TensorFlow.js: Machine Learning for the Web and Beyond

Daniel Smilkov [*1] **Nikhil Thorat** [*1] **Yannick Assogba** [1] **Ann Yuan** [1] **Nick Kreeger** [1] **Ping Yu** [1] **Kangyi Zhang** [1] **Shanqing Cai** [1] **Eric Nielsen** [1] **David Soergel** [1] **Stan Bileschi** [1] **Michael Terry** [1] **Charles Nicholson** [1] **Sandeep N. Gupta** [1] **Sarah Sirajuddin** [1] **D. Sculley** [1] **Rajat Monga** [1] **Greg Corrado** [1] **Fernanda B. Viégas** [1] **Martin Wattenberg** [1]

Abstract

TensorFlow.js is a library for building and executing machine learning algorithms in JavaScript. TensorFlow.js models run in a web browser and in the Node.js environment. The library is part of the TensorFlow ecosystem, providing a set of APIs that are compatible with those in Python, allowing models to be ported between the Python and JavaScript ecosystems. TensorFlow.js has empowered a new set of developers from the extensive JavaScript community to build and deploy machine learning models and enabled new classes of on-device computation. This paper describes the design, API, and implementation of TensorFlow.js, and highlights some of the impactful use cases.

1 Introduction

Machine learning (ML) has become an important tool in software systems, enhancing existing applications and enabling entirely new ones. However, the available software platforms for ML reflect the academic and industrial roots of the technology. Production-quality ML libraries are typically written for Python and C++ developers. Nonetheless, there is a vast community of both frontend and backend JavaScript (JS) developers that continues to grow at a high pace. There were 2.3 million GitHub pull requests in JS in 2017, compared to 1 million in Python (GitHub.com, 2017). According to the Stack Overflow Developer Survey in 2018, JS is the most commonly used programming language (StackOverflow.com, 2018).

This lack of attention matters. The JS environment has the potential to support a new and distinctive class of applications. On-device computation has a number of benefits, including data privacy, accessibility, and low-latency interactive applications. Empowering the community of JS developers may lead to new classes of applications.

This paper describes the design and development of the TensorFlow.js library, which was motivated by the importance of the JS community and web-based applications for ML. A first-class citizen in the TensorFlow (Abadi et al., 2016) ecosystem, the platform brings high-performance ML and numeric computation capabilities to JS. While several open source JS platforms for ML have appeared, to our knowledge TensorFlow.js is the first to enable integrated training and inference on the GPU from the browser, and offer full Node.js integration for server-side deployment. We have attempted to ensure that TensorFlow.js meets a high standard of productionization, including high-level libraries, comprehensive testing, and explicit extensibility. It has already seen significant uptake by the JS community.

We discuss three main aspects of our experience building TensorFlow.js. First, we describe some of the unique challenges and advantages of the JS environment. Second, we cover the design details of the library, its APIs, which represents a balance between standard web development practice and compatibility with TensorFlow, and the techniques we used to overcome the limitations of the JS environment. Finally, we describe a few interesting and new use cases that have been enabled by TensorFlow.js.

2 Background and Related Work

The design of TensorFlow.js is grounded in specific constraints of the JS environment. Here we detail the technical challenges of ML with JS and related efforts to address them.

2.1 The JavaScript environment

Different environments. One of the challenges of JS is that it runs in different environments. Computation can happen client-side in a browser, server-side, most notably as part of the Node.js framework, and more recently on the desktop

[*]Equal contribution [1]Google Brain. Correspondence to: Daniel Smilkov <smilkov@google.com>, Nikhil Thorat <nsthorat@google.com>.

Proceedings of the 2^{nd} SysML Conference, Palo Alto, CA, USA, 2019.

via frameworks like Electron. TensorFlow.js is designed to work in all these settings, although the majority of our work to date has been tuning it for client-side development in a web browser.

Performance. A second key challenge, specific to the browser environment, is performance. JS is an interpreted language so it does not typically match the speed of a compiled language like C++ or Java for numerical computation. Unlike Python which can bind to C++ libraries, browsers do not expose this capability. For security reasons, browser applications don't have direct access to the GPU, which is typically where numerical computation happens for modern deep learning systems.

To address these performance issues, a few new JS standards are emerging. One notable solution is WebAssembly (Haas et al., 2017), a method for compiling C++ programs to bytecode that can be interpreted and executed directly in the browser. For certain tasks, WebAssembly can outperform plain JS. Most modern browsers also support WebGL (Kronos, 2011), an API that exposes OpenGL to JS. OpenGL is a cross-language, cross-platform API for rendering 2D and 3D vector graphics, enabling games and other high-performance rendering tasks directly in a webpage. On the server-side, JS libraries can bind to existing native modules that are written in C and C++ via Node.js's N-API interface (Nodejs.org, 2017).

Cross-browser compatibility. JS is designed to be a cross-platform language supported by all major browsers with standardized Web APIs that make it easy to write applications that run on all platforms. In practice, browsers are built by several different vendors with slightly different implementations and priorities. For example, while Chrome and Firefox support WebGL 2.0 (a significant improvement over WebGL 1.0), Apple's Safari has settled on WebGL 1.0 and shifted focus to future technologies such as WebGPU (Jackson, 2017). Web application authors have to work hard to hide this inconsistency in their applications, often requiring extensive testing infrastructure to test across large number of platforms.

Single-threaded execution. One of the other challenges of the JS environment is its single threaded nature. JS has a 'main thread' (also known as the 'UI thread'), which is where webpage layout, JS code, event processing and more happen. While this greatly simplifies some aspects of the development model, it also means that application developers need to be careful not to block the main thread as it will cause other parts of the page to slow down. A well-designed JS library therefore requires a careful balance between the simplicity of synchronous APIs and the non-blocking benefits of asynchronous APIs.

2.2 Opportunities in a browser-based environment

Shareability. A major motivation behind TensorFlow.js is the ability to run ML in standard browsers, without any additional installations. Models and applications written in TensorFlow.js are easily shared on the web, lowering the barrier to entry for machine learning. This is particularly important for educational use cases and for increasing the diversity of contributors to the field.

Interactivity. From a machine learning perspective, the interactive nature of web browsers and versatile capabilities of Web APIs open the possibility for a wide range of novel user-centric ML applications which can serve both education and research purposes. Visualizations of neural networks such as (Olah, 2014) and (Smilkov et al., 2016) have been popular to teach the basic concepts of machine learning.

On-device computation. Lastly, standardized access to various components of device hardware such as the web camera, microphone, and the accelerometer in the browser allow easy integration between ML models and sensor data. An important result of this integration is that user data can stay on-device and preserve user-privacy, enabling applications in the medical, accessibility, and personalized ML domains. For example, speech-impaired users can use their phones to collect audio samples to train a personalized model in the browser. Another technology, called Federated Learning (McMahan et al., 2016), enables devices to collaboratively train a centralized model while keeping sensitive data on device. Browsers are a natural a platform for this type of application.

2.3 Related work

Given the popularity and the unique benefits of the JS ecosystem, it is no surprise that many open-source browser-based ML libraries exist. ConvNetJS (Karpathy, 2014), Synaptic (Cazala, 2014), Brain.js (Plummer, 2010), Mind (Miller, 2015) and Neataptic (Wagenaar, 2017) each provide a simple JS API that allows beginners to build and train neural networks with only a few lines of code. More specialized JS ML libraries include Compromise (Kelly, 2014) and Natural (Umbel, 2011), which focus on NLP applications, and NeuroJS (Huenermann, 2016) and REINFORCEjs (Karpathy, 2015), which focus on reinforcement learning. ML.js (Zasso, 2014) provides a more general set of ML utilities, similar to the Python-based scikit-learn (Pedregosa et al., 2011).

These libraries do not provide access to hardware acceleration from the browser which we have found to be important for computational efficiency and minimizing latency for interactive use cases and state of the art ML models. A few libraries have attempted to take advantage of hardware

acceleration, notably TensorFire (Kwok et al., 2017), Propel (built on top of TensorFlow.js) (Dahl, 2017) and Keras.js (Chen, 2016), however they are no longer actively maintained.

WebDNN (Hidaka et al., 2017) is another deep learning library in JS that can execute pretrained models developed in TensorFlow, Keras, PyTorch, Chainer and Caffe. To accelerate computation, WebDNN uses WebGPU (Jackson, 2017), a technology initially proposed by Apple. WebGPU is in an early exploratory stage and currently only supported in Safari Technology Preview, an experimental version of the Safari browser. As a fallback for other browsers, WebDNN uses WebAssembly (Haas et al., 2017), which enables execution of compiled C and C++ code directly in the browser. While WebAssembly has support across all major browsers, it lacks SIMD instructions, a crucial component needed to make it as performant as WebGL and WebGPU.

3 Design and API

The goals of TensorFlow.js differ from other popular ML libraries in a few important ways. Most notably, TensorFlow.js was designed to bring ML to the JS ecosystem, empowering a diverse group of JS developers with limited or no ML experience (Anonymous, 2018). At the same time, we wanted to enable experienced ML users and teaching enthusiasts to easily migrate their work to JS, which necessitated wide functionality and an API that spans multiple levels of abstraction. These two goals are often in conflict, requiring a fine balance between ease-of-use and functionality. Lastly, as a new library with a growing user base, missing functionality was prioritized over performance.

These goals differ from popular deep learning libraries (Abadi et al., 2016; Paszke et al., 2017), where performance is usually the number one goal, as well as other JS ML libraries (see Section 2.3), whose focus is on simplicity over completeness of functionality. For example, a major differentiator of TensorFlow.js is the ability to author and train models directly in JS, rather than simply being an execution environment for models authored in Python.

3.1 Overview

The API of TensorFlow.js is largely modeled after TensorFlow, with a few exceptions that are specific to the JS environment. Like TensorFlow, the core data structure is the *Tensor*. The TensorFlow.js API provides methods to create tensors from JS arrays, as well as mathematical functions that operate on tensors.

Figure 1 shows a high level schematic view of the architecture. TensorFlow.js consists of two sets of APIs: the *Ops API* which provides lower-level linear algebra operations (e.g. matrix multiplication, tensor addition, etc.), and the *Layers API*, which provides higher-level model building blocks and best practices with emphasis on neural networks. The *Layers API* is modeled after the *tf.keras* namespace in TensorFlow Python, which is based on the widely adopted Keras API (Chollet et al., 2015).

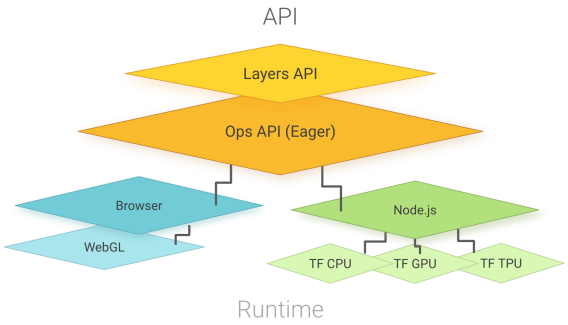

Figure 1. Overview of the TensorFlow.js architecture

TensorFlow.js is designed to run in-browser and server-side, as shown in Figure 1. When running inside the browser, it utilizes the GPU of the device via WebGL to enable fast parallelized floating point computation. In Node.js, TensorFlow.js binds to the TensorFlow C library, enabling full access to TensorFlow. TensorFlow.js also provides a *slower* CPU implementation as a fallback (omitted in the figure for simplicity), implemented in plain JS. This fallback can run in any execution environment and is automatically used when the environment has no access to WebGL or the TensorFlow binary.

3.2 Layers API

Beginners and others who are not interested in the operation-level details of their model might find the low-level operations API complex and error prone. The widely adopted Keras library (Chollet et al., 2015), on the other hand, provides higher-level building blocks with emphasis on deep learning. With its carefully thought out API, Keras is popular among deep learning beginners and applied ML practitioners. At the heart of the API is the concept of a model and layers. Users can build a model by assembling a set of pre-defined layers, where each layer has reasonable default parameters to reduce cognitive load.

For these reasons, TensorFlow.js provides the *Layers API*, which mirrors the Keras API as closely as possible, including the serialization format. This enables a two-way door between Keras and TensorFlow.js; users can load a pretrained Keras model (see Section 5.1) in TensorFlow.js, modify it, serialize it, and load it back in Keras Python.

Listing 1 shows an example of training a model using the Layers API.

```
// A linear model with 1 dense layer.
const model = tf.sequential();
model.add(tf.layers.dense({
  units: 1, inputShape: [1]
}));

// Specify the loss and the optimizer.
model.compile({
  loss: 'meanSquaredError',
  optimizer: 'sgd'
});

// Generate synthetic data to train.
const xs =
    tf.tensor2d([1, 2, 3, 4], [4, 1]);
const ys =
    tf.tensor2d([1, 3, 5, 7], [4, 1]);

// Train the model using the data.
model.fit(xs, ys).then(() => {
  // Do inference on an unseen data point
  // and print the result.
  const x = tf.tensor2d([5], [1, 1]);
  model.predict(x).print();
});
```

Listing 1. An example TensorFlow.js program that shows how to build a single-layer linear model with the layers API, train it with synthetic data, and make a prediction on an unseen data point.

3.3 Operations and Kernels

As in TensorFlow, an *operation* represents an abstract computation (e.g. matrix multiplication) that is independent of the physical device it runs on. Operations call into *kernels*, which are device-specific implementations of mathematical functions which we go over in Section 4.

3.4 Backends

To support device-specific kernel implementations, TensorFlow.js has a concept of a *Backend*. A backend implements kernels as well as methods such as *read()* and *write()* which are used to store the *TypedArray* that backs the tensor. Tensors are decoupled from the data that backs them, so that operations like reshape and clone are effectively free. This is achieved by making shallow copies of tensors that point to the same data container (the *TypedArray*). When a tensor is disposed, we decrease the reference count to the underlying data container and when there are no remaining references, we dispose the data container itself.

3.5 Automatic differentiation

Since wide functionality was one of our primary design goals, TensorFlow.js supports automatic differentiation, providing an API to train a model and to compute gradients.

The two most common styles of automatic differentiation are graph-based and eager. Graph-based engines provide an API to construct a computation graph, and execute it later. When computing gradients, the engine statically analyzes the graph to create an additional gradient computation graph. This approach is better for performance and lends itself easily to serialization.

Eager differentiation engines, on the other hand, take a different approach (Paszke et al., 2017; Abadi et al., 2016; Maclaurin et al., 2015). In eager mode, the computation happens immediately when an operation is called, making it easier to inspect results by printing or using a debugger. Another benefit is that all the functionality of the host language is available while your model is executing; users can use native *if* and *while* loops instead of specialized control flow APIs that are hard to use and produce convoluted stack traces.

Due to these advantages, eager-style differentiation engines, like TensorFlow Eager (Shankar & Dobson, 2017) and PyTorch (Paszke et al., 2017), are rapidly gaining popularity. Since an important part of our design goals is to prioritize ease-of-use over performance, TensorFlow.js supports the eager style of differentiation.

3.6 Asynchronous execution

JS runs in a single thread, shared with tasks like page layout and event handling. This means that long-running JS functions can cause page slowdowns or delays for handling events. To mitigate this issue, JS users rely on event callbacks and promises, essential components of the modern JS language. A prominent example is Node.js which relies on asynchronous I/O and event-driven programming, allowing the development of high-performance, concurrent programs.

However, callbacks and asynchronous functions can lead to complex code. In service of our design goal to provide intuitive APIs, TensorFlow.js aims to balance the simplicity of synchronous functions with the benefits of asynchronous functions. For example, operations like *tf.matMul()* are purposefully synchronous and return a tensor whose data might not be computed yet. This allows users to write regular synchronous code that is easy to debug. When the user needs to retrieve the data that is backing a tensor, we provide an asynchronous *tensor.data()* function which returns a promise that resolves when the operation is finished. Therefore, the use of asynchronous code can be localized to a single *data()* call. Users also have the option to call *tensor.dataSync()*, which is a blocking call. Figures 2 and 3 illustrate the timelines in the browser when calling *tensor.dataSync()* and *tensor.data()* respectively.

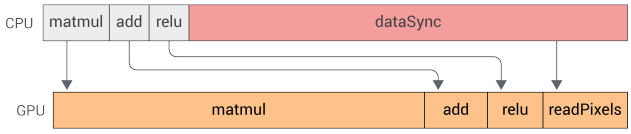

Figure 2. The timeline of a synchronous and blocking *tensor.dataSync()* in the browser. The main thread blocks until the GPU is done executing the operations.

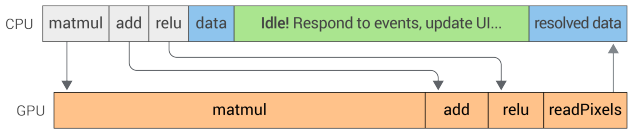

Figure 3. The timeline of an asynchronous call to data() in the browser. The main thread is released while the GPU is executing the operations and the data() promise resolves when the tensor is ready and downloaded.

3.7 Memory management

JS provides automatic garbage collection. However, in the browser WebGL memory is not automatically garbage collected. Because of this, and the lack of finalization, we expose an API for all backends to explicitly manage memory.

To dispose the memory allocated by a tensor, users can call *tensor.dispose()*. This approach is relatively straightforward, but the user has to have a reference to all tensor objects so they can be disposed. Often models are written as chained blocks of operations, so breaking up the chains for disposal can be cumbersome. Since tensors are immutable and operations are functional, a single op call can allocate a significant number of intermediate tensors. Forgetting to dispose these intermediate tensor results in memory leaks and slows down the application significantly.

TensorFlow.js offers an alternative approach. Since functions are first-order citizens in JS, and a large portion of the native JS API uses functions as arguments, we decided to provide a scoping mechanism where the user can wrap any synchronous function f by calling *tf.tidy(() ⇒ f())*. This results in calling f immediately, and disposing all intermediate tensors created inside once f finishes, except for the return result of f. We use this mechanism extensively in our library. Users of the Layers API do not need explicit memory management due to model-level APIs such as *model.fit()*, *model.predict()* and *model.evaluate()* which internally manage memory.

3.8 Debugging and profiling

TensorFlow.js provides a rich set of debugging tools to help developers understand common problems with performance and numerical stability, accessible either via a URL change or a feature flag. Users can profile every kernel that gets called, seeing the output shape, memory footprint, as well as device-specific timing information. In this mode, every tensor gets downloaded from the GPU and is checked for NaNs, throwing an exception at the first line a NaN is introduced, showing model developers which operation is the source of the numerical instability.

TensorFlow.js also provides *tf.time(f)* for timing a function that calls TensorFlow.js operations. When calling *tf.time(f)*, the function f will be executed and timed. Each backend is responsible for timing functions, as timing may be device specific. For example, the WebGL backend measures the exact GPU time, excluding time for uploading and downloading the data.

A more generic API, *tf.profile(f)*, similarly takes a function f and returns an object representing the function's effect on memory. The object contains the number of newly allocated tensors and bytes created by executing the function, as well as the peak tensors and memory allocated inside the function. Understanding peak memory usage is especially important when running on devices with limited memory such as mobile phones.

3.9 Performance

While performance was not the single most important goal, it was critical in enabling real-world ML in JS. In the browser, TensorFlow.js utilizes the GPU using the WebGL API to parallelize computation. By using WebGL for numerical computation, we were able to achieve 2 orders of magnitude speedup, which is what fundamentally enabled running real-world ML models in the browser. On the server-side, TensorFlow.js binds directly to the TensorFlow C API, which takes full advantage of native hardware acceleration.

Table 1 shows the speedups of these implementations relative to the plain JS CPU counterpart. We measure a single inference of MobileNet v1 1.0 (Howard et al., 2017) with an input image of size *224x224x3*, averaged over 100 runs. All measurements, other than those mentioning GTX 1080, are measured on a MacBook Pro 2014 laptop, while the GTX 1080 measurements are done on a desktop machine. Note that the WebGL and the Node.js CPU backends are two orders of magnitude faster than the plain JS backend, while those utilizing the capable GTX 1080 graphics card are three orders of magnitude faster.

Since the launch of TensorFlow.js, we have made significant progress on improving our WebGL utilization. One notable improvement is *packing*, where we store floating

Backend	Time (ms)	Speedup
Plain JS	3426	1x
WebGL (Intel Iris Pro)	49	71x
WebGL (GTX 1080)	5	685x
Node.js CPU w/ AVX2	87	39x
Node.js CUDA (GTX 1080)	3	1105x

Table 1. Speedups of the WebGL and Node.js backends over the plain JS implementation. The time shows a single inference of MobileNet v1 1.0 (Howard et al., 2017), averaged over 100 runs.

point values in all 4 channels of a texel (instead of using only 1 channel). Packing resulted in 1.3-1.4x speedup of models such as PoseNet (Oved, 2018) across both mobile and desktop devices.

While we will continue to work on our WebGL implementation, we observed a 3-10x gap in performance between WebGL and CUDA. We believe the gap to be due to WebGL's lack of work groups and shared memory access, benefits provided by general-purpose computing (GPGPU) frameworks like CUDA(Nickolls et al., 2008) and OpenGL Compute shaders (Shreiner et al., 2013). As we discuss below in Section 4.3, we believe that the upcoming WebGPU (Jackson, 2017) standard is a promising avenue for bridging the gap in performance.

4 Implementation

This section describes the specific constraints and implementations of the various backends that are supported by TensorFlow.js.

4.1 Browser and WebGL

With the advent of deep learning and scientific computing in general, and advances in modern GPU architectures, the use of GPGPU has grown tremendously. While modern JS virtual machines can optimize plain JS extensively, its performance is far below the computational power that GPUs provide (see Table 1). In order to utilize the GPU, TensorFlow.js uses WebGL, a cross-platform web standard providing low-level 3D graphics APIs. Unlike OpenCL and CUDA, the WebGL API is based on OpenGL ES specification (Shreiner et al., 2013) which has no explicit support for GPGPU.

Among the three TensorFlow.js backends, the WebGL backend has the highest complexity. This complexity is justified by the fact that it is two orders of magnitude faster than our CPU backend written in plain JS. The realization that WebGL can be re-purposed for numerical computation is what fundamentally enabled running real-world ML models in the browser.

To work around the limitations and the complexities of WebGL, we wrote a layer of abstraction called the *GPGPUContext* which executes WebGL fragment shaders representing computation. In a graphics program, fragment shaders are typically used to generate the colors for the pixels to be rendered on the screen. Fragment shaders run for each pixel independently and in parallel; TensorFlow.js takes advantage of this parallelization to accelerate ML computation.

In the WebGL backend, the draw pipeline is set up such that the scene geometry represents a unit square. When we execute a fragment shader program, we bind the texture that backs the output tensor to the frame buffer and execute the fragment shader program. This means that the fragment shader *main()* function is executed in parallel for each output value, as shown in 4. For simplicity, we only use the red channel of the texture that backs the tensor (shown as 'R' in the figure). On WebGL 2.0 devices, we use the *gl.R32F* texture type which allows us to avoid allocating memory for the green, blue, and alpha channels (shown as 'G', 'B', and 'A' respectively). In future work, TensorFlow.js will take advantage of all channels for WebGL 1.0 devices, which will better utilize the GPU's sampler cache.

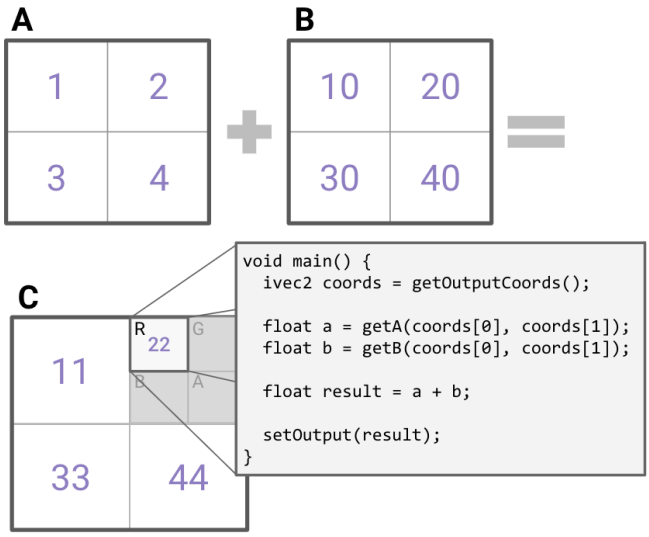

Figure 4. The addition of two equally shaped matrices as executed by the WebGL backend, and the GLSL code of the fragment shader that represents the element wise addition computation. The GLSL function, *main()*, runs in the context of each output value and in parallel, with no shared memory.

Writing OpenGL Shading Language (GLSL) code can be error prone and difficult. To make it significantly easier to write and debug GPGPU programs, we wrote a shader compiler. The shader compiler provides high-level GLSL functions that the shader author can call. Listing 2 shows the GLSL source code for matrix multiplication where the

shared dimension N is assumed to be a multiple of 4 for simplicity. The functions marked with bolded font are provided by our shader compiler.

Using the higher level functions generated by the shader compiler has multiple benefits. First, the user-defined GLSL code operates in high-dimensional 'logical' space instead of the physical $2D$ texture space. For example, the GLSL implementation of *tf.conv2d()* uses the auto-generated *getA(batch, row, column, depth)* method to sample from a 4D tensor. This makes the user code simpler, more readable and less error-prone.

Second, the separation of logical and physical shape allows the framework to make intelligent decisions about memory layout, avoiding device-specific size limits of WebGL textures.

Third, we can optimize the mapping from high-dimensional space to the $2D$ space. For example, assume the logical shape of tensor A is $4D$ with shape *1x3x1x2*. When A gets uploaded to the GPU, the backend will allocate a physical *3x2* texture and the compiler will generate a $getA(a, b, c, d)$ method whose implementation ignores a and c and directly maps b and d into the $2D$ texture space. We observed that this optimization leads to 1.3x speedup on average.

Last, there is a single GLSL implementation of *tf.matMul()* regardless of the browser's WebGL capabilities. In Chrome we render to a 32bit single-channel floating texture, while in iOS Safari we render to a 16bit single-channel floating point texture. In both cases, the user code is the same, using the high-level *setOutput(value)* GLSL method with the browser-specific implementation generated by the compiler.

```
void main() {
  ivec2 coords = getOutputCoords();
  int aRow = coords.x;
  int bCol = coords.y;
  float result = 0.0;
  for (int i=0; i <${N}; i+=4) {
    vec4 a = vec4(
      getA(aRow, i), getA(aRow, i+1),
      getA(aRow, i+2), getA(aRow, i+3));
    vec4 b = vec4(
      getB(i, bCol), getB(i+1, bCol),
      getB(i+2, bCol), getB(i+3, bCol));
    result += dot(a, b);
  }
  setOutput(result);
}
```

Listing 2. GLSL code for matrix multiplication using the higher-level utility functions marked in bold font.

4.1.1 Asynchronous execution

With WebGL, programs get scheduled by the CPU and run on the GPU, which is a separate thread from the main JS thread. This means that while programs are running on the GPU, the CPU is free to respond to events and run other JS code.

When the user calls an operation, we enqueue a program onto the GPU command queue, which typically takes sub-millisecond time, and immediately return a handle to the resulting tensor despite the computation not being done. Users can later retrieve the actual data by calling *tensor.dataSync()* or *tensor.data()*, which returns a *TypedArray*.

As mentioned in Section 3.6, we encourage the use of the asynchronous *tensor.data()* method, which avoids blocking the main thread, and returns a promise that resolves when the computation is done (see Figures 2 and 3). However, to retrieve the underlying data of a texture, the WebGL API only provides a blocking *gl.readPixels()* method. To get around this limitation, we approximate when the GPU is done executing the operations, postponing the call to *gl.readPixels()*, which releases the main thread in the meantime.

Approximating when the GPU has finished executing programs can be done in a couple of ways. The first approach taken in TensorFlow.js, for WebGL 1.0 devices, uses the *EXT_disjoint_timer_query* WebGL extension. This extension can be used to accurately measure the GPU time of programs, but also implicitly has a bit that gets flipped when a program is done executing. The second approach, for WebGL 2.0 devices, uses the *gl.fenceSync()* API by inserting a 'fence' into the GPU command queue and polling a query which returns true when the fence has flipped.

4.1.2 Memory management

Disposing and re-allocating WebGL textures is relatively expensive, so we don't release memory when a tensor gets disposed. Instead, we mark the texture for reuse. If another tensor gets allocated with the same physical texture shape, we simply recycle the texture. The texture recycler gives us significant performance wins since multiple passes through the same ML model often generate tensors of the same shapes.

One of the common problems with manual memory management is memory leaks, where a program with a loop creates one or more tensors during each tick that never get disposed. This will eventually cause the application to run out of memory and crash. Since one of our primary design principles is easy-of-use over performance, we provide built-in heuristics to avoid crashing the application. Specifically, we automatically page WebGL textures to the CPU when the total amount of GPU memory allocated exceeds a threshold which can be estimated from the screen size. At the same time, the paging logic will not take effect for users that explicitly manage memory using *tf.tidy()* or *tensor.dispose()*.

4.1.3 Device support

While WebGL is a slow-evolving web standard, it has ubiquitous support. TensorFlow.js can run on desktop, tablet and mobile devices. The WebGL backend requires a WebGL 1.0 compatible device that supports the *OES_texture_float* extension which enables uploading and reading from floating point textures. According to WebGLStats.com (webglstats.com, 2018), a website that tracks the WebGL capabilities of devices and their market share on the web, TensorFlow.js can run on 99% of desktop devices, 98% of iOS and Windows mobile devices, and 52% of Android devices. We believe that the reason for the significant Android discrepancy is due to a large number of older Android devices that have no GPU hardware, and that this gap will gradually close as users migrate to newer Android devices.

While WebGL has wide support on different platforms, not all browsers have the exact same implementation of the WebGL specification. This leads to problems with the numerical stability of the library. For example, on iOS devices, the WebGL API is explicit about the lack of 32bit float support, reverting to 16bit floats, which is aligned with the underlying capability of mobile GPUs. On the other hand, mobile Chrome hides that detail by allowing developers to upload and write in 32bit float regardless of the underlying precision. This led to numerical precision problems on Android devices: $log(x + \epsilon)$ resulted in $log(x + 0)$ since the default $\epsilon = 1 \times 10^{-8}$ was not representable in 16bit float and implicitly rounded to 0. To solve this problem, we adjust the global ϵ value according to the device capabilities.

4.2 Node.js

With the advent of Node.js and event-driven programming, the use of JS in server-side applications has been steadily growing (Tilkov & Vinoski, 2010). While the browser as an execution platform has significant limitations, server-side JS has full access to the filesystem, native operating system kernels, and existing C and C++ libraries.

To support the use-case of server-side ML in JS, we provide a Node.js backend that binds to the official TensorFlow C API using the N-API (Nodejs.org, 2017). While the internal backend has a different implementation than the WebGL backend, they share the same user-facing API, enabling full portability between the server and the web browser.

The Node.js backend has distinct advantages. Since Node.js and Google's V8 JS engine exposes finalization APIs, it eliminates the need for manual memory management, reducing the cognitive overhead for our users. Binding to the TensorFlow C library means full advantage of the flexibility and performance of TensorFlow. Under the hood, we utilize hardware acceleration both on the CPU, with AVX instructions, and the GPU with the CUDA and CuDNN libraries (Nickolls et al., 2008). Binding to the TensorFlow C API means that TensorFlow.js will support TPUs (Tensor Processing Units) in a future release of the Node.js binding.

4.3 Future backends

Two new web standards, WebAssembly and WebGPU, have potential to improve TensorFlow.js performance.

WebAssembly is a binary instruction format for the web, designed as a potential target for compilation of languages like C and C++. At present, WebAssembly is enabled in most major browsers. By shipping lower-level instructions, WebAssembly compiled from C can see performance gains over vanilla JS. Moreover, WebAssembly will allow writing SIMD code in C, which speeds up linear algebra by computing dot products of vectors in a single instruction. Currently, WebAssembly does not support SIMD.

WebGPU is the working name for a future web standard and JS API for accelerated graphics and compute. WebGPU provides a more generic way to express parallelizable computation on the GPU, which would allow us to write more optimized linear algebra kernels than the ones with the WebGL backend.

5 Ecosystem Integration

This section describes how TensorFlow.js fits into the broader TensorFlow ecosystem.

5.1 Model Converter

There is a plethora of pre-trained, open-sourced models, developed in TensorFlow, that are targeted for edge devices. TensorFlow.js offers a model converter that can load and execute pre-trained TensorFlow SavedModels and Keras models, allowing these models to be available in JS.

To port an existing model to TensorFlow.js, the user runs a Python script that converts the existing format to the TensorFlow.js web format. TensorFlow.js optimizes the model by pruning unnecessary operations (e.g. training operations) and packs weights into 4MB files, optimizing for browser auto-caching. The user can also quantize the weights, reducing the model size by 4X. After the conversion, in the JS application the user can call *tf.loadModel(url)* to load the model, providing a URL where the web files are hosted.

5.2 Models repo

One of the major benefits of the JS ecosystem is the ease at which JS code and static resources can be shared. TensorFlow.js takes advantage of this by hosting an official repository of useful pretrained models, serving the weights on a publicly available Google Cloud Storage bucket. This

makes it easy for beginners to integrate these models into their existing applications.

Furthermore, to address our core goal of enabling ML beginners, we provide wrapper APIs that hide tensors from the user. The model prediction methods always take native JS objects like DOM elements or primitive arrays and return JS objects that represent human-friendly predictions. Listing 3 shows an example application using PoseNet (Oved, 2018), a model hosted in the repository that computes human pose estimations from an image. Note that the user does not need to use *tf.Tensor* to use the PoseNet model.

One of our core design principles is not to sacrifice functionality for simplicity. For these reasons, we expose APIs to work with tensors for expert users. For these users, these models can be used in a transfer learning setting, enabling personalized applications with on-device training with relatively little user data.

```
const imageElement =
    document.getElementById('person');

// Estimate a single pose from the image.
posenet.estimateSinglePose(imageElement)
    .then(pose => console.log(pose));

Console output:
{
  "score": 0.32,
  "keypoints": [
    {
      "position": {"x": 253.37, "y": 76.29},
      "part": "nose",
      "score": 0.99
    },
    ...
  ]
}
```

Listing 3. An example showing the PoseNet API which allows passing an *HTMLImageElement* to the pose estimate method and returns a JS object with the predictions.

6 Examples and Usage

Since launching TensorFlow.js publicly in March 2018, we have seen excitement about TensorFlow.js in a few different domains which we outline in this section.

6.1 Education and Learning

The in-browser implementation of TensorFlow makes it easy to get started with (Anonymous, 2018). Educators can deploy interactive lessons to students without the additional burden of software installation. One early success we saw with TensorFlow.js in the educational deep learning space was Teachable Machine (Google, 2017), an application that allows visitors to build their own image classifier directly in the browser using their webcam, no coding required. The site saw over 450,000 unique visits, and the GitHub code has over 3000 stars. Another successful education application is GAN Lab (Kahng et al., 2018) [1], a tool that enables interactive visual exploration of Generative Adversarial Networks (GANs). Targeted towards non-experts, it helps users develop a greater intuitive sense of how GANs work during training and inference.

TensorFlow.js extends the deep learning ecosystem to the JS communities that are unfamiliar with Python and C. Dan Shiffman, at NYUs Interactive Telecommunications Program, has made both an online video course on ML with JS, and a JS ML library called ML5 (ml5js.org, 2018), built on top of TensorFlow.js. ML5 aims to provide 'friendly ML for the web' and is geared towards artists, creative coders and students, empowering them to innovate in new ways. The ML5 team identified installation issues as the first barrier that beginners face when trying to approach ML (Shiffman, 2018) and built upon TensorFlow.js to overcome that barrier.

The Deep Learning Practicum at MIT (Abelson & Lao, 2018) is another example of the value of TensorFlow.js in an educational setting. This course provides undergraduate students with hands-on experience building deep learning models in JS while they learn the theory and concepts behind the algorithms they are working with.

We also see a great deal of self-directed learning and exploration using TensorFlow.js. Many people who had a passing interest in ML, but found it difficult to access, use TensorFlow.js as an opportunity to learn about the new technology with fewer barriers to entry. This includes people building simple demos and training their own object recognizers, to those exploring cutting edge topics such as NeuroEvolution (Thebe, 2018) and Reinforcement Learning (Neveu, 2018). These demos were built by the authors to develop their ML skills and also to allow others to easily experiment without the infrastructure typically associated with these types of algorithms. These use cases demonstrate a trade-off between accessibility and performance where a bias towards accessibility is acceptable and even embraced. Many more examples of this type of application have been built and are accessible through our community project gallery [2]: Simple MNIST GAN (Chang, 2018), Emotion Extractor (Sudol, 2018), Next Word Predictor (Malviya, 2018) and more.

6.2 Gestural Interfaces

Real time applications that make use of gestural inputs with the webcam is another area where TensorFlow.js has seen promising results. TensorFlow.js users have built applications that enable sign language to speech translation (Singh,

[1] https://poloclub.github.io/ganlab/

[2] https://github.com/tensorflow/tfjs/blob/master/GALLERY.md

2018), enable individuals with limited motor ability control a web browser with their face (Ramos, 2018), and perform real-time facial recognition and pose-detection (Friedhoff & Alvarado, 2018). Each of these examples is powered by a pre-trained image model, usually MobileNet (Howard et al., 2017), that is fine-tuned for the project, or expose interactive fine-tuning interfaces to the end user.

6.3 Research Dissemination

TensorFlow.js has also enabled ML researchers to make their algorithms more accessible to others. For example, the Magenta.js (Roberts et al., 2018) library provides in-browser access to generative music models developed by the Magenta team and ported to the web with TensorFlow.js. Magenta.js has increased the visibility of their work with their target audience, namely musicians. This has unleashed a wide variety of ML powered music apps built by the community such as Latent Cycles (Parviainen, 2018a) and Neural Drum Machine (Parviainen, 2018b). These and more examples can be found at https://magenta.tensorflow.org/demos.

We have also seen TensorFlow.js used to power interactive client-side ML in web-based scholarly articles (Ha & Schmidhuber, 2018) (Carter & Nielsen, 2017), offering a richer communication medium where dynamic behaviour of models can be demonstrated in real time and manipulated by the reader.

6.4 Numeric Applications

Another category of applications that TensorFlow.js enables is GPU accelerated tools for numerical computation. An example is tfjs-tsne (Pezzotti et al., 2018), a novel linear time approximation of the t-SNE algorithm that runs in the browser. TensorFlow.js's GPU acceleration primitives make it practical to run it interactively in the browser for datasets of tens of thousands of points.

6.5 Desktop and Production Applications

An emerging area where JS has been applied is in desktop and production applications, demonstrating the wide reach of the JS ecosystem.

Node Clinic is an open source Node.js performance profiling tool that recently integrated a TensorFlow.js model to separate CPU usage spikes caused by the user from those caused by Node.js internals (e.g. garbage collection) (Nearform.com, 2018).

Mood.gg Desktop (Farza, 2018), created by a student, is a desktop application powered by Electron, a popular framework for writing cross-platform desktop apps in JS. It uses a TensorFlow.js model trained to detect which character the user is playing in a popular team based game called Overwatch, by looking at the user's screen. This is used to play a custom soundtrack from a music streaming site, Mood.gg, that matches the music to the playing style of the in-game character (e.g. 'death metal' for the character called 'Reaper'). The character prediction from the pixels of the screen happens entirely client-side, preserving the privacy of the player and highlighting a key advantage of client-side ML. The author reports that over 500,000 people use the site.

7 Conclusion and Future Work

TensorFlow.js is a high-performance deep learning toolkit in JS that runs both on the client and the server. It is an accessible on-ramp for deep learning to a community that often focuses on the end user. As such, TensorFlow.js has the potential to greatly broaden the set of people who can take advantage of modern ML techniques. We have already seen a rich variety of applications of TensorFlow.js.

A key technical contribution of TensorFlow.js is the set of techniques used to repurpose the web platform's graphics APIs for high-performance numeric computing while maintaining compatibility with a large number of devices and execution environments.

We believe there are a number of opportunities to extend and enhance TensorFlow.js. Given the rapid progress of browser development, it seems likely that additional GPU programming models may become available. In particular, we see conversations by browser vendors to implement general purpose GPU programming APIs (Apple, 2017) (W3C, 2017) that will make these kinds of toolkits more performant and easier to maintain.

Future work will focus on improving performance, continued progress on device compatibility (particularly mobile devices), and increasing parity with the Python TensorFlow implementation. We also see a need to provide support for full machine learning workflows, including data input, output, and transformation. More generally, we see a broad opportunity for contributing to the burgeoning JS data science ecosystem (Davis, 2018), with the goal of decreasing the difficulty of ML development, increasing participation in the ML community, and allowing new types of applications.

References

Abadi, M., Barham, P., Chen, J., Chen, Z., Davis, A., Dean, J., Devin, M., Ghemawat, S., Irving, G., Isard, M., et al. Tensorflow: a system for large-scale machine learning. In *OSDI*, volume 16, pp. 265–283, 2016.

Abelson, H. and Lao, N. 6.S198 Deep Learning Practicum, 2018. URL https://goo.gl/tp7Xut.

Anonymous. Non-specialists learning machine learning: Motivations, hurdles, and desires. *In Submission*, 2018.

Apple. Next-generation 3d Graphics on the Web, February 2017. URL https://webkit.org/blog/7380/next-generation-3d-graphics-on-the-web/.

Carter, S. and Nielsen, M. Using Artificial Intelligence to Augment Human Intelligence. *Distill*, 2(12):e9, December 2017. ISSN 2476-0757. doi: 10.23915/distill.00009. URL https://distill.pub/2017/aia.

Cazala, J. Synaptic. https://github.com/cazala/synaptic, 2014. Accessed: 2018-08-25.

Chang, D. MNIST GAN, 2018. URL https://mwdchang.github.io/tfjs-gan/.

Chen, L. Keras.js. https://github.com/transcranial/keras-js, 2016. Accessed: 2018-08-25.

Chollet, F. et al. Keras. https://keras.io, 2015.

Dahl, R. Propel. https://github.com/propelml/propel, 2017. Accessed: 2018-08-25.

Davis, A. *Data Wrangling with JavaScript*. Manning Shelter Island, NY, 2018.

Farza. DeepOverwatch combining TensorFlow.js, Overwatch, Computer Vision, and Music. *Medium*, May 2018. URL https://bit.ly/2zAvo1e.

Friedhoff, J. and Alvarado, I. Move Mirror: An AI Experiment with Pose Estimation in the Browser using TensorFlow.js. *TensorFlow Medium*, July 2018. URL https://bit.ly/2JMCEsF.

GitHub.com. The state of the octoverse 2017. https://octoverse.github.com/, 2017. Accessed: 2018-09-28.

Google. Teachable Machine, October 2017. URL https://teachablemachine.withgoogle.com/.

Ha, D. and Schmidhuber, J. World Models. *World Models*, 1(1):e10, March 2018. doi: 10.5281/zenodo.1207631. URL https://worldmodels.github.io/.

Haas, A., Rossberg, A., Schuff, D. L., Titzer, B. L., Holman, M., Gohman, D., Wagner, L., Zakai, A., and Bastien, J. Bringing the web up to speed with webassembly. In *ACM SIGPLAN Notices*, volume 52, pp. 185–200. ACM, 2017.

Hidaka, M., Kikura, Y., Ushiku, Y., and Harada, T. Webdnn: Fastest dnn execution framework on web browser. In *Proceedings of the 2017 ACM on Multimedia Conference*, pp. 1213–1216. ACM, 2017.

Howard, A. G., Zhu, M., Chen, B., Kalenichenko, D., Wang, W., Weyand, T., Andreetto, M., and Adam, H. Mobilenets: Efficient convolutional neural networks for mobile vision applications. *arXiv preprint arXiv:1704.04861*, 2017.

Huenermann, J. neurojs. https://github.com/janhuenermann/neurojs, 2016. Accessed: 2018-08-25.

Jackson, D. Next-generation 3D Graphics on the Web. https://webkit.org/blog/7380/next-generation-3d-graphics-on-the-web/, 2017. Accessed: 2018-08-26.

Kahng, M., Thorat, N., Chau, D. H., Vigas, F., and Wattenberg, M. GAN Lab: Understanding Complex Deep Generative Models using Interactive Visual Experimentation. *IEEE Transactions on Visualization and Computer Graphics*, pp. 11, 2018. ISSN 1077-2626, 1941-0506, 2160-9306. doi: 10.1109/TVCG.2018.2864500. URL http://arxiv.org/abs/1809.01587. arXiv: 1809.01587.

Karpathy, A. ConvNetJS: Deep learning in your browser. http://cs.stanford.edu/people/karpathy/convnetjs, 2014. Accessed: 2018-08-25.

Karpathy, A. REINFORCEjs: Reinforcement learning agents in javascript. https://github.com/karpathy/reinforcejs, 2015. Accessed: 2018-08-25.

Kelly, S. compromise. https://github.com/spencermountain/compromise, 2014. Accessed: 2018-08-25.

Kronos. WebGL. https://www.khronos.org/webgl/, 2011. Accessed: 2018-09-21.

Kwok, K., Webster, G., Athalye, A., and Engstrom, L. Tensorfire. https://tenso.rs/, 2017. Accessed: 2018-08-25.

Maclaurin, D., Duvenaud, D., and Adams, R. P. Autograd: Effortless gradients in numpy. In *ICML 2015 AutoML Workshop*, 2015.

Malviya, R. Next Word Predictor, 2018. URL https://nxt-word.firebaseapp.com/.

McMahan, H. B., Moore, E., Ramage, D., Hampson, S., et al. Communication-efficient learning of deep networks from decentralized data. *arXiv preprint arXiv:1602.05629*, 2016.

Miller, S. Mind. https://github.com/stevenmiller888/mind, 2015. Accessed: 2018-08-25.

ml5js.org. ml5js Friendly Machine Learning For The Web., 2018. URL ml5js.org/index.html.

Nearform.com. Node Clinic - An Open Source Node.js performance profiling suite, 2018. URL https://bit.ly/2Dxtfaq.

Neveu, T. metacar: A reinforcement learning environment for self-driving cars in the browser, August 2018. URL https://github.com/thibo73800/metacar. original-date: 2018-06-06T09:00:41Z.

Nickolls, J., Buck, I., Garland, M., and Skadron, K. Scalable parallel programming with cuda. In *ACM SIGGRAPH 2008 classes*, pp. 16. ACM, 2008.

Nodejs.org. N-API: Next generation Node.js APIs for native modules. https://bit.ly/2Dlv5ew, 2017. Accessed: 2018-09-21.

Olah, C. Neural networks, manifolds, and topology, 2014. URL http://colah.github.io/posts/2014-03-NN-Manifolds-Topology/.

Oved, D. Real-time human pose estimation in the browser with TensorFlow.js. *TensorFlow Medium*, May 2018. URL https://bit.ly/2KMnwgv.

Parviainen, T. Latent Cycles, 2018a. URL https://codepen.io/teropa/details/rdoPbG.

Parviainen, T. Neural Drum Machine, 2018b. URL https://codepen.io/teropa/pen/JLjXGK.

Paszke, A., Gross, S., Chintala, S., Chanan, G., Yang, E., DeVito, Z., Lin, Z., Desmaison, A., Antiga, L., and Lerer, A. Automatic differentiation in pytorch. 2017.

Pedregosa, F., Varoquaux, G., Gramfort, A., Michel, V., Thirion, B., Grisel, O., Blondel, M., Prettenhofer, P., Weiss, R., Dubourg, V., Vanderplas, J., Passos, A., Cournapeau, D., Brucher, M., Perrot, M., and Duchesnay, E. Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, 12:2825–2830, 2011.

Pezzotti, N., Mordvintsev, A., Hollt, T., Lelieveldt, B. P., Eisemann, E., and Vilanova, A. Linear tsne optimization for the web. *arXiv preprint arXiv:1805.10817*, 2018.

Plummer, R. brain.js. https://github.com/BrainJS/brain.js, 2010. Accessed: 2018-08-25.

Ramos, O. Handsfree.js, 2018. URL https://handsfree.js.org/.

Roberts, A., Hawthorne, C., and Simon, I. Magenta.js: A javascript api for augmenting creativity with deep learning. In *Joint Workshop on Machine Learning for Music (ICML)*, 2018.

Shankar, A. and Dobson, W. Eager execution: An imperative, define-by-run interface to tensorflow. 2017. URL https://ai.googleblog.com/2017/10/eager-execution-imperative-define-by.html.

Shiffman, D. ml5: Friendly Open Source Machine Learning Library for the Web. *ADJACENT Issue 3*, June 2018. URL https://bit.ly/2ye021D.

Shreiner, D., Sellers, G., Kessenich, J., and Licea-Kane, B. *OpenGL programming guide: The Official guide to learning OpenGL, version 4.3*. Addison-Wesley, 2013.

Singh, A. alexa-sign-language-translator: A project to make Amazon Echo respond to sign language using your webcam, August 2018. URL https://github.com/shekit/alexa-sign-language-translator. original-date: 2018-07-28T20:04:44Z.

Smilkov, D., Carter, S., Sculley, D., Viegas, F., and Wattenberg, M. Direct-manipulation visualization of deep networks. *ICML Workshop on Visualization in Deep Learning*, 2016.

StackOverflow.com. Stack overflow developer survey 2018. https://insights.stackoverflow.com/survey/2018#technology, 2018. Accessed: 2018-09-28.

Sudol, B. Emotion Extractor, 2018. URL https://brendansudol.com/faces/.

Thebe, A. EvolutionSimulator: Evolution Simulator using NeuroEvolution. Created with Matter.js, p5.js and Tensorflow.js, August 2018. URL https://github.com/adityathebe/evolutionSimulator. original-date: 2018-05-09T12:42:15Z.

Tilkov, S. and Vinoski, S. Node. js: Using javascript to build high-performance network programs. *IEEE Internet Computing*, 14(6):80–83, 2010.

Umbel, C. Natural. https://github.com/NaturalNode/natural, 2011. Accessed: 2018-08-25.

W3C. GPU for the Web Community Group, 2017. URL https://www.w3.org/community/gpu/.

Wagenaar, T. Neataptic. https://github.com/wagenaartje/neataptic, 2017. Accessed: 2018-08-25.

webglstats.com. WebGL Stats. https://webglstats.com/webgl/extension/OES_texture_float, 2018. Accessed: 2018-09-21.

Zasso, M. ml.js. https://github.com/mljs/ml, 2014. Accessed: 2018-08-25.