Machine Learning in the Browser

A thesis presented by
Tomas Reimers
to
The Department of Computer Science

in partial fulfillment of the requirements for the degree of Bachelor of Arts in the subject of Computer Science

Harvard University
Cambridge, Massachusetts
March 2017
Contents

1 Introduction 3
  1.1 Background ............................................. 3
  1.2 Motivation ............................................. 4
    1.2.1 Privacy .......................................... 4
    1.2.2 Unavailable Server ................................. 4
    1.2.3 Simple, Self-Contained Demos ....................... 5
  1.3 Challenges ............................................ 5
    1.3.1 Performance ....................................... 5
    1.3.2 Poor Generality .................................... 7
    1.3.3 Manual Implementation in JavaScript ................. 7

2 The TensorFlow Architecture 7
  2.1 TensorFlow’s API ........................................ 7
  2.2 TensorFlow’s Implementation ................................ 9
  2.3 Portability ............................................ 9

3 Compiling TensorFlow into JavaScript 10
  3.1 Motivation to Compile .................................. 10
  3.2 Background on Emscripten ................................ 10
    3.2.1 Build Process .................................. 12
    3.2.2 Dependencies .................................... 12
    3.2.3 Bitness Assumptions ............................... 13
    3.2.4 Concurrency Model ................................ 13
  3.3 Experiences ........................................... 14

4 Results 15
  4.1 Benchmarks ........................................... 15
  4.2 Library Size .......................................... 16
  4.3 WebAssembly .......................................... 17

5 Developer Experience 17
  5.1 Universal Graph Runner ................................. 17
  5.2 Developing with TFJS ................................ 18
  5.3 Evaluating Image Models ............................... 19

6 Weight Quantization and Compression 21
  6.1 The Need for Weight Compression ......................... 21
  6.2 Lossy Weight Compression ................................ 21

7 Future Work 22
  7.1 Web Technologies on the Horizon ......................... 22
  7.2 Training .............................................. 23

8 Conclusion 24
Abstract

The past decade has seen the rise of rich, dynamic Web applications, and it has also seen the popularization of machine learning. Despite this, we have not seen web applications that evaluate machine learning models in the browser because of technical limitations that make it difficult to do so quickly. I motivate and present a machine learning library for the web, which interoperates with one of the most popular machine learning libraries and is capable of evaluating such models quickly. I further discuss the new classes of features and products enabled by such a library, including: privacy, offline mode, and self-contained demos. The library runs within an order of magnitude of the speed that the native, single-threaded equivalent runs in. For example, one popular image recognition model, Inception, ran in 0.67 seconds natively and as fast as 2.59s on commodity browsers by leveraging emerging web technologies. I also discuss the challenges of building and some of the use cases of such a library.

1 Introduction

1.1 Background

The past decade has seen dramatic growth, in both users and content, on the Web as well as the rise of Web-based applications. While the Web was originally intended to disseminate documents from those with servers to those with clients, the “Web 2.0” paradigm introduced dynamic, interactive content. Recently, this dynamic, interactive content has lead to the development of completely Web-based, interactive applications (including graphics editors, messaging apps, and full office-suites). To support this new wave of applications, the browser has been transformed from a static content viewer into a complex, feature-rich execution platform capable of running untrusted code on any modern machine.

In response to this dramatic growth, new browser APIs have emerged to provide Web apps with access to sensors (e.g. microphone and camera), more powerful hardware-level primitives (e.g. ArrayBuffers and SIMD), and previously impossible computational capabilities (e.g. WebGL and Crypto)\(^1\). These APIs are largely standardized across browsers and devices, and browser vendors are continuously adding support for them \([11]\). A variety of libraries have also proliferated to support the development of richer, more powerful browser applications. They have enabled totally new classes of applications, such as video chat and web-based in-browser games.

Despite these important developments, one technology that has not yet reached the browser is machine learning (ML). Machine learning models, specifically deep artificial neural nets, have recently become popular as a result of: improved algorithms, the proliferation of datasets, and increased computational power. This computational power has been enabled through the creation and availability of large-scale, distributed computational frameworks which can evaluate many mathematical operations rapidly. Deep nets are currently used to

\(^1\)To learn more about emerging browser technologies, visit https://www.html5rocks.com/
classify images, understand natural language, and make recommendations to users [21].

Even though machine learning does not yet run in the browser, it does exist in many Web applications; the machine learning models are just evaluated on the server. Currently, if a Web application requires the use of machine learning, it must send the data to the server, have the server run models against the data, and then return the data to the client. This approach has historically been the standard because evaluating JavaScript-based algorithms at near-native speeds was impossible as browsers did not expose primitives for threads, GPU access, or raw memory allocation.

1.2 Motivation

In this thesis, I motivate and explore the possibility of evaluating machine learning models in the browser quickly, by compiling the TensorFlow library into JavaScript. TensorFlow is one of the fastest growing machine learning libraries. It was developed by Google Brain, and open sourced in late 2015 [3]. I theorize that as machine learning becomes ubiquitous, and more applications move to the cloud, applications will need to leverage ML in offline mode or when there is no server available. I imagine the use case for ML in this context will be primarily the evaluation of pre-trained models. This technology will enable users to have increased privacy, applications to function with an unavailable server, and developers to create simple, self-contained demos.

1.2.1 Privacy

This use case of ML in the browser is the most straightforward. Many companies have, or are, developing their technology to enable some on-device machine learning for mobile platforms. Some use cases include voice recognition, face detection for cameras, and hot word detection for voice-activated home assistants. I suspect that as web-apps take off as their own platform there will be a need to perform machine learning without having to query a server.

1.2.2 Unavailable Server

There are a variety of cases where code running client-side in the web browser may not be able to send information back to a server for processing, and this limits functionality. For example, with the application cache HTML5 spec, Web pages can now define HTML/CSS/JavaScript bundles to run as offline apps, even when no connection to a server is available [30]. One such Web application is Google Docs, which uses this technology to enable offline editing. A second use case are websites that only serve static content from the server—even though that content executes and leads to dynamic interaction on the client-side. These applications are referred to as “unhosted” and lack a server that performs computation [29]. Lastly, in some cases, the server is available, but sending data to the server is unfeasible because the network is slow or throttled.
This technology could be used to accelerate bandwidth-limited applications: one can imagine a use-case where the network is sufficiently slow (or possibly throttled, such as a roaming network) such that sending data (e.g. an image) to be classified on a remote server might actually be slower than running the data against a model that is cached on the client. Whether the client is offline or the network is throttled, being able to run machine learning client-side would enable functionality that is not currently possible.

1.2.3 Simple, Self-Contained Demos

The final application I envision for this technology is to more quickly help people explore machine learning models and easily perform inference on test inputs. Right now, if people want to exhibit a model, they have to download the model file, hope they have the same library (and version numbers) and figure out how to feed example inputs to the model. One could imagine a world where every machine learning model that is open sourced came with a demo website. Right now, this does not happen because no one is willing to pay for the compute, but this technology would remove that requirement.

1.3 Challenges

While one might expect that if machine learning models are just math, and the browser is capable of doing math, then technically it should be possible to run ML models in the browser without issue. Despite this, client-side machine learning is not common, and the few frameworks that exist to do it are not widely used. I examined these frameworks and tried to extract challenges from them: Keras.js\(^2\), a hand-written port of Keras to the browser; MXNet.js\(^3\), a compiled version of MXNet, with little support; ConvNet.js\(^4\), a demo meant to show how one can implement neural nets; and Deep Belief Kit SDK\(^5\), which is specific to image recognition. I include a summary of differences in Table 1.

1.3.1 Performance

Machine learning has increased in popularity over the past few years. One of the reasons for this increasing popularity is increased computational power. While the core idea behind neural networks is still the same, models are much deeper—and computationally intensive—today. For example, of the top nets in ILSVRC 2014\(^6\): the GoogleNet model has approximately 4M parameters, AlexNet has approximately 60M parameters, and VGG has approximately 138M parameters \([19]\). This significant size is the reason why, when running machine learning, it is critical to be able to evaluate activations, convolutions, and other components of the model quickly. Many of the current libraries lack performance.

\(^2\)Keras.js is hosted at https://github.com/transcranial/keras-js
\(^3\)MXNet.js is hosted at https://github.com/dmlc/mxnet.js/
\(^4\)ConvNet.js is hosted at http://cs.stanford.edu/people/karpathy/convnetjs/
\(^5\)DeepBeliefKit SDK is hosted at https://github.com/jetpacapp/DeepBeliefSDK
\(^6\)ILSVRC is one of the most well known image recognition challenges \([25]\)
Table 1: Comparisons of Various Client-Side Machine Learning Libraries. TensorFlow.js is the library I developed.

<table>
<thead>
<tr>
<th>ML Framework</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Keras.js</td>
<td>Keras</td>
</tr>
<tr>
<td></td>
<td>Does not leverage the speed of ASM.js. Also, models must be packaged with their packager into a custom JSON format, and not all op types are supported.</td>
</tr>
<tr>
<td>ConvNet.js</td>
<td>None</td>
</tr>
<tr>
<td></td>
<td>Because this is not backed by a machine learning library, it does not interoperate with any popular libraries. Also, it is handwritten and does not take advantage of ASM.js and has limited op support.</td>
</tr>
<tr>
<td>MXNet.js</td>
<td>MXNet</td>
</tr>
<tr>
<td></td>
<td>The API is different from standard MXNet. Also, weights must be packaged into JSON.</td>
</tr>
<tr>
<td>DeepBeliefKit SDK</td>
<td>DeepBeliefKit</td>
</tr>
<tr>
<td></td>
<td>DeepBeliefKit SDK is highly specialized for image recognition.</td>
</tr>
<tr>
<td>TensorFlow.js*</td>
<td>TensorFlow</td>
</tr>
<tr>
<td></td>
<td>Leverages the speed of ASM.js. Reads standard TensorFlow GraphDef files. The API is identical to the TensorFlow API.</td>
</tr>
</tbody>
</table>
because they don’t leverage newer web technologies (discussed below); for ex-
example, Keras.js runs the Inception image-recognition model at 24s (which runs
at an average of 0.67s in singletthreaded, native TensorFlow), and 12s if the
developer enables GPU optimizations (which prevents the computation from
occurring in a background thread).

1.3.2 Poor Generality

Another set of challenges that developers face when developing with these li-
braries is that the libraries tend to be custom built and do not interoperate with
popular machine learning frameworks. In modern machine learning, a common
pipeline is to “train” a model, and then deploy it for inputs to be evaluated
against the pre-trained model. Cae, Torch, and TensorFlow are all examples
of libraries that save their models in framework-specific model files. Because
trained models are saved in framework-specific model files, it is important for
any library to interact with those files. I would want any library I use to interact
with the model files of at least one of the major machine learning frameworks,
either by directly reading them or by having a compiler that can compile those
into whatever format the library uses.

Additionally, the JavaScript versions of these libraries did not match those
of the libraries that they were modeled o of. For example, even though Keras.js
was modeled after Keras, and MXNet.js was modeled after MXNet, both have
distinct APIs from the libraries they are modeled after. Even though they
function with the same model files, the interface to pass data in is diierent. I
believe this adds an additional layer of friction for developers looking to use the
library and ultimately slows adoption.

1.3.3 Manual Implementation in JavaScript

Finally, we observed that because some of these libraries were manually ported
to JavaScript, they lacked feature parity with their native equivalents or other,
more mature, machine learning libraries. This is because implementing those
features would require large amounts of manual eort. For example, Keras.js
only supports a subset of operations within models when compared to Tensor-
Flow; Keras supports 52, while TensorFlow core (TensorFlow’s mobile model
evaluation platform) includes 216+ operations.

2 The TensorFlow Architecture

2.1 TensorFlow’s API

Given that I wanted to interoperate with a major machine learning library, I
examined many popular libraries. Ultimately I selected TensorFlow, as it is one
of the fastest growing machine learning frameworks with a vibrant open-source
community. TensorFlow was developed and open-sourced by Google, where
it was originally used by the Google Brain team to implement many of their
import tensorflow as tf

y = tf.add(
    tf.mul(
        tf.placeholder(name="m"),
        tf.placeholder(tf.int32, shape=1, name="x")
    ),
    tf.placeholder(name="b")
)

Figure 1: The python to create a basic TensorFlow graph.

machine learning models [3]. At the time of this writing, TensorFlow recently released v1.0 (in February 2017 [23]).

Machine learning frameworks all face a trade-off where they are required to run quickly, while the logic is sufficiently complicated that writing them in C, C++, or any other low-level language is a non-trivial hindrance. For this reason, most machine learning libraries rely on some low level core (some only for math operations, and some to do all of the computation) and allow developers to write the model description and data pipeline in a high level language. To implement this separation of logic, TensorFlow follows the paradigm of many modern machine learning libraries (e.g. Caffe) and constructs a computation graph in Python, which it serializes and evaluates in a C++ core [2].

The computational graph provides a way to serialize the computational steps of machine learning models, allowing for model building in a high level language while providing the computational horsepower of C++. These compute graphs consist of two key components: Ops and Tensors. Ops (or “Operations”) represent nodes in the graph. Tensors represent edges of the graph, and are the multidimensional arrays passed between the ops. For example, the computation:

\[ y = mx + b \]

Can be modeled as Figure 1 in Python.

This graph (also depicted in Figure 2) has five ops: m (a variable), b (a variable), x (a variable), Mul (a standard multiplication), Add (a standard addition). It also has 5 tensors: m:0, b:0, x:0, Add:0, and Mul:0. Tensors are named as OP_NAME:OUTPUT_INDEX (as one op can have multiple outputs, although many have only one).

TensorFlow model graphs must be loaded into sessions (which persist model state, such as variable values). To evaluate a model, one “runs” the session while providing a feed dictionary and an fetch array. The feed dictionary is a mapping from tensor names to tensors, and the fetch array is a list of tensor names. The graph is then evaluated by iterating through all the nodes, evaluating those whose input tensors have already been fed or computed, and ultimately returning the requested nodes to the caller once they have been computed.
2.2 TensorFlow’s Implementation

Underlying the API, TensorFlow is implemented as a C++ core and a Python wrapper. These two systems interact using SWIG and Protobufs, as depicted in Figure 3. SWIG is a popular framework that creates C/C++ bindings into a variety of scripting languages (e.g. JavaScript, PHP, Python, and Ruby) [4]. Protobufs are Google’s custom binary serialization format used for passing data structures between programs, independent of the programming language [10]. The graph is constructed in the Python layer, and sent to the C++ core (using SWIG) where it is then decoded and constructed into a set of Classes. To run a session, the inputs are sent down to the C++ layer, the graph is evaluated, and the results are serialized and returned to the Python layer.

2.3 Portability

One of the appeals of this architecture is that the low level execution core can be compiled to a variety of platforms. TensorFlow also runs in a variety of high-level languages, including Ruby and Java. This variety of bindings is possible because SWIG makes it easy to add additional hooks into the C++ core. Also, TensorFlow runs on a variety of devices because the C++ core can be compiled to target many different CPU architectures (including laptop and mobile CPU architectures).

In modern compilers, source code is transformed by the “compiler frontend”
into an intermediate representation. The intermediate representation is then optimized and further compiled into architecture-specific machine code by the “compiler backend”. In the case of LLVM, which is a compiler infrastructure notably used by Apple, the intermediate representation is called LLVM-IR. This entire pipeline is depicted in Figure 4.

3 Compiling TensorFlow into JavaScript

3.1 Motivation to Compile

Compilers enable the TensorFlow evaluation engine to run on many platforms. There are multiple benefits to compiling TensorFlow over manually implementing its API in JavaScript. First of all, machine learning is still a rapidly evolving field, and keeping up with a library like TensorFlow is a considerable effort. The TensorFlow codebase still changes thousands of lines per week [7]. Second, compiled JavaScript can be readily jitted\(^7\) by the interpreter, and thus runs faster than handwritten JavaScript in many cases (discussed later) [20]. Lastly, the technology landscape on the Web is ever-changing, and if a new technology were to emerge (i.e. WebAssembly, also discussed later), compiling would enable me to readily target that technology.

3.2 Background on Emscripten

Emscripten is an open source project that compiles C/C++ into JavaScript [34]. This was a good candidate to compile TensorFlow because TensorFlow and all of its dependencies are written in C or C++. Emscripten works by extending the LLVM compiler backend to compile LLVM-IR into a subset of JavaScript called ASM.js, see Figure 5. ASM.js is a subset of JavaScript intended as a compilation target. Similar to many CPU architectures, it is performant for

---

\(^7\) JIT-ing, or just in time compilation, is a process by which parts of an interpreted language are compiled at run time to improve performance.
function MeanASMModule(stdlib, ffi, buffer) {
    "use asm";
    function computeMean(a, b) {
        a = a|0;
        b = b|0;
        return +((a + b)|0);
    }
    return { computeMean: computeMean };  
}

At first glance, this code looks fairly confusing and idiomatic; ASM.js uses JavaScript idioms to force types. For example, ASM will express any integer, e.g. x, with the expression x|0. ASM.js uses this syntax because all numbers default to doubles in JS, but by executing a logical or between the variable and 0 the interpreter coerces the variable to an integer (as bitwise operations can only make sense on integers). By prepending expressions with a +, ASM.js can coerce values into doubles, as JavaScript tries to turn the right-hand value into a positive number (and all numbers in JavaScript are doubles).

ASM.js has the added benefit of fast execution times. Some JavaScript engines, such as FireFox’s JavaScript engine, will special case ASM modules, as identified by their "use asm"; statement. These engines will do a static validation of ASM code, and then compile them ahead-of-time (AOT), allowing code to run much faster than if it had to be interpreted [17]. Mozilla, the creator of FireFox, reports that ASM.js can perform certain tasks at only 2x slower than native [33]. Browsers without AOT compilation are still able to run ASM
quickly. For example, benchmarks for Chrome demonstrate continually increasing speeds for ASM.js, because their V8 engine can JIT ASM.js effectively [36]. ASM.js provides these speed benefits because it gives the interpreter information about types through its idiomatic pattern, and because it can apply all of the compilation tricks that typically make compiler-optimized assembly faster than hand-written assembly.

Emscripten is implemented as a drop-in replacement for a configure/make build process. `emcc` has the same interface as `gcc`, `emar` replaces `ar`, and `emconfigure` sets all the standard environment variables and tool paths for configure.

### 3.2.1 Build Process

I used Emscripten to successfully compile TensorFlow into JavaScript. When compiling with Emscripten, the first challenge I faced was the TensorFlow build process. Emscripten was created as a drop-in replacement for GCC and was built to compile projects with Makefiles. TensorFlow does not use a Makefile, rather TensorFlow uses Google’s proprietary build tool, Bazel. While Bazel can theoretically be configured to use a custom compiler tool chain (i.e. `emcc`) using a configuration file called a crosstool, it is poorly documented and would require me to rewrite many of the build files [5]. I did implement a crosstool for Emscripten, but quickly decided it would be unsustainable to maintain.

Fortunately, the TensorFlow project maintains a Makefile for TensorFlow core. TensorFlow core is the the collection of code necessary to evaluate models. The Makefile is specifically targeted for Android and iOS where one is expected to evaluate models, but not necessarily train them. The Makefile produces an archive that can be linked against other TensorFlow projects. I found this to be a good candidate for building, and added a JavaScript target to the Makefile. After doing this, I had a basis for compiling the code base into JavaScript.

### 3.2.2 Dependencies

The next challenge I faced was compiling TensorFlow’s dependencies into JavaScript. Again, a useful metaphor is to think of JavaScript as a new CPU architecture that I am targeting. This means I cannot depend on any libraries that I have not compiled into the target architecture. TensorFlow requires me to compile three additional libraries to run: Protobufs, ZLib, Lib Math. There was prior work on compiling Protobufs into JavaScript [9]; at the time of writing, I had to update the version, but the changes to the library were relatively light (simply a few lines around Protobuf’s concurrency model, discussed in the context of TensorFlow below) [10]. ZLib is maintained as a part of the emscripten-ports

---

8For more details, visit [http://kripken.github.io/emscripten-site/docs/compiling/Building-Projects.html#building-projects](http://kripken.github.io/emscripten-site/docs/compiling/Building-Projects.html#building-projects)

9For example, [https://github.com/invokr/protobuf-emscripten](https://github.com/invokr/protobuf-emscripten)

10My Protobuf fork is hosted at [https://github.com/tomasreimers/protobuf-emscripten](https://github.com/tomasreimers/protobuf-emscripten)
project, and can be readily linked against\textsuperscript{11}. LibMath is included by default with Emscripten because of its ubiquity.

### 3.2.3 Bitness Assumptions

In JavaScript all numbers are implemented as IEEE754 double precision floating points\textsuperscript{12}. Because of this, the JavaScript numbers only have 53 bits of integer precision\textsuperscript{18}. Due to JavaScript’s limited precision, Emscripten can only reliably address a 32-bit address space and can only emulate 32-bit programs. However, CPU-only TensorFlow assumes a 64-bit system. This caused a problem, and I had to modify TensorFlow to run on 32-bit architectures.

TensorFlow is underpinned by Google’s linear algebra library called Eigen\textsuperscript{14}. Internally, Eigen uses a custom type for Indicies and Scalars, \texttt{Eigen::Index}. In order to run on both 32- and 64-bit systems, Eigen defines this type to be \texttt{std::ptrdiff\_t}, which is an \texttt{int64} on 64-bit systems and \texttt{int32} on 32-bit systems\textsuperscript{12}.

Commonly, TensorFlow will do something similar to Figure 7. This will cause the compiler to complain that the code is narrowing the type. In order to overcome this, I had to modify all instances where the code base explicitly defined 64-bit integer types to the semantically correct \texttt{Eigen::Index}. This was actually a fairly small number of changes, and only required the modification of 6 Op files. I include a sample of the patch in Appendix B.

### 3.2.4 Concurrency Model

After correcting the build process, resolving dependencies, and solving the type issues, TensorFlow compiles into a JavaScript library that can be linked against by other C/C++ libraries that are being compiled into JavaScript. Trying to run any of these programs will result in the code hanging as soon as the program attempts to evaluate a session.

This is because Emscripten lacks support for threads, and trying to wait on a thread will cause the program to hang. Currently, JavaScript can create OS-level threads through the use of the WebWorkers API\textsuperscript{31}, but WebWorkers do not share memory. Without shared memory, it is not possible (or at least\textsuperscript{\textsuperscript{11}}\textsuperscript{12})

\begin{verbatim}
int64_t a = 5;
// ...other code...
Eigen::Index b = a;
\end{verbatim}

Figure 7: Type narrowing on 32-bit systems.
not easy) to recreate the POSIX threading model. For this reason, I had to single-thread TensorFlow.

Fortunately, the TensorFlow concurrency model is fairly simple. All of TensorFlow’s parallelization is dependent on the threadpool class, which creates several threads and then dispatches function closures to them to compute\textsuperscript{13}. To single-thread TensorFlow, I modified the thread pool’s \texttt{schedule} method, which queues and dispatches function closures, to simply execute the passed function closure instead of scheduling it. This also meant that all the synchronization primitives work out of the box, because by the time the code reached any barrier method, all the work that had been dispatched must have been completed. To help illustrate this, imagine five closures are dispatched and then a barrier is invoked to wait for the five closures. In this case, all the closures will have finished executing by the time the barrier is invoked. I include the core of the patch in Appendix B.

3.3 Experiences

Completing the above steps creates a library that can be linked against when compiling other C/C++ programs into JavaScript\textsuperscript{14}. The process of discovering how to port TensorFlow to JavaScript was non-trivial because of a lack of debugging tools, poor documentation for building TensorFlow for novel architectures, and intricacy of the system I was porting. The process of porting involved rewriting libraries that already existed, trying various approaches that I would later simplify/consolidate, and thinking that various implementations were correct until a bug manifested much later. The best advice I can give to future developers would be to create integration tests from the beginning whose results can be manually verified, to understand the system at a high-level before implementing anything (including understanding the tooling around the system), and to look into compilation for resource-constrained environments (e.g. mobile) for inspiration and guidance.

I believe that these results do not pose a significant change to the code base and will hopefully lead to future compatibility with minimal changes. Thus far, I have bumped the version on every release from v0.10.0, v0.11.0, v0.12.0, and v1.0.0. Anecdotally, each bump has taken me less than 15 minutes and usually relies on a changing a new type or resolving a merge conflict. While I cannot predict how the code base will change, I think that my changes will generalize to future versions.

I have to maintain the Emscripten compiled protobufs, which is simple. My few changes to the TensorFlow codebase include modifying a few call sites to replace \texttt{int64} with \texttt{Eigen::Index}; this change should probably be merged into master as it is semantically more correct. In addition, I add a few lines inside a C preprocessor macro to the threadpool schedule method. Lastly, my Makefile

\textsuperscript{13}Implementation is at https://github.com/tensorflow/tensorflow/blob/ff2da4156bcde26a253fd98198c6f19389df921d9/tensorflow/core/lib/core/threadpool.cc

\textsuperscript{14}My modified TensorFlow is at https://github.com/tomasreimers/tensorflow-emscripten
Table 2: Average runtimes in seconds over 100 trials running on a Macbook Pro 2015 with an 2.9GHz i5 processor. The number in parenthesis represents the ratio when compared to singlethreaded C.

<table>
<thead>
<tr>
<th></th>
<th>MNIST</th>
<th>Inception v3</th>
</tr>
</thead>
<tbody>
<tr>
<td>C (multithreaded, cpu-only)</td>
<td>0.00340323 (0.91)</td>
<td>0.374765 (0.56)</td>
</tr>
<tr>
<td>C (singlethreaded, cpu-only)</td>
<td>0.00373343 (1.00)</td>
<td>0.667573 (1.00)</td>
</tr>
<tr>
<td>Node (v7.6.0)</td>
<td>0.02843 (7.61)</td>
<td>6.27623 (9.40)</td>
</tr>
<tr>
<td>Safari (v10.0.3)</td>
<td>0.02934 (7.86)</td>
<td>7.5565 (11.32)</td>
</tr>
<tr>
<td>Chrome (v56.0.2924.87)</td>
<td>0.02849 (7.63)</td>
<td>6.26183 (9.38)</td>
</tr>
<tr>
<td>Firefox (v52.0.1)</td>
<td>0.01913 (5.12)</td>
<td>5.36378 (8.03)</td>
</tr>
<tr>
<td>WASM (Firefox)</td>
<td>0.01061 (2.84)</td>
<td>2.59406 (3.86)</td>
</tr>
</tbody>
</table>

changes can likely be merged into the mobile platform Makefile, and JavaScript can simply be offered as another compile target. Because the change is relatively light, I believe it should not pose a great challenge to maintain.

4 Results

4.1 Benchmarks

I hypothesized that running a neural network at near-native speeds should be possible in the browser. To measure this, I compiled CPU-only TensorFlow into JavaScript and benchmarked it in four execution environments: Node.js (a tool to evaluate JavaScript from the command line) and three of popular, commodity browsers.

My benchmarks are two C/C++ programs that I linked against the library and compiled into both C and JavaScript. They include two separate model definitions: deep MNIST, and Inception V3. Deep MNIST is the canonical TensorFlow tutorial for neural networks, and uses a 5-layer convolutional neural network to classify the MNIST dataset of handwritten digits [9]. Inception V3 is one of Google’s latest models for computer vision that beat state-of-the-art algorithms when it was released [27, 28]. It is a 42-layer convolutional neural network. My full results are in Table 2.

I found that the speed at which my compiled TensorFlow evaluated graphs was within an order of magnitude of the singlethreaded, native equivalent. The native version ran MNIST at 0.0037 seconds and Inception at 0.668 seconds,
measured as the average over 100 trials; the compiled JavaScript ran MNIST in 0.028 seconds and Inception in 6.26 seconds on Chrome. Firefox is even faster than the other commodity browsers because of it is AOT compilation for ASM.js, and it ran MNIST at 0.019 seconds and Inception at 5.36 seconds. It may be worth noting that this only measures graph evaluations, and does not include the one-time setup time for the library to initialize. I chose not to include this number because it is a flat, upfront cost that can be done asynchronously while the web page loads, so I did not consider it to be along the critical path.

One may correctly point out that my benchmarks are focusing solely on image recognition. I did this intentionally because image models are very developed and benchmarks are readily available. I believe these results should be fairly generalizable to other neural network architectures as I have no reason to believe image recognition architectures should be specifically optimized or unoptimized for browser evaluation.

### 4.2 Library Size

Web developers spend extraordinary amounts of time optimizing for page load time, and at the end of the day, any code that a web page has to run must be transmitted to the client. For this reason, a reasonable question for any potential JavaScript library is the size. The full file size of my TensorFlow.js library is 30MiB, with another 4.7MiB to initialize Emscripten.

The HTTP/1.1 standard includes provisions for compressing data sent between the client and the server. The “Accept-Encoding” header allows the developer to specify which encodings to be used in communication [13, Section 14.3]. One of the most common encodings is gzip, although brotli (Google’s competitor) is beginning to challenge that[8]. For this reason, I wanted to explore performance when taking into account GZip. After GZip compression, the library is 4.2MiB and the initialization code is 408KiB. While this is large compared to other JavaScript libraries, it is reasonable to send over the browser.

I can further reduce file size by choosing to selectively include Ops. One theme I found throughout this project was that compiling into JavaScript is similar to compiling for a mobile or embedded device. Mobile and embedded devices tend to also be compute-constrained, memory-constrained, and to use different architecture than the host (i.e. ARM and mobile architectures). One of the issues mobile developers also face is reducing the library size, both because device memory is limited and because app stores, such as the Apple app store, impose restrictions that apps must be under 100MiB to update or install over cellular networks [15]. To overcome this, the community has developed the print selective registration header tool. Normally TensorFlow ships with hundreds of unique ops (216 op files are included in TensorFlow core, some defining more than one op). However, most models only require a small subset of these; for example, the trained Inception model only requires 11 unique

---

15For more details, see https://github.com/tensorflow/tensorflow/blob/master/tensorflow/python/tools/print_selective_registration_header.py
ops. The `print_selective_registration_header` tool allows developers to selectively compile only Op definitions that will be used in a specific graph, thus significantly reducing file size.

### 4.3 WebAssembly

WebAssembly (WASM) is a new technology and has been described as “ASM.js done right” [35]. It is a binary format being actively developed by a W3 Community Group with representatives from every major browser vendor; it is supported by every major commodity browser, albeit behind a config flag in most browsers [24]. At the time of this writing, FireFox is the only browser that supports WebAssembly out of the box [6]. In my benchmarks, WebAssembly MNIST and Inception (measured on FireFox) run significantly faster than the ASM.js versions, and only appear to be 4x slower than native single-threaded code.

Because WebAssembly is a binary protocol, it is also significantly smaller than ASM.js (which is ASCII based). The library compiled to WASM is only 17MiB (no initialization code needed), and compresses to 2.4MiB with GZip.

### 5 Developer Experience

#### 5.1 Universal Graph Runner

While the library allows developers to write machine learning programs in C/C++ and compile into JavaScript, I did not think this was a great developer experience, as writing machine learning in C/C++ is hard and requires the developer to write in multiple languages. Returning to the design principles that lead the TensorFlow developers to write a Python front-end for TensorFlow, I wanted developers to be able to write in their normal, high-level scripting language while keeping what makes the library performant behind an abstraction barrier. This follows the same architecture used by normal TensorFlow, where a developer writes in Python and does not have to concern themself with the C++ engine.

Emscripten provides an framework called `embind` to call into arbitrary functions in Emscripten-compiled C++. To call one of these functions, the developer only needs to “bind” it by defining a JavaScript string that can be used to refer to and call the function. `Embind` can only pass function arguments that are numeric types, strings, booleans, and references to classes that have been returned from previous `embind` function calls\(^\text{16}\). The main challenge I faced was that the data structures I needed to pass to the ASM.js engine were more complicated than the simple types provided by `Embind`. For example, my inputs are all tensors that could not be easily represented by a single value.

\(^{16}\)For more details, see [https://kripken.github.io/emscripten-site/docs/porting/connecting_cpp_and_javascript/embind.html#built-in-type-conversions](https://kripken.github.io/emscripten-site/docs/porting/connecting_cpp_and_javascript/embind.html#built-in-type-conversions)
I overcame this difficulty by leveraging the serialization format that TensorFlow already ships with, Protobufs. Protobufs are a language-independent, binary serialization format. The developer defines a protobuf definition in a .proto file, which is similar to the definition of a struct. This definition can then be compiled to any supported language, including C, Python, and JavaScript, using the protoc compiler. The compiled code is able to read and write byte arrays that all encode the same struct. Because byte arrays are not available in all languages, the Protobuf library tends to encode serialized protobufs as strings, which allocate at least a byte per char in every supported language.

I compiled the tensor protobufs into JavaScript, and wrote a library to convert multidimensional, JavaScript arrays to and from tensor protobufs. Then I bound functions to create a TensorFlow session (with a graph protobuf), to pass the session inputs, and to retrieve its outputs. I compiled this program from C++ into JavaScript, so that any developer can load it in their web page, load arbitrary graphs, and evaluate them. I call this the “universal graph runner” as it is sufficiently generalizable to run any graph.

5.2 Developing with TFJS

While the developer can now evaluate any model client-side writing only JavaScript, the Emscripten abstraction is still leaky. When returning classes from Emscripten, those classes must be manually garbage collected (as Emscripten does not know when to free the memory), and arguments must still be encoded as tensor protobufs. For this reason, I created TFJS, which handles with details such as loading the library and doing garbage collection.

An example JavaScript program is outlined in Figure 9. Deconstructing this line by line: `TFJS.for_browser('/tensorflowjs/')` loads the library from the URL at '/tensorflowjs' and wraps the library loading in a JavaScript Promise. This allows the developer to load the library asynchronously to the main execution thread. Additionally, it means that the library can be copied to the build directory of a site or served directly from a content distribution network (CDN, rather than having to be `require()`'d). By serving it separately, the library can be served with much more aggressive caching headers than the other JavaScript

---

17 My TensorJS library is hosted at https://github.com/tomasreimers/tensorjs

18 TFJS is hosted at https://github.com/tomasreimers/tfjs
const TFJS = require('tfjs');
const tensorjs = require('tensorjs');

// const simple_addition_graph_pb = ...

TFJS.for_browser('/tensorflowjs/').
  then(lib => {
    const sess = new lib.Session(simple_addition_graph_pb);

    const results = sess.run(
      {
        "a:0": tensorjs.intTensor(a),
        "b:0": tensorjs.intTensor(b)
      },
      "o:0"
    );

    console.log(results[0]);
  });

Figure 9: Example TFJS JavaScript program.

that invokes it. This is helpful because currently it is common to compress all
JavaScript sources into one 'bundle'. I also observed that the 30MiB library
breaks several modern JavaScript bundling tools on default settings, as they are
not used to single files of that size. The for_browser() function call returns a
promise, whose then method is executed once the library is loaded with a single
argument for the library. Additionally there are corresponding for_node() and
for_web_worker() methods, which can load the library in those environments.

The next few lines create the session object, wrapping a native TensorFlow
session. The session exposes a run() method that takes two arguments: a feed
dictionary and a fetch list. The feed dictionary is a mapping from tensor names
to tensors, and the fetch list is a list of tensor names to fetch the value for.
At runtime the feeds are populated, then the graph is run forward (evaluating
any nodes who have all their inputs), at which point the fetches are returned
in a list corresponding to their fetching order. This directly corresponds with
TensorFlow's Python binding API. Results are returned as JavaScript multi-
dimensional arrays; the reason that inputs must be explicitly encoded by the
developer is that JavaScript does not have a sufficiently robust type system to
infer how to encode the inputs.

5.3 Evaluating Image Models

One common set of TensorFlow graph ops not compiled into my universal graph
runner are the TensorFlow image_ops, such as the DecodeJpeg op, which can
const TFJS = require( `tfjs` );
const tensorjs = require( `tensorjs` );

// const mnist_graph_pb = ...

TFJS.for_browser( `/tensorflowjs/` ).then(lib => {
    const sess = new lib.Session(mnist_graph_pb);

    const canvas = document.getElementById( `handwriting` );
    const context = canvas.getContext( `2d` );
    const img_data = context.getImageData( 0, 0, canvas.width, canvas.height);
    const img_array = lib.image_ops.get_array(img_data, true, 0, 255);

    const results = sess.run(
        {
            "Reshape:0": tensorjs.floatTensor(img_array),
            "dropout:0": tensorjs.floatTensor(1.0)
        },
        ["prediction_onehot:0"]
    );
});

Figure 10: A sample TFJS program with image data.

take a string-encoded JPEG and convert it into a uint8 tensor. The reason I have not compiled image ops is because they do not ship with core by default, and they are dependent on various image libraries that non-trivially increase my file size. Additionally, a lot of this functionality, such as reading JPEG files or resizing images is already possible in the browser using native JavaScript technologies. To help reduce developer friction, I created the get_array() method to convert a standard JavaScript image into a multidimensional array (so that it can be converted into a tensor). For an example, see Figure 10.

This code takes the array buffer returned by Canvas' native getImageData() and turns it into a multidimensional array. The multidimensional array is of the form [batch, height, width, channel], where batch is always of length 1. I do this because it is the convention followed in all TensorFlow image ops. The get_array() method can takes 3 additional arguments: the first tells it whether to have three channels (i.e. r, g, b) or simply 1 (reducing the image to grayscale). The latter two are the mean and the standard deviation, every color is set to be (original_color - mean) / std, which allows the developer to

Implementation at https://github.com/tensorflow/tensorflow/blob/2771108b5f6ce2e1692f9440631a183b3808fa01/tensorflow/core/ops/image_ops.cc#L314

For more details, see https://github.com/tensorflow/tensorflow/blob/master/tensorflow/examples/label_image/main.cc#L114
normalize image data. Other than the image processing, this demo is identical to the simple graph demo above.

6 Weight Quantization and Compression

6.1 The Need for Weight Compression

For the same reasons that I am interested in reducing my library size, such as load times and bandwidth consumption, I am also interested in minimizing my model sizes. Using the same benchmark models from above, DeepMNIST is 12MiB and Inception v3 is 91MiB. These file sizes are large to stream from a Web server to a Web browser, and so I looked for ways to reduce the network load. While compression worked well on the library, it works less well on the model files. Using GZip, DeepMNIST barely compresses and is still 12MiB. Inception compresses slightly, reducing the file size from 91MiB to 85MiB. Neither of these results are very satisfactory though, because streaming 85MiB to the client is still fairly large for the web.

Knowing that the naive approach to compression failed, I began to look at ways to either lossily compress the data or find a way to take advantage of the structure to achieve higher compression rates. TensorFlow graph files are largely weights. Stripping the weight tensors from the GraphDef files, the DeepMNIST graph is only 3.7KiB and the Inception graph is only 120KiB. Knowing that, I began to consider ways to compress the weights.

6.2 Lossy Weight Compression

There is a body of literature on “weight quantization” [15]. In fact, TensorFlow even uses it internally when transferring weights within a distributed system [3]. In this section, I motivate and build off this literature. We also propose a more optimal clustering algorithm for weights$^{21}$. Weight quantization is a method to reduce model file sizes, stemming from the realization that neural network weights are inherently resilient to small changes, since they are trained through stochastic gradient descent. Currently, each weight can take on any of $2^{32}$ unique values, and so must be represented by 32 bits. However, if the weights are sufficiently densely clustered, one may not need that many unique values. Instead, one could cluster all of the weights into fewer unique values, and have fewer bits to represent each weight. To find the values for each weight, they could either use a numeric representation with fewer bits, or possibly have a codebook to translate bit patterns into specific 32-bit numbers.

One approach to this is to cluster the weights into $2^n$ clusters. Replacing each weight with its cluster centroid would accomplish no file size reduction (as the original weight and centroid are both 32 bits). However, one can replace each weight with an $n$ bit number, and a cryptographic codebook can be appended to

$^{21}$Much of this content originally appeared online at https://medium.com/@tomasreimers/when-smaller-better-4b54cedc3402#.zhvtzq8oa
the end to provide a lookup from the \( n \) bits to the cluster centroid. Clustering can either be done using k-means clustering, or using dynamic programming to find the optimal clustering (as shown in Appendix A). Assuming the codebook is of negligible size (when compared to the multi MiB weights), this approach should reduce the file size to \( n/32 \) its original size. To further improve this ratio, one could even implement Huffman coding to represent the weights. Weights in these graphs are not evenly distributed, making them a good target for Huffman coding; the distribution of these weights are shown in Figure 11.

Alternately, I can replace the weights with their cluster centroids and not implement the codebook. While this will not decrease the file size, it will allow GZip to further compress the file by increasing data redundancy. This approach saves the additional time that encoding and decoding would take, because all network traffic is going through GZip anyway. I tried this on DeepMNIST and Inception, clustering the 32bit weights into 8 clusters. After Gzip compression, DeepMNIST achieved a compression to 5MiB (clustering all the weights in the graph), and on Inception compressed to 37MiB (clustering weights in each layer of the graph).

7 Future Work

7.1 Web Technologies on the Horizon

The Web is constantly improving, and the technologies and APIs available to developers are constantly changing. Various technologies are gaining adoption that make me think that machine learning in the browser will become even more feasible and performant. Three technologies that I see as candidates for immediate integration are:

1. GPU Acceleration with WebGL: WebGL is a technology that allows developers to execute GLSL (GPU Shader Code) from the web browser [32]. This allows JavaScript applications to leverage the GPU in the same way native apps leverage the GPU. WeBLAS has already done this to speed up linear algebra, and the same approach is theoretically possible to speed
2. Threading: POSIX-style threading is not possible in the browser because I cannot create multiple OS level threads with shared memory. At the time of this writing, Firefox Nightly currently supports an experimental threading model based on a shared memory array between Web workers that is currently being pushed into the standards track\textsuperscript{23}. Additionally, Emscripten has the ability to compile threaded C programs to this experimental API, although that is not supported by other browsers yet. As soon as support becomes more ubiquitous, I will explore this route.

3. SIMD Instructions: SIMD is a type of computer architecture where a processor can process multiple pieces of data with the same command simultaneously. This increases data parallelism and can be very useful when dealing with multiple items in a batch or members of a vector. While SIMD instructions are currently leveraged in native code, JavaScript is just now starting to have an API to access them \textsuperscript{26}. Emscripten already writes code that complies to this standard (using a polyfill to support browsers that do not support it)\textsuperscript{24}, however, I elected not to include results for this because there is very little browser support for it. As support for this becomes more common, I will benchmark it and expect to see the vectorized code speed up on machines that support SIMD.

By integrating those three technologies, I will achieve even greater speeds. There are other technologies, which are further on the horizon, that will lead to even faster speeds. One such technology is WebCL, which brings OpenCL to the web. I chose not to include it because browser support is sufficiently sparse and the specification still being written. [1].

7.2 Training

Up until this point, we have only discussed running inference in the browser. A logical next step would be to consider training models in the browser. Currently, TensorFlow officially only supports training via the Python layer (because the core logic for reverse-mode automatic differentiation, which is necessary to compute gradients for back-propagation, is implemented in Python\textsuperscript{25}). However, it is expected this API will be implemented in C++, and the official TensorFlow repository now ships with an example of basic training in C++\textsuperscript{26}. As training

\textsuperscript{22}WeBLAS is hosted at https://github.com/waylonflinn/weblas


\textsuperscript{24}For more details, see https://kripken.github.io/emscripten-site/docs/porting/simd.html

\textsuperscript{25}Implementation at https://github.com/tensorflow/tensorflow/blob/master/tensorflow/python/ops/gradients_impl.py

\textsuperscript{26}For further information regarding differentiation in C++ see https://github.com/tensorflow/tensorflow/blob/master/tensorflow/python/ops/gradientsImpl.cc, and for an example training graph in C++ is see https://github.com/tensorflow/tensorflow/blob/master/tensorflow/cc/tutorials/example_trainer.cc
becomes available in the C++ API of TensorFlow, I expect to be able to compile that code directly into my library. Even if C++ does not gain the ability to create the reverse-mode automatic differentiation graph, I could imagine constructing the graph in Python and shipping the trainable graph to the browser. Regardless of how it is implemented, this technology would have applications in fine-tuning networks to user-specific data and leveraging the compute of multiple user devices.

While full scale training might be too resource intense, one can imagine applications of either Transfer Learning (where a neural network is tuned to user specific data) or crowd training (where a collection of clients are used to train a neural net) [16]. One application of this latter approach would be to train a collective neural net on individual data while preserving privacy [22]\(^{27}\).

8 Conclusion

I presented TensorFlow.js, a compilation of Google’s library TensorFlow into JavaScript. I decided to compile for the speed benefits, ability to interoperate with the model files of an already-popular library, and feasibility to maintain without needing to replicate all of TensorFlow’s changes. To create the compilation, I leveraged Emscripten, a compiler from C/C++ to JavaScript. I also had to modify TensorFlow slightly in order to port the dependencies to JavaScript, correct type assumptions, and match the concurrency model with the abstraction provided by JavaScript.

I found that TensorFlow.js operates at a speed within an order of magnitude of native, single-threaded TensorFlow. To measure this, I introduced two benchmarking models: DeepMNIST, a handwriting recognition model that is the introductory TensorFlow tutorial, and Inception, Google’s image recognition net. Singlethreaded TensorFlow ran inception at 0.67 seconds, while TensorFlow.js ran it at 6.26s in Chrome. Additionally, TensorFlow.js ran faster on FireFox, which leverages ahead-of-time compilation for ASM.js, at 5.36s; it ran even faster when compiled to WebAssembly, an emerging binary execution format for the web, at 2.59s.

As part of my benchmarks, I also discussed file sizes and potential strategies to reduce those numbers. Uncompressed, my library is approximately 30MiB. However, because most web traffic is compressed during transmission, I also measured compressed file sizes. Using GZip, I compressed the library to 5MiB. Using WebAssembly, I can further reduce this number to 17MiB uncompressed and 2.4MiB compressed because it is a binary protocol. I also explain how selectively compiling parts of the library could be used to further reduce file size. Furthermore, I discuss how I can compress model files using weight quantization, reducing the Inception model by almost a factor of 3, from 91MiB to 37MiB.

I imagine that being able to evaluate machine learning in the browser will have three major use cases: (1) privacy, for when the user does not want to

\(^{27}\)An example TensorFlow implementation can be found at https://github.com/tensorflow/models/tree/master/differential_privacy/multiple_teachers
send data to a server; (2) when the server is offline or bandwidth is sufficiently limited where it is preferable to run it locally; and (3) to create totally self-contained demos that run on any platform. I discussed previous attempts to bring machine learning to the browser, and some challenges those attempts faced, including speed and interoperability.

In an effort to improve developer experience, I designed and implemented a wrapper for TensorFlow.js that hides the performant library behind an abstraction barrier. This allows developers to run models performantly, without having to write C/C++ themselves. I discuss challenges of developing the wrapper, including model and input serialization. The wrapped library exposes an API that is identical to TensorFlow’s Python API.

Lastly, I end by discussing ways in which this library will improve in speed as various technologies are implemented by browser vendors. Three technologies I am particularly excited about are GPU access, threading, and SIMD instructions. I also discuss interesting applications of this technology to training models on client machines.

I believe this technology is valuable, and am excited for it to enable people to write a new class of machine learning enabled applications that have been impossible until now. I envision a future where machine learning is everywhere, even client-side on the web. One could even imagine a future, once machine learning is more standardized, where browser vendors provide an API similar to TensorFlow for performant evaluation of machine learning models.

9 Acknowledgments

It has been an awesome journey taking this from what my advisers referred to as a “fever dream” to a completely functioning, documented, and benchmarked piece of software. I would like to thank, first and foremost, my senior thesis adviser, James Mickens, for guiding me throughout this project, reading countless drafts and status updates, and placing enough trust in me to join me as I pursued this project (despite there being times when I was thoroughly convinced that my JavaScript benchmarks were running faster than their C equivalents). I would also like to thank my two thesis readers, Jim Waldo and Alexander Rush for mentoring me and giving me feedback along the way.

I want to give a shout out to the TensorFlow team at Google and the creator of Emscripten, Alon Zakai (Kripken) at Mozilla, as without their work, none of this would have been possible. Specifically on the TensorFlow team, I would like to thank Asim Shankar, Neal Wu, Justin Hseu, Pete Warden, Todd Wang, Doug Fritz, Martin Wattenberg, Charles Nicholson, Nikhil Torat, Jeff Dean, and everyone else that took the time to meet with me and help me on my way. I also want to give a shout out to Leon Chen at md.ai, Andrej Karpathy at OpenAI, and Reza Zadeh at Matroid for motivating this work and providing guidance and support.

Last, but not least, I would like to thank my friends, roommates, and family for helping me think through my ideas and supporting me through-
out this project. I would especially like to thank the following individuals, who either proofread early drafts or discussed deeply-technical details: Athalye, Danoff, Fay, Foster, Getahun, Goel, Gupta, Kahn, Kern, Khanna, Kraft, Krentz, Kwok, Levenson, Li, Mishra, Mohamedali, Piana, Pochtar, Schluntz, Shumikihn, Siegel, Srinivasan, Szulanski, Torres, Wang, Woodin, Yang, Zurawicki, and everyone else who helped shape this work.

References


10 Appendix A: Optimal Clustering Algorithm for points in a 1-Dimensional Space

To optimally cluster points in 1-dimensional space, let all points be \( P \) and the clustering be \( C \). Also let the cost function of a specific clustering configuration be the sum of \( \text{cost}(p, c) \) over all point \( p \) and cluster \( c \) pairs, where point \( p \) is in cluster \( c \). To find the optimal clustering of this one dimensional data, use the following algorithm (for my purposes, I defined cost to be the distance to the cluster centroid squared):

10.1 Minimizing Loss

To minimize loss, create an array of all the points sorted, as I rely on passing around indices into that array. Then define \( f(i, c) \), which represents the minimum cost of the optimal clustering of points \([i, P]\) using \( c \) clusters. In the case that \( c = 0 \), \( f \) is 0 if \( i = P \) (as there are no more points left to cluster), and infinity otherwise (because a clustering configuration that does not include all the points is invalid). In all other cases, \( f \) is the minimum of \( \text{cost}(i, j) + f(j, c-1) \) for all values of \( j \) in \([i, P]\). \( f(i, c) \) should be memoized, or populate a 2-dimensional lookup table. Call \( f(i = 0, c = 0|\text{clusters}) \) to find the minimal cost.

This is correct because if you were to split an optimal clustering at any cluster boundary, the two resulting values should also be optimally clustered (as if they were not, there would be a more optimal clustering by optimizing that half with however many clusters it had). Populating any cell in the lookup table should take \( P \) time (if you have all other values in the table), and there are \( CP \) cells, so the entire algorithm should run in \( O(CP^2) \) time and \( O(CP) \) space.

10.2 Optimal Clustering

The above does not actually provide the optimal clustering, simply the minimal cost. Fortunately, it can be modified to provide the optimal clustering without modifying the run time. Simple, modify the above algorithm so \( f(i, c) \) returns a tuple, where the first element is the cost and the second element is the value for \( j \) used to minimize the cost. After calling \( f(0, 0|\text{clusters}) \), store the \( j \), call \( f(j, 0|\text{clusters}) \) (which should be memoized), store the \( j \), and repeat until you have stored all the cluster partitions.
11 Appendix B: Code Samples

11.1 Type Corrections

```cpp
diff --git a/tensorflow/core/kernels/resize_bilinear_op.cc b/tensorflow/core/kernels/resize_bilinear_op.cc
index 2c5aeaa..461e333 100644
--- a/tensorflow/core/kernels/resize_bilinear_op.cc
+++ b/tensorflow/core/kernels/resize_bilinear_op.cc
@@ -154,14 +154,14 @@ void scale_down_image(typename TTypes<T,
               4>::ConstTensor images,
               6
               if (channels == 3) {
               7                   for (int b = 0; b < batch_size; ++b) {
               8                       // Compute the interpolation
               9                       for (int64 y = 0; y < out_height; ++y) {
+                             const Eigen::Index ys_lower = ys[y].lower;
+                             const Eigen::Index ys_upper = ys[y].upper;
                       +                   for (Eigen::Index y = 0; y < out_height; ++y) {
                       +                       const Eigen::Index ys_lower = ys[y].lower;
                       +                       const Eigen::Index ys_upper = ys[y].upper;
                       +                       const float ys_lerp = ys[y].lerp;
                       +                       const CachedInterpolation* xs_ptr = xs_vec.data();
                       +                   for (Eigen::Index x = 0; x < out_width; ++x) {
                       +                       const Eigen::Index xs_lower = xs_ptr->lower;
                       +                       const Eigen::Index xs_upper = xs_ptr->upper;
                       +                   for (Eigen::Index x = 0; x < out_width; ++x) {
                       +                       const Eigen::Index xs_lower = xs_ptr->lower;
                       +                       const Eigen::Index xs_upper = xs_ptr->upper;
                       +                       const float xs_lerp = xs_ptr->lerp;
                       +                   xs_ptr++;
```

This is exemplary of the type replacement that had to occur to make TensorFlow function on 32-bit systems. `Eigen::Index` is defined to be `std::ptrdiff_t`, which causes problems further in the code where `Eigen::Index` variables were explicitly assigned to variables such as `ys_lower`. 
11.2 Concurrency

This is the core of my threading change. If the preprocessor macro \_\_SINGLE\_\_THREAD\_\_ is defined, it replaces the schedule method with code that immediately executes the passed function closure.