MLiTB: Machine Learning in the Browser

Edward Meeds, Remco Hendriks, Said Al Faraby, Magiel Bruntink, Max Welling

With few exceptions, the field of Machine Learning (ML) research has largely ignored the browser as a computational engine. Beyond an educational resource for ML, the browser has vast potential to not only improve the state-of-the-art in ML research, but also, inexpensively and on a massive scale, to bring sophisticated ML learning and prediction to the public at large. This paper introduces MLitB, a prototype ML framework written entirely in Javascript, capable of performing large-scale distributed computing with heterogeneous classes of devices. The development of MLitB has been driven by several underlying objectives whose aim is to make ML learning and usage ubiquitous (by using ubiquitous compute devices), cheap and effortlessly distributed, and collaborative. This is achieved by allowing every internet capable device to run training algorithms and predictive models with no software installation and by saving models in universally readable formats. Our prototype library is capable of training deep neural networks with synchronized, distributed stochastic gradient descent. MLitB offers several important opportunities for novel ML research, including: development of distributed learning algorithms, advancement of web GPU algorithms, novel field and mobile applications, privacy preserving computing, and green grid-computing. MLitB is available as open source software.

MLitB: MACHINE LEARNING in the BROWSER

Edward Meeds, Remco Hendriks, Said Al Faraby, Magiel Bruntink, and Max Welling

Abstract—With few exceptions, the field of Machine Learning (ML) research has largely ignored the browser as a computational engine. Beyond an educational resource for ML, the browser has vast potential to not only improve the state-of-the-art in ML research, but also, inexpensively and on a massive scale, to bring sophisticated ML learning and prediction to the public at large. This paper introduces MLitB, a prototype ML framework written entirely in Javascript, capable of performing large-scale distributed computing with heterogeneous classes of devices. The development of MLitB has been driven by several underlying objectives whose aim is to make ML learning and usage ubiquitous (by using ubiquitous compute devices), cheap and effortlessly distributed, and collaborative. This is achieved by allowing every internet capable device to run training algorithms and predictive models with no software installation and by saving models in universally readable formats. Our prototype library is capable of training deep neural networks with synchronized, distributed stochastic gradient descent. MLitB offers several important opportunities for novel ML research, including: development of distributed learning algorithms, advancement of web GPU algorithms, novel field and mobile applications, privacy preserving computing, and green grid-computing. MLitB is available as open source software.

Index Terms—Machine learning, Ubiquitous computing, Distributed computing, Client-server systems, Mobile computing, Pervasive computing, Social computing, Crowdsourcing

1 INTRODUCTION

THE field of Machine Learning (ML) is currently lacking a common platform for the development of massively distributed and collaborative computing. As a result, there are impediments to leveraging and reproducing the work of other ML researchers, potentially slowing down the progress of the field. The ubiquity of the browser as a computational engine makes it an ideal platform for the development of massively distributed and collaborative ML. Machine Learning in the Browser (MLitB) is an ambitious software development project whose aim is to bring ML, in all its facets, to an audience that includes both the general public and the research community.

By writing ML models and algorithms in browser-based programming languages, many research opportunities become available. The most obvious is software compatibility: nearly all computing devices can collaborate in the training of ML models by contributing some computational resources to the overall training procedure and, with the same code, can harness the power of sophisticated predictive models on the same devices (see Fig. 1). This ubiquitous ML goal has several important consequences: training ML models can now occur on a massive, even global scale, with minimal cost, and ML research can now be shared and reproduced everywhere, by everyone, making ML models a freely accessible, public good. In Section 2 we describe in more detail our vision for MLitB in terms of three main objectives.

In Section 3 we describe the current state of the MLitB software library, where we also present arguments for using JavaScript and other modern web libraries and utilities. The high-level design is explained in Section 3.2. In Section 4 we describe the prototype ML model of the current version (a Deep Neural Network (DNN) implemented in JavaScript [1], modified for MLitB). We have implemented a synchronized, distributed SGD algorithm for training DNNs. The functionality of MLitB is illustrated by training an image classifier using the CIFAR-10[1] dataset and predicting test image classes.

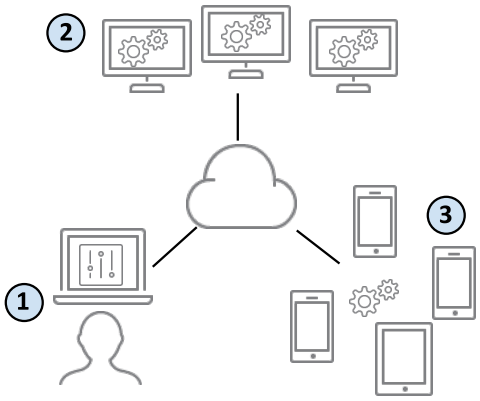

Fig. 1. **Overview of MLitB. (1)** A researcher sets up a learning problem in his/her browser. **(2)** Through the internet, grid and desktop machines contribute computation to solve the problem. **(3)** Heterogeneous devices, such as mobile phone and tablets, connect to the same problem and contribute computation. At any time, connected clients can download the model configuration and parameters, or use the model directly in their browsing environment.

MLitB is influenced and inspired by current volunteer computing projects. These and other related projects, including those from machine learning, are presented in Section 5. Our prototype has exposed several challenges and opportunities for further research; these are presented in Section 6, which is followed by an outline of possible future MLitB development (Section 7), along with several interesting application avenues for MLitB.

• *All authors are with the Informatics Institute, University of Amsterdam, Amsterdam, The Netherlands.*
E-mail: tmeeds@gmail.com

Manuscript received April 19, 2005; revised September 17, 2014.

1. CIFAR Image Datasets: www.cs.toronto.edu/~kriz/cifar.html

2 Vision and Objectives

MLitB is an ongoing research project at the University of Amsterdam with three major objectives that define its long-term vision:

Objective 1: models can be executed in any web browsing environment without further software installation.

Objective 2: distributed computing algorithms can be executed on existing grid, cloud, etc., computing resources with minimal (possibly no) software installation and can be easily managed remotely via the web.

Objective 3: *research closures* that provide, in one universally readable object, a model, its parameters, training algorithms, and so on, that can be executed as part of Objectives 1 and 2.

2.1 Ubiquitous Machine Learning

The *browser* is the most ubiquitous computing device of our time, running, in some shape or form on desktops, laptops, and mobile devices. Software for state-of-the-art ML algorithms and models, on the other hand, are very sophisticated software libraries written in highly specific programming languages within the ML research community ([2], [3], [4]). As research tools, these software libraries have been invaluable. We argue, however, that to make ML truly ubiquitous requires writing ML models and algorithms with web programming languages and using the browser as the engine.

The software we propose can run sophisticated predictive models on cell phones or super-computers; for the former this extends the distributed nature of ML to a global internet. By further encapsulating the algorithms and model together, the benefit of powerful predictive modeling becomes a public commodity.

2.2 Cheap Distributed Computing

The usage of web browsers as compute nodes provides the capability of running sophisticated ML algorithms without the expense and technical difficulty of using custom grid or super-computing facilities (e.g. Hadoop cloud computing [5]). It has long been a dream to use volunteer computing to achieve real massive scale computing. Successes include Seti@Home [6] and protein folding [7]. MLitB is being developed to not only run natively on browsers but also for scaled distributed computing on existing cluster and/or grid resources and, by harnessing the capacity of non-traditional devices, for extremely massive scale computing with a global volunteer base. In the former set-up, low communication overhead and homogeneous devices (a "typical" grid computing solution) can be exploited. In the latter, volunteer computing via the internet opens the scaling possibilities tremendously, albeit at the cost of unreliable compute nodes, variable power, limited memory, etc. Both have serious implications for the user, but, most importantly, both are implemented by the same software.

Although the current version of MLitB does not provide GPU computing, it does not preclude its implementation in future versions. It is therefore possible to seamlessly provide GPU computing when available on existing grid computing resources. Using GPUs on mobile devices is a more delicate proposition since power consumption management is of paramount importance for mobile devices. However, it is possible for MLitB to manage power intelligently by detecting, for example, if the device is plugged in, its temperature, and whether it is actively used for other activities. A user might volunteer periodic "mini-bursts" of GPU power towards a learning problem with minimal disruption to or power consumption from their device. In other words, MLitB will be able to take advantage of the improvements and breakthroughs of GPU computing for web engines and mobile chips, with minimal software development and/or support.

2.3 Reproducible and Collaborative Research

Reproducibility is a difficult yet fundamental requirement for science [8]. Reproducibility is now considered just as essential for high-quality research as peer review; simply providing mathematical representations of models and algorithms is no longer considered acceptable [9]. Furthermore, merely replicating other work, despite its importance, can be given low publication priority [10] even though it is considered a prerequisite for publication. In other words, submissions must demonstrate that their research has been, or could be, independently reproduced.

For ML research there is no reason for not providing working software that allows reproduction of results (for other fields in science, constraints restricting software publication may exist). Currently, the main bottlenecks are the time cost to researchers for making research available, and the incompatibility of the research (i.e. code) for others, which causes an even higher time investment for researchers. One of our primary goals for MLitB is to provide reproducible research with minimal to no time cost to both the primary researcher and other researchers in the community. Following [11], our goal is "setting the default to reproducible."

For ML disciplines, this means other researchers should be able to not only use a model reported in a paper to verify the reported results, but also retrain the model using the reported algorithm. This higher standard is difficult and time-consuming to achieve, but fortunately this approach is being adopted more and more often, in particular by a subdiscipline of machine learning called *deep learning*. In the deep learning community, the introduction of new datasets and competitions, along with innovations in algorithms and modeling, have produced a rapid progress on many ML prediction tasks. Model collections (also called *model zoos*), such as for Caffe [3] make this collaboration explicit and easy to access for researchers. However, there remains a significant time investment to run any particular DNN model (these include compilation, library installations, platform dependencies, GPU dependencies, etc). We argue that these are real barriers to reproducible research and choosing ubiquitous software and compute engines makes it easier. For example, during our testing we converted a very performant computer vision model [12] into JSON and it can now be used on any browser with minimal effort.

In a nod to the closures concept common in functional programming, our approach treats a learning problem as a *research closure*: a single object containing model and algorithm configuration and code along with model parameters

that can be run (and therefore tested and analyzed) by other researchers.

3 MLitB Software Development

MLitB is an ongoing research and development project at the University of Amsterdam, with members from the Software Engineering and Machine Learning groups. The MLitB project and its accompanying software (APIs, libraries, etc.) are built entirely in JavaScript. We have taken a pragmatic software development approach to achieve the objectives we set out in Section 2. To leverage our software development process, we have chosen, wherever possible, well-supported and actively developed external technology. By making these choices we have been able to quickly develop a working MLitB prototype that not only satisfies our objectives, but is also technologically future proof. To demonstrate MLitB on a meaningful ML problem, we have similarly incorporated an existing JavaScript implementation of a Deep Neural Network into MLitB (see Section 4).

The full implementation of the MLitB prototype can be found on GitHub[2].

3.1 Why JavaScript?

JavaScript is a pervasive web programming language, approximately 90% of web-sites use it [13]. This pervasiveness means it is highly supported [14], is actively developed for efficiency ([15], [16]) and functionality. As a result, JavaScript is the most popular programming language on GitHub and its popularity is continuing to grow [17]. JavaScript may soon be the language of choice for enterprise computing ([18], [19]), hinting at its future role as the language for applications traditionally executed on powerful desktops.

The main challenge for scientific computing with JavaScript is the lack of high-quality scientific libraries compared to platforms such as Matlab and Python. With the potential of native computational efficiency (or better, GPU computation) becoming available for JavaScript, it is only a matter of time before JavaScript bridges this gap.

3.2 General Architecture and Design

This section describes the overall architecture of the MLitB prototype and a specific instance of the MLitB for ML is presented in Section 4.

The minimal requirements for MLitB are based on the scenario of running the network as *public resource computing*. The downside of public resource computing is the lack of control over the computing environment. Participants are free to leave (or join) the network at anytime and their connectivity may be variable with high latency. MLitB is designed to be robust to these potentially destabilizing events. The loss of a participant results in the loss of computational power and data allocation. Most importantly, MLitB must robustly handle new and lost clients, re-allocation of data, and client variability in terms of computational power, storage capacity, and network latency.

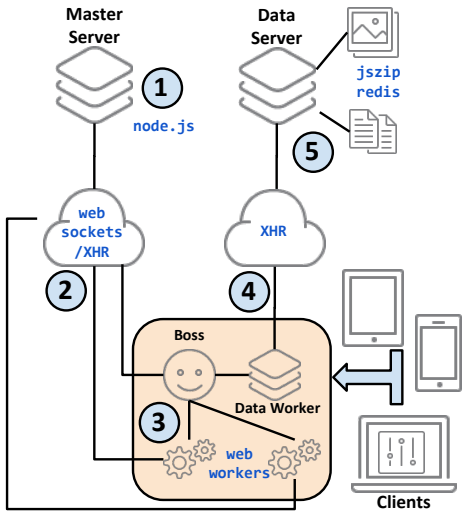

Fig. 2. **MLitB architecture and technologies. (1)** Servers are *Node.js* applications. The *master server* is the main server controlling communication between clients and hosting ML projects. **(2)** Communication between the master server and clients occurs over *web sockets*. **(3)** When heterogeneous devices connect to the master server they use *web workers* to perform different tasks. Upon connection, a UI worker, or boss, is instantiated. Web workers perform all the other tasks on a client and are controlled by the boss. See Fig. 3 for a detailed view of a client's slave workers and their tasks. **(4)** A special data worker on the client communicates with the data server using XHR instead of WebSockets because they communicate large zip-files. **(5)** The *data server*, also a Node.js application, manages uploading of data in *zip* format and serves data vectors to the client data workers. The data server uses specialized Javascript APIs *unzip.js* and *redis-server*

Fig. 2 shows the high-level architecture and web technologies used in MLitB. Modern web browsers provide functionality for two essential aspects of MLitB: Web Workers [20] for parallelizing program execution with threads and Web Sockets [21], for fast bi-directional communication channels to exchange messages more quickly between server and browser.

The general design of MLitB is composed of several parts. A *master server* hosts ML problems/projects and connects clients to them. The master server also manages the *main event loop*, where client triggered events are handled, along with the reduce steps of the map-reduce procedure used for computation. When a browser (i.e. a heterogeneous device) makes an initial connection to the master server, a UI client (aka a *boss*) is instantiated. Through the UI, clients can add *workers* that can perform different tasks (e.g., train a model, download parameters, take a picture, etc). An independent *data server* serves data to clients using zip files and prevents the master server from blocking while serving data. For efficiency, data transfer is performed using XHR[3]. Trained models can be saved into JSON objects at any point in the training process; these can later be loaded in lieu of creating new models.

2. GitHub MLitB repository: https://github.com/software-engineering-amsterdam/MLitB.

3. XMLHttpRequest (XHR) draft specification: www.w3.org/TR/XMLHttpRequest

3.2.1 Master Server

The master node (server) is implemented with the JavaScript library Node.js[4]. Communication between the master and slave nodes is handled by Web Sockets. The master server hosts multiple ML problems/projects simultaneously along with all clients' connections. All processes within the master are event-driven, triggered by actions of the slave nodes. Calling the appropriate functions by slave nodes to the master node is handled by the *router*. The master must efficiently perform its tasks (data reallocation and distribution, reduce-steps) because the clients are idle awaiting new parameters before their next work cycle. New clients must also wait until the end of an iteration before joining a network. The MLitB network is dynamic and permits slave nodes to join and leave during processing. The master monitors its connections and is able to detect lost participants. When this occurs, data that was allocated to the lost client is re-allocated the remaining clients, if possible, otherwise it is marked as *to be allocated*.

3.2.2 Data Server

The data server is also a Node.js application. One task a client can perform is data uploading; currently, the data server loads zipped image classification datasets (where sub-directory names define class labels). Data is downloaded from the data server and zipped files are sent to to clients using XHR and unzipped and processed locally. A redundant cache of data is stored locally in the clients' browser's memory. For example, a client may store 10,000 data vectors, but at each iteration it may only have the computational power to process 100 data vectors in its scheduled iteration duration.

3.2.3 Clients

Clients are browser connections from heterogeneous devices that visit the master server's url. Clients interact through a UI worker, called a *boss*, and can create slave workers to perform various tasks (see Workers, below). Client bosses use a *data worker* to download data from the data server using XHR. Clients therefore require no software installation other than its native browser. Clients can contribute to any project hosted by the master server. Clients can trigger several events through the UI worker. These include adjusting hyper-parameters, adding data, and adding slave workers, etc (Fig. 3 shows all the slave worker tasks). Most tasks are run in a separate WebWorker thread, ensuring a non-blocking and responsive client UI. Data downloading is a special task that, via the boss and the data worker, uses XHR to download from the data server.

3.2.4 Workers

In Fig, 3 the tasks implemented using WebWorker threads are shown. At the highest-level is the client UI, with which the user interacts with ML problems and controls their slave workers. From the client UI, a user can create a new project, load a project from file, upload data to a project, or add slave workers for a project. Slaves can perform several tasks; most important is the *trainer*, which connects to an event loop of a ML project and contributes to its computation (i.e. its map step). To train, the user sets the slave task to train and selects start/restart. This will trigger a join event at the master server; model parameters and data will be downloaded and the slave will begin computation upon completion of the data download. The user can remove a slave at any time.

Other slave tasks are *tracking* a project, which requires receiving model parameters from the master and 1) saving it to file, 2) using it as a predictive model, 3) monitoring error on a dataset over time (as the parameters change).

4. Node.js: http://nodejs.org.

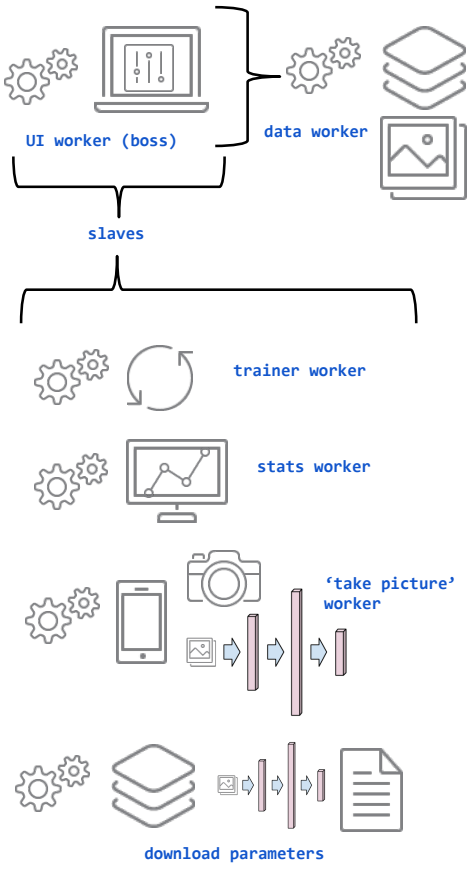

Fig. 3. **MLitB Client Workers.** Each client connection to the master server initiates a *UI worker*, aka a *boss*. For uploading data from a client to the data server and for downloading data from the data server to a client, a separate WebWorker called the *data worker* is used. The data worker and server communicate using XHR and pass zip files in both directions. The boss handles unzipping and decoding data for slaves that request data. Users can add slaves through the UI worker; each slave performs a separate task as using a WebWorker. These include: *training*, *statistics*, *take-a-picture*, and *downloading*. Each slave worker communicates directly to the master server using WebSockets. For the latter three tasks, the communication is mainly for sending requests for models parameters and receiving them. The training slave has more complicated behavior because it must download data then perform computation as part of the main event loop.

3.3 Events and Software Behavior

The MLitB network is constructed as a *master-slave* relationship, with one server and multiple slave client nodes. The setup for computation is similar to a MapReduce network [22]; however, the master server performs many tasks within *iteration* of the *master event loop*, including a reduce step, but also several other important tasks.

The specific tasks will be dictated by events triggered by the client, such as requests for parameters, new client workers, removed/lost clients, etc. Our master event loop can be considered a synchronized map-reduce algorithm with a user defined iteration duration T, where values of T may range from 1 to 30 seconds, depending on the size of the network and the problem. MLitB is not limited to a map-reduce paradigm and in fact we believe that our framework opens the door to peer-to-peer or gossip algorithms [23]. We are currently developing *asynchronous* algorithms to improve the scalability of MLitB.

3.3.1 Master Event Loop

The master event loop consists of five steps and is executed by the master server node as long there is at least one slave node connected. Each loop includes one map-reduce step, and runs for at least T seconds. The following steps are executed, in order:

1) New data uploading and allocation.
2) New client trainer initialization and data allocation.
3) Training workers reduce step.
4) Latency monitoring and data allocation adjustment.
5) Master broadcasts parameters.

3.3.1.1 **New data uploading and allocation**: When a client boss uploads data, it directly communicates with the data server using XHR. Once the data server has uploaded the zip file, it sends the data indices and classification labels to the boss. The boss then registers the indices with the master server. Each data index is managed: MLitB stores an *allocated* index (the worker that is allocated this id) and a *cached* index (the worker that has cached the id). The master ensures that the data allocation is balanced amongst its clients. Once a data set is allocated on the master server, the master allocates indices and sends the set of ids to workers. Workers can then request data from the boss, who in turn use its data downloader worker to download those worker specific ids from the data server. The data server sends a zipped file to the data downloader, which are then unzipped and processed by the boss (e.g. JPEG decoding for images). The zip file transfers are fast but the decoding can be slow. We therefore allow workers to begin computing before the entire dataset is downloaded and decoded, allowing projects to start training almost immediately while data gets cached in the background.

3.3.1.2 **New client trainer initialization and data allocation**: When a client boss adds a new slave, a request to join the project is sent to the master. If there is unallocated data, a balanced fraction of the data is allocated to the new worker. If there is no unallocated data, a pie-cutter algorithm is used to remove allocated data from other clients and assign it to the new client. This prevents unnecessary data transfers. The new worker is sent a set of data ids it will need to download from the client's data worker. Once the data has been downloaded and put into the new worker's cache, the master will then add the new worker to the computation performed at each iteration. The master server is immediately informed when a client or one of its workers is removed from the network. Because of this, it can manage the newly unallocated data (that were allocated to the lost client).

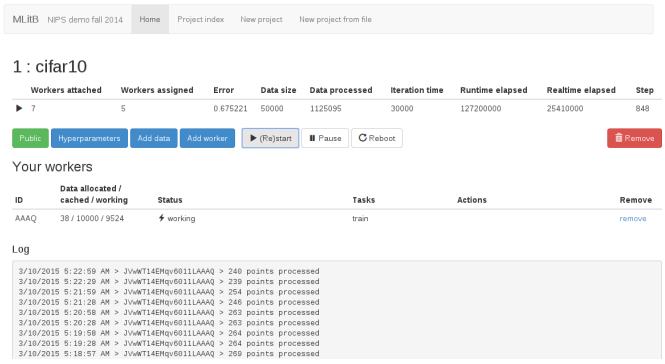

Fig. 4. **CIFAR-10 project loaded in MLitB.**

3.3.1.3 **Training workers' reduce step**: The reduce step is completely problem specific. In our prototype, workers compute gradients with respect to model parameters over their allocated data vectors, and the reduce step sums over the gradients and updates the model parameters.

3.3.1.4 **Latency monitoring and data allocation adjustment**: The interval T represents both the time of computation *and* the latency between the client and the master node. The synchronization is stochastic and adaptive. At each reduce step, the master node estimates the latency between the client and the master and informs the client worker how long it should run for. A client does not need to have a batch size because it just clocks its own computation and returns results at the end of its scheduled work time. Under this setting, it is possible to have mobile devices that compute only a few gradients per second and a powerful desktop machine that performs hundreds or thousands. This simple approach also allows the master to account for unexpected user activity: if the user's device slows or has increased latency, the master will decrease the load on the device for the next iteration. Generally, devices with a cellular network connection communicate with a longer delay than hardwired machines. In practice, this means the reduction step in the master node receives delayed responses from slave nodes, forcing it to run the reduction function after the slowest slave node (with largest latency) has returned. This is called *asynchronous reduction callback delay*.

3.3.1.5 **Master broadcasts parameters**: An array of model parameters is broadcast to each clients' boss worker using XHR; when the boss receives new parameters, they are given to each of its workers who then start another computation iteration.

4 ML Prototype: Deep Neural Networks

The current version of the MLitB software is built around a pervasive ML use-case: deep neural networks (DNNs). DNNs are the current state-of-the-art prediction models for many tasks, including computer vision ([12], [24]), speech recognition [25], and natural language processing and machine translation ([26], [27], [28]).

Our implementation only required *superficial* modifications to an existing Javascript implementation [1] to fit into our network design. We provide a minimalist user interface for designing neural networks; with this UI the

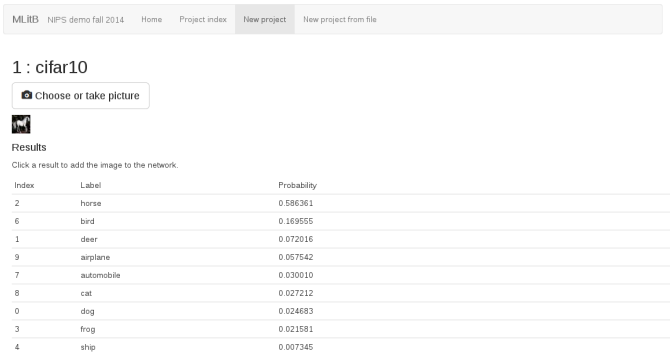

Fig. 5. **Testing an image using the latest NN parameters.**

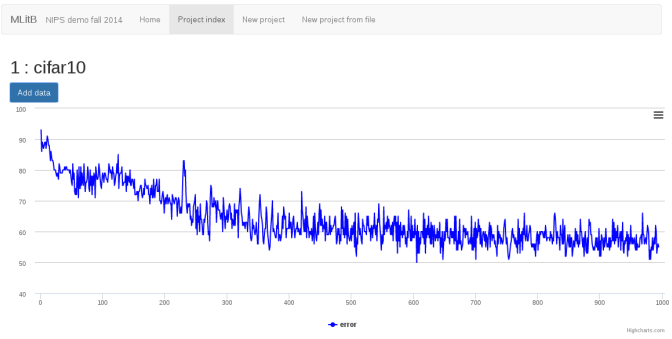

Fig. 6. **Tracking the test error for a NN trained with CIFAR-10.**

user can choose from convolutional layers, traditional NN hidden and outputs layers, learning rates, regularization, including L1/L2 norm, and dropout. Once a NN is specified (or loaded from a previously saved JSON file), it appears in the display, along with other NNs also controlled by the master node. By selecting a specific NN, the user can then add workers and data (e.g. project *cifar10* in Fig. 4). Image classification data is simple to upload using named directory structures for image labels.

In *training* mode, in each *map* step of the main event loop, a training worker performs as many gradient computations as possible within the iteration duration T. The total gradient and the number of gradients is sent to the master, which then in the *reduce* step computes a weighted average of gradients from all workers and takes a gradient step using AdaGrad [29]. At the end of the main event loop, new neural network weights are sent to workers via WebSockets for the next gradient computations.

In *tracking* mode, users can predict class labels for images taken with a device's camera or locally stored images. Users can learn a new classification problem on the fly by taking a picture and giving it a new label; this is treated as a new data vector and a new output neuron is added dynamically to the neural network if the label is also new. Fig. 5 shows a test image being classified by the *cifar10* NN. Users may also create a stats worker and upload test images and track their error over time; after each complete evaluation of the test images, the latest neural network will be downloaded and tested. Fig. 6 shows the error for *cifar10* using a small test set for the first 1000 parameter updates. Finally, users can download the neural network into a JSON file which can then be shared or stored for later use.

The synchronized map-reduce-like algorithm we have implemented has many obvious directions for improvement. Note, however, that we are agnostic to the distributed computation algorithm. In fact, we feel that learning a DNN with heterogeneous devices is both a challenging and necessary endeavor as learning models becomes a global, collaborative effort. In section 7 we describe several possible research directions for our learning scenario.

5 Related Work

MLitB has been influenced by a several different technologies and ideas presented by previous authors and from work in different specialization areas. We briefly summarize this related work below.

5.1 Volunteer Computing

BOINC [30] is an open-source software library used to set up a grid computing network, allowing anyone with a desktop computer connected to the internet to participate; this is called *public resource computing*. Public resource or volunteer computing was popularized by SETI@Home [6], a research project that analyzes radio signals from space in the search of signs of extraterrestrial intelligence. More recently, protein folding has emerged as significant success story [7]. Hadoop [5] is an open-source software system for storing very large datasets and executing user application tasks on large networks of computers. MapReduce [22] is a general solution for performing computation on large datasets using computer clusters.

5.2 Javascript Applications

It has been argued [31] that the performance of the JavaScript engine in modern web browsers has increased so drastically that the number of potential applications now include more complex computations like regular expression matching and binary tree modifications. The same web extensions MLitB uses (Web Workers and Web Sockets) are used to build a computing network of web browsers named *WeevilScout* [31].

ConvNetJS [1] Javascript implementation of a convolutional neural-network; it can be seen as the non-distributed predecessor of MLitB. ConvNetJS was developed primarily for educational purposes, ConvNetJS is a JavaScript library with the capacity of building diverse neural networks to run in a single web browser and trained using stochastic gradient descent (SGD) [32].

5.3 Distributed Machine Learning

The most performant deep NN models are trained with sophisticated scientific libraries written for GPUs ([3], [4], [33]) that provide orders of magnitude computational speed-ups compared to CPUs. Each implements some form of stochastic gradient descent (SGD) [32] as the training algorithm. Most implementations are limited to running on the cores of a single machine and therefore by the memory limitations of the GPU. Exceptionally, there are distributed deep learning algorithms for using a farm of GPUs (e.g.

Downpour SGD [34]) and using farms of commodity servers (e.g. *COTS-HPS* [35]). Other distributed ML algorithm research includes the parameter server model [36], parallelized SGD [37], and distributed SGD [38]. Note that our system can push commodity computing to the extreme using pre-existing devices, some of which may be GPU capable, with and without an organization's existing computing infrastructure. As we discuss below, there are still many open research questions and opportunities for distributed ML algorithm research.

6 Opportunities and Challenges

In tandem with our vision, there are several directions the next version of MLitB can take, both in terms of the library itself and the potential kinds of applications a ubiquitous ML framework like MLitB can offer. We begin with the challenges, and conclude with the opportunities and the next steps for MLitB.

6.1 Challenges

There are several key challenges that future MLitB development needs to address, most notable memory and bandwidth limitations on mobile devices. We believe that each of these can be solved to various degrees.

6.1.1 Memory Limitations

State-of-the-art Neural Network models have huge numbers of parameters. This prevents them from fitting onto mobile devices. There are two possible solutions to this problem. The first solution is to learn smaller neural networks. Recently smaller NN models have shown promise on image classification performance. In particular the Network in Network [12] model from the Caffe model zoo, is 16MB, and outperforms AlexNet which is 256MB [3]. Another solution is to distribute the NN (during training and prediction) across clients. An example of this approach is Downpour SGD [34].

6.1.2 Communication Overhead

With large models, large of numbers of parameters are communicated regularly. This is a similar issue to memory limitation and could benefit from the same solutions. However, given a fixed bandwidth and asynchronous parameter updates, we can ask what parameter updates (from master to client) and which gradients (from client to master) should be communicated. An algorithm could transmit a random subset of the weight gradients, or send the most informative. In other words, with a fixed bandwidth, maximize the information transferred per iteration.

6.1.3 Performance Efficiency

Perhaps the biggest argument against scientific computing with JavaScript is its performance efficiency. We disagree that this should prevent the widespread adoption of browser-based, scientific computing because the goal of several groups to achieve native performance in JavaScript ([15], [16]) and GPU kernels are becoming part of existing web engines (e.g. WebCL[5]) and they can be seamlessly incorporated into existing JavaScript libraries, though they have yet to be written for ML.

5. WebCL by Kronos: www.khronos.org/webcl.

6.2 Opportunities

6.2.1 Massively Distributed Learning Algorithms

The challenges presented are opportunities for the development of distributed training algorithms for machine learning models at a truly global scale. A synchronized event loop will no longer be sufficient. Instead, asynchronous, distributed algorithms and models will be required. This is a largely untapped research area for ML, but are currently being developed for the next version of MLitB.

6.2.2 Field Research

Moving data collection and predictive models onto mobile devices makes is easy to bring models into the field. Connecting users with mobile devices to powerful NN models can aid field research by bringing the predictive models to the field, e.g. for fast labeling and data gathering. For example, a pilot program of crop surveillance in Uganda currently uses bespoke computer vision models for detecting pestilence (insect eggs, leaf diseases, etc) [39]. Projects like these could leverage publicly available, state-of-the-art computer vision models to bootstrap their field research.

6.2.3 Privacy Preserving Computing and Mobile Health

Our MLitB framework provides a natural platform for the development of real privacy-preserving applications [40] by naturally protecting user information contained on mobile devices, yet allowing the data to be used for valuable model development. The current version of MLitB does not provide privacy preserving algorithms such as [41], but these could be easily incorporated into MLitB. It would therefore be possible for a collection of personal devices to collaboratively train machine learning models using sensitive data stored locally and with modified training algorithms that guarantee privacy. One could imagine, for example, using privately stored images of a skin disease to build a classifier based on large collection of disease exemplars, yet with the data always kept on each patient's mobile device, thus never shared, and trained using privacy preserving algorithms.

6.2.4 Green Computing

One of our main objectives was to provide simple, cheap, distributed computing capability with MLitB. Because MLitB runs with minimal software installation (in most cases requiring none), it is possible to use this framework for low-power consumption distributed computing. By using existing organizational resources running in low-energy states (dormant or near dormant) MLitB can wake the machines, perform some computing cycles, and return them to their low-energy states. This is in stark contrast to a data center approach which has near constant, heavy energy usage [42].

7 Future MLitB Development

The next phases of development will focus on the following directions: a visual programming user interface for model configuration, development of a library of ML models and algorithms, development of performant scientific libraries in JavaScript with and without GPUs, and model archiving with the development of a research closure specification.

7.1 Visual Programming

Many ML models are constructed as chains of processing modules. This lends itself to a visual programming paradigm, where the chains can be constructed by dragging and dropping modules together. This way models can be visualized and compared, dissected, etc. Algorithms are tightly coupled to the model and a visual representation of the model can allow interaction with the algorithm as it proceeds. For example, learning rates for each layer can be adjusted while monitoring error rates (even turned off for certain layers), or training modules can be added to improve learning of hidden layers for very deep neural networks, as done in [43]. With a visual UI it would be easy to pull in other existing, pre-trained models, remove parts, and train on new data. For example, a researcher could start with a pre-trained image classifier, remove the last layer, and easily train a new image classifier, taking advantage of an existing, generalized image representation model.

7.2 Machine Learning Library

We currently have built a prototype around an existing JavaScript implementation of DNNs [1]. In the near future we plan on implementing other models (e.g. latent Dirichlet allocation) and algorithms (e.g. distributed MCMC [38]). MLitB is agnostic to learning algorithms and therefore is a great platform for researching novel distributed learning algorithms.

7.3 GPU implementations

Implementation of GPU kernels can bring MLitB performance up to the level of current state-of-the-art scientific libraries such as Theano ([2], [33]) and Caffe [3], while retaining the advantages of using heterogeneous devices. For example, balancing computational loads during training is very simple in MLitB and any learning algorithm can be shared by GPU powered desktops and mobile devices. Smart phone could be part of the distributed computing process by permitting the training algorithms to use short bursts of GPU power for their calculations, and therefore limiting battery drain and user disruption.

7.4 Research closures

MlitB can save and load JSON model configurations and parameters, allowing researchers to share and build upon other researchers' work. However, it does not quite achieve our goal of a research closure where all aspects—code, configuration, parameters, etc–are saved into a single object. In addition to research closures, we hope to develop a model zoo, akin to Caffe's for posting and sharing research. Finally, some kind of system for verifying models, like recomputation.org, would further strengthen the case for MLitB being truly reproducible (and provide backwards compatibility).

8 Conclusion

In this paper we have introduced MLitB: Machine Learning in the Browser, an alternative framework for ML research based entirely on using the browser as the computational engine. MLitB is based upon three overarching objectives of providing *ubiquitous ML* capability to every computing device, *cheap distributed computing*, and *reproducible research*.

Our current implementation is written entirely in Javascript and makes extensive use of existing JavaScript libraries, including Node.js for servers, Web Workers for non-blocking computation, and Web Sockets for communication between clients and servers. A prototype version of MLitB demonstrates the potential of this approach on a ML test case: an implementation of Deep Neural Networks trained using Stochastic Gradient Descent using heterogenous devices, including dedicated grid-computing resources and mobile devices, using the same interface and with no client-side software installation. Clients simply connect to the server and computing begins. This test case has provided valuable information for future versions of MLitB, exposing both existing challenges and interesting research and application opportunities. We have also advocated for a framework which supports reproducible research; MLitB naturally provides this by allowing models and parameters to be saved to a single object which can be reloaded by all other researchers and used immediately.

Acknowledgments

The authors acknowledge funding support from Amsterdam Data Science and computing resources from SurfSara.

References

[1] A. Karpathy, "ConvNetJS Deep Learning in the browser," http://www.convnetjs.com, 2014, accessed: 2014-07-09.

[2] F. Bastien, P. Lamblin, R. Pascanu, J. Bergstra, I. Goodfellow, A. Bergeron, N. Bouchard, D. Warde-Farley, and Y. Bengio, "Theano: new features and speed improvements," *arXiv preprint arXiv:1211.5590*, 2012.

[3] Y. Jia, E. Shelhamer, J. Donahue, S. Karayev, J. Long, R. Girshick, S. Guadarrama, and T. Darrell, "Caffe: Convolutional architecture for fast feature embedding," *arXiv preprint arXiv:1408.5093*, 2014.

[4] R. Collobert, K. Kavukcuoglu, and C. Farabet, "Torch7: A matlab-like environment for machine learning," in *BigLearn, NIPS Workshop*, no. EPFL-CONF-192376, 2011.

[5] K. Shvachko, H. Kuang, S. Radia, and R. Chansler, "The Hadoop distributed file system," in *Mass Storage Systems and Technologies (MSST), 2010 IEEE 26th Symposium on*. IEEE, 2010, pp. 1–10.

[6] D. P. Anderson, J. Cobb, E. Korpela, M. Lebofsky, and D. Werthimer, "SETI@Home: an experiment in public-resource computing," *Communications of the ACM*, vol. 45, no. 11, pp. 56–61, 2002.

[7] T. J. Lane, D. Shukla, K. A. Beauchamp, and V. S. Pande, "To milliseconds and beyond: challenges in the simulation of protein folding," *Current opinion in structural biology*, vol. 23, no. 1, pp. 58–65, 2013.

[8] R. FitzJohn, M. Pennell, A. Zanne, and W. Cornwell, "Reproducible research is still a challenge," http://ropensci.org/blog/2014/06/09/reproducibility/, 2014, blog post: June 9, 2014.

[9] V. Stodden, P. Guo, and Z. Ma, "Toward reproducible computational research: an empirical analysis of data and code policy adoption by journals," *PloS one*, vol. 8, no. 6, p. e67111, 2013.

[10] A. Casadevall and F. C. Fang, "Reproducible science," *Infection and immunity*, vol. 78, no. 12, pp. 4972–4975, 2010.

[11] V. Stodden, J. Borwein, and D. H. Bailey, ""setting the default to reproducible in computational science research," http://www.siam.org/news/news.php?id=2078, 2013, siam news blog post: June 3, 2013.

[12] M. Lin, Q. Chen, and S. Yan, "Network in network," *arXiv preprint arXiv:1312.4400*, 2013.

[13] W3Techs, "Usage of JavaScript for websites," http://w3techs.com/technologies/details/cp-javascript/all/all, 2014, accessed: 2014-06-17.

[14] Can I Use, "Javascript API support comparison," http://caniuse.com/#cats=JS_API, 2014, accessed: 2014-11-26.

[15] C. JavaScript V8, https://developers.google.com/v8, 2014, accessed: 2014-11-26.

[16] M. asm.js, http://asmjs.org, 2014, accessed: 2014-11-26.

[17] D. Berkholz, "Github language trends and the fragmenting landscape," http://redmonk.com/dberkholz/2014/05/02/github-language-trends-and-the-fragmenting-landscape, 2014, redMonk blog post: May 2, 2014.

[18] K. Cagle, "Why your next programming language will be javascript," http://semanticmodeling.blogspot.nl/2013/02/why-your-next-programming-language-will.html, 2013, semantic Modeling blog post: February 12, 2013.

[19] N. Wright, "Why javascript will become the dominant programming language of the enterprise," http://readwrite.com/2013/08/09/why-javascript-will-become-the-dominant-programming-language-of-the-enterprise, 2014, readWrite blog post: August 9, 2013.

[20] W3C, "Web workers, editor's draft 19 may 2014," http://dev.w3.org/html5/workers, 2014, accessed: 2014-12-04.

[21] IETF, "The websocket protocol," http://tools.ietf.org/html/rfc6455, 2011, accessed: 2014-12-04.

[22] J. Dean and S. Ghemawat, "MapReduce: simplified data processing on large clusters," *Communications of the ACM*, vol. 51, no. 1, pp. 107–113, 2008.

[23] S. Boyd, A. Ghosh, B. Prabhakar, and D. Shah, "Randomized gossip algorithms," *Information Theory, IEEE Transactions on*, vol. 52, no. 6, pp. 2508–2530, 2006.

[24] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," in *Advances in neural information processing systems*, 2012, pp. 1097–1105.

[25] G. Hinton, L. Deng, D. Yu, G. E. Dahl, A.-r. Mohamed, N. Jaitly, A. Senior, V. Vanhoucke, P. Nguyen, T. N. Sainath *et al.*, "Deep neural networks for acoustic modeling in speech recognition: The shared views of four research groups," *Signal Processing Magazine, IEEE*, vol. 29, no. 6, pp. 82–97, 2012.

[26] S. Liu, N. Yang, M. Li, and M. Zhou, "A recursive recurrent neural network for statistical machine translation," in *Proceedings of ACL*, 2014, pp. 1491–1500.

[27] D. Bahdanau, K. Cho, and Y. Bengio, "Neural machine translation by jointly learning to align and translate," *arXiv preprint arXiv:1409.0473*, 2014.

[28] I. Sutskever, O. Vinyals, and Q. V. Le, "Sequence to sequence learning with neural networks," *arXiv preprint arXiv:1409.3215*, 2014.

[29] J. Duchi, E. Hazan, and Y. Singer, "Adaptive subgradient methods for online learning and stochastic optimization," *The Journal of Machine Learning Research*, vol. 12, pp. 2121–2159, 2011.

[30] D. P. Anderson, "BOINC: A system for public-resource computing and storage," in *Proceedings of the Workshop on Grid Computing, Nov. 2004*. IEEE, 2004, pp. 4–10.

[31] R. Cushing, G. H. H. Putra, S. Koulouzis, A. Belloum, M. Bubak, and C. De Laat, "Distributed computing on an ensemble of browsers," *Internet Computing, IEEE*, vol. 17, no. 5, pp. 54–61, 2013.

[32] L. Bottou, "Large-scale machine learning with stochastic gradient descent," in *Proceedings of COMPSTAT'2010*. Springer, 2010, pp. 177–186.

[33] J. Bergstra, O. Breuleux, F. Bastien, P. Lamblin, R. Pascanu, G. Desjardins, J. Turian, D. Warde-Farley, and Y. Bengio, "Theano: a CPU and GPU math expression compiler," in *Proceedings of the Python for Scientific Computing Conference (SciPy)*, Jun. 2010, oral Presentation.

[34] J. Dean, G. Corrado, R. Monga, K. Chen, M. Devin, M. Mao, A. Senior, P. Tucker, K. Yang, Q. V. Le *et al.*, "Large scale distributed deep networks," in *Advances in Neural Information Processing Systems*, 2012, pp. 1223–1231.

[35] A. Coates, B. Huval, T. Wang, D. Wu, B. Catanzaro, and N. Andrew, "Deep learning with cots hpc systems," in *Proceedings of The 30th International Conference on Machine Learning*, 2013, pp. 1337–1345.

[36] M. Li, D. G. Andersen, J. W. Park, A. J. Smola, A. Ahmed, V. Josifovski, J. Long, E. J. Shekita, and B.-Y. Su, "Scaling distributed machine learning with the parameter server," in *Proc. OSDI*, 2014, pp. 583–598.

[37] M. Zinkevich, M. Weimer, L. Li, and A. J. Smola, "Parallelized Stochastic Gradient Descent," in *Advances in Neural Information Processing Systems*, 2010, pp. 2595–2603.

[38] S. Ahn, B. Shahbaba, and M. Welling, "Distributed Stochastic Gradient MCMC," *ICML*, 2014.

[39] J. A. Quinn, K. Leyton-Brown, and E. Mwebaze, "Modeling and monitoring crop disease in developing countries." in *AAAI*, 2011.

[40] C. Dwork, "Differential privacy: A survey of results," in *Theory and Applications of Models of Computation*. Springer, 2008, pp. 1–19.

[41] S. Han, W. K. Ng, L. Wan, and V. Lee, "Privacy-preserving gradient-descent methods," *Knowledge and Data Engineering, IEEE Transactions on*, vol. 22, no. 6, pp. 884–899, 2010.

[42] "Data center efficiency assessment," http://www.nrdc.org/energy/files/data-center-efficiency-assessment-IP.pdf, 2014, issue Paper IP:14-08-A, August 2014.

[43] C. Szegedy, W. Liu, Y. Jia, P. Sermanet, S. Reed, D. Anguelov, D. Erhan, V. Vanhoucke, and A. Rabinovich, "Going deeper with convolutions," *CoRR*, vol. abs/1409.4842, 2014.