585. (visited on 2019-07-16).

[50] John Aycock. A brief history of just-in-time. ACM Comput. Surv., 35(2):97–113, June 2003.

[51] Google. v8/ieee754.cc. https://github.com/v8/v8/blob/master/src/base/ieee754.cc. (visited on 2019-07-23).

[52] Mozilla. mozilla-central: /modules/fdlibm/src/. https://hg.mozilla.org/mozilla-central/file/tip/modules/fdlibm/src. (visited on 07/23/2019).

[53] Google. wasm-external-refs.cc. https://cs.chromium.org/chromium/src/v8/src/wasm/wasm-external-refs.cc?rcl=7550297429f9368192c5ef4570696526e8e0b773. (visited on 2019-07-23).

[54] Yann LeCun, Léon Bottou, Yoshua Bengio, Patrick Haffner, et al. Gradient-based learning applied to document recognition. Proceedings of the IEEE, 86(11):2278–2324, 1998.

[55] Tariq Rashid. Make Your Own Neural Network. CreateSpace Independent Publishing Platform, USA, 1st edition, 2016.

[56] Sebastian Raschka. Python Machine Learning. Packt Publishing, 2015.

[57] Andrew Ng. Neural Networks and Deep Learning. https://www.coursera.org/learn/neural-networks-deep-learning?specialization=deep-learning.

[58] Samuel L. Smith, Pieter-Jan Kindermans, and Quoc V. Le. Don't Decay the Learning Rate, Increase the Batch Size. CoRR, abs/1711.00489, 2017.

[59] Nitish Shirish Keskar, Dheevatsa Mudigere, Jorge Nocedal, Mikhail Smelyanskiy, and Ping Tak Peter Tang. On Large-Batch Training for Deep Learning: Generalization Gap and Sharp Minima. CoRR, abs/1609.04836, 2016.

[60] Yoshua Bengio. Practical recommendations for gradient-based training of deep architectures. CoRR, abs/1206.5533, 2012.

[61] Chigozie Nwankpa, Winifred Ijomah, Anthony Gachagan, and Stephen Marshall. Activation functions: Comparison of trends in practice and research for deep learning. CoRR, abs/1811.03378, 2018.

[62] Sang-Hoon Oh. Statistical analyses of various error functions for pattern classifiers. In International Conference on Hybrid Information Technology, pages 129–133. Springer, 2011.

[63] Xavier Glorot and Yoshua Bengio. Understanding the difficulty of training deep feedforward neural networks. In Proceedings of the thirteenth international conference on artificial intelligence and statistics, pages 249–256, 2010.

[64] Yann LeCun, Léon Bottou, Genevieve B. Orr, and Klaus-Robert Müller. Efficient BackProp. In Neural Networks: Tricks of the Trade, This Book is an Outgrowth of a 1996 NIPS Workshop, pages 9–50, London, UK, UK, 1998. Springer-Verlag.

[65] Léon Bottou. Large-scale machine learning with stochastic gradient descent. In Proceedings of COMPSTAT'2010, pages 177–186. Springer, 2010.

[66] Nitish Srivastava, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. Dropout: A Simple Way to Prevent Neural Networks from Overfitting. Journal of Machine Learning Research, 15:1929–1958, 2014.

[67] Andrew Y. Ng. Feature Selection, L1 vs. L2 Regularization, and Rotational Invariance. In Proceedings of the Twenty-first International Conference on Machine Learning, ICML '04, pages 78–, New York, NY, USA, 2004. ACM.

[68] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. In Advances in neural information processing systems, pages 1097–1105, 2012.

[69] Siwei Lai, Liheng Xu, Kang Liu, and Jun Zhao. Recurrent convolutional neural networks for text classification. In Twenty-ninth AAAI conference on artificial intelligence, 2015.

[70] Ling Liu and Ryen White, editors. WWW '19: The World Wide Web Conference, New York, NY, USA, 2019. ACM.

[71] Volker Strassen. Gaussian elimination is not optimal. Numerische mathematik, 13(4):354–356, 1969.

[72] François Le Gall. Powers of Tensors and Fast Matrix Multiplication. CoRR, abs/1401.7714, 2014.

[73] Gaël Guennebaud, Benoît Jacob, et al. Eigen v3. http://eigen.tuxfamily.org, 2010.

[74] C. L. Lawson, R. J. Hanson, D. R. Kincaid, and F. T. Krogh. Basic Linear Algebra Subprograms for Fortran Usage. ACM Trans. Math. Softw., 5(3):308–323, September 1979.

[75] David Goldberg. What every computer scientist should know about floating-point arithmetic. ACM Computing Surveys (CSUR), 23(1):5–48, 1991.

[76] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, et al. Tensorflow: Large-scale machine learning on heterogeneous distributed systems. arXiv preprint arXiv:1603.04467, 2016.

[77] Using the gnu compiler collection (gcc): Optimize options. `https://gcc.gnu.org/onlinedocs/gcc/Optimize-Options.html`. (visited on 2019-08-02).

[78] Samuel Neves and Jean-Philippe Aumasson. Implementing BLAKE with AVX, AVX2, and XOP. IACR Cryptology ePrint Archive, 2012:275, 2012.

[79] Hui Zou and Trevor Hastie. Regularization and variable selection via the elastic net. Journal of the royal statistical society: series B (statistical methodology), 67(2):301–320, 2005.

[80] G. Bradski. The OpenCV Library. Dr. Dobb's Journal of Software Tools, 2000.

[81] Sajjad Taheri, Alexander Vedienbaum, Alexandru Nicolau, Ningxin Hu, and Mohammad R. Haghighat. Opencv.js: Computer vision processing for the open web platform. In Proceedings of the 9th ACM Multimedia Systems Conference, MMSys '18, pages 478–483, New York, NY, USA, 2018. ACM.

[82] Daniel Lehmann and Michael Pradel. Wasabi: A framework for dynamically analyzing webassembly. In Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS '19, pages 1045–1058, New York, NY, USA, 2019. ACM.

[83] Wenhao Wang, Benjamin Ferrell, Xiaoyang Xu, Kevin W. Hamlen, and Shuang Hao. Seismic: Secure in-lined script monitors for interrupting cryptojacks. In Javier Lopez, Jianying Zhou, and Miguel Soriano, editors, Computer Security, pages 122–142, Cham, 2018. Springer International Publishing.

[84] Conrad Watt, John Renner, Natalie Popescu, Sunjay Cauligi, and Deian Stefan. Ct-wasm: Type-driven secure cryptography for the web ecosystem. CoRR, abs/1808.01348, 2018.

[85] Craig Disselkoen, John Renner, Conrad Watt, Tal Garfinkel, Amit Levy, and Deian Stefan. Position paper: Progressive memory safety for WebAssembly. In Proceedings of the 8th International Workshop on Hardware and Architectural Support for Security and Privacy, HASP '19, pages 4:1–4:8, New York, NY, USA, 2019. ACM.

[86] Abhinav Jangda, Bobby Powers, Arjun Guha, and Emery Berger. Mind the gap: Analyzing the performance of WebAssembly vs. native code. CoRR, abs/1901.09056, 2019.

[87] Peter Jensen, Ivan Jibaja, Ningxin Hu, Dan Gohman, and John McCutchan. Simd in javascript via c++ and emscripten. In Workshop on Programming Models for SIMD/Vector Processing, 2015.

[88] Prabhjot Sandhu, David Herrera, and Laurie Hendren. Sparse matrices on the web: Characterizing the performance and optimal format selection of sparse matrix-vector multiplication in javascript and WebAssembly. In Proceedings of the 15th International Conference on Managed Languages & Runtimes, ManLang '18, pages 6:1–6:13, New York, NY, USA, 2018. ACM.