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).

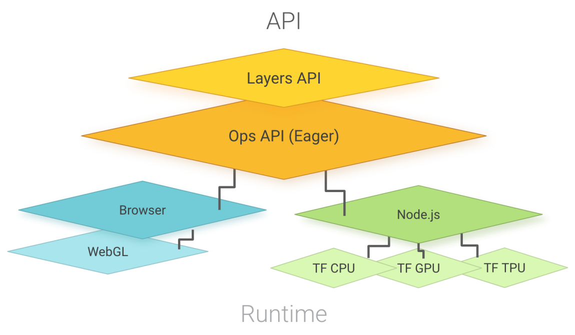

Figure 1. Overview of the TensorFlow.js architecture

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.