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[1], **Remco Hendriks**[1], **Said Al Faraby**[1], **Magiel Bruntink**[1], **and Max Welling**[1,2,3]

[1]**Informatics Institute, University of Amsterdam**
[2]**Donald Bren School of Information and Computer Sciences, University of California, Irvine**
[3]**Canadian Institute for Advanced Research**

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.

Keywords: Machine learning, Ubiquitous computing, Distributed computing, Client-server systems, Mobile computing, Pervasive computing, Social computing, Crowdsourcing

1 INTRODUCTION

The field of Machine Learning (ML) currently lacks 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 can, with the same code, harness the power of sophisticated predictive models on the same devices (see Fig. 1). This goal of ubiquitous ML 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 this paper, we present both a long-term **vision** for MLitB and a light-weight **prototype** implementation of MLitB, that represents a first step in completing the vision, and is based on an important ML use-case, Deep Neural Networks.

In Section 2 we describe in more detail our vision for MLitB in terms of three main objectives: 1) make ML models and algorithms ubiquitous, for both the public and the scientific community, 2) create an framework for cheap distributed computing by harnessing existing infrastructure and personal devices as novel computing resources, and 3) design *research closures*, software objects that archive ML models, algorithms, and parameters to be shared, reused, and in general, support reproducible research.

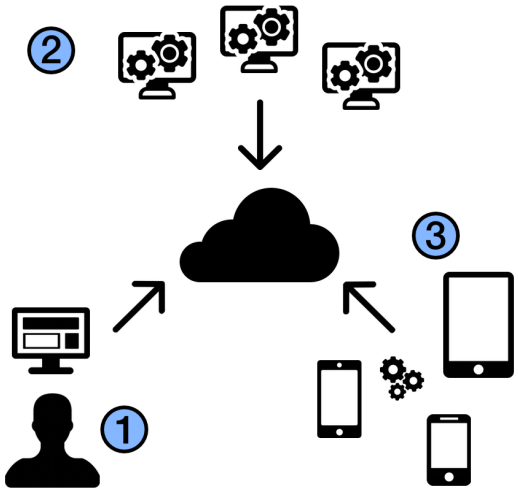


Figure 1. Overview of MLitB. **(1)** A researcher sets up a learning problem in his/her browser. **(2)** Through the internet, grid and desktop machines contribute computation to solve the problem. **(3)** Heterogeneous devices, such as mobile phone and tablets, connect to the same problem and contribute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

In Section 3 we describe the current state of the MLitB software implementation, the MLitB prototype. We begin with a description of our design choices, including arguments for using JavaScript and the other modern web libraries and utilities. Then we describe a bespoke map-reduce synchronized event-loop, specifically designed for training a large class of ML models using distributed stochastic gradient descent (SGD). Our prototype focuses on a specific ML model, Deep Neural Networks (DNNs), using an existing JavaScript implementation (Karpathy, 2014), modified only slightly for MLitB. We also report results of a scaling experiment, demonstrating the feasibility, but also the engineering challenges of using browsers for distributed ML applications. We then complete the prototype description with a walk-through of using MLitB to specify and train a neural network for image classification.

MLitB is influenced and inspired by current volunteer computing projects. These and other related projects, including those from machine learning, are presented in Section 4. Our prototype has exposed several challenges requiring further research and engineering; these are presented in Section 5, along with discussion of interesting application avenues MLitB makes possible. The most urgent software development directions follow in Section 6.

2 MLITB: VISION

Our long-term vision for MLitB is guided by three overarching objectives:
Ubiquitous ML: models can be training and executed in any web browsing environment without any further software installation.
Cheap distributed computing: algorithms can be executed on existing grid, cloud, etc., computing resources with minimal (and possibly no) software installation, and can be easily managed remotely via the web; additionally, small internet enabled devices can contribute computational resources.
Reproducibility: MLitB should foster reproducible science with *research closures*, universally readable objects containing ML model specifications, algorithms, and parameters, that can be used seamlessly to achieve the first two objectives, as well as support sharing of ML models and collaboration within the research community and the public at large.

2.1 Ubiquitous Machine Learning

The *browser* is the most ubiquitous computing device of our time, running, in some shape or form on all 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 (Bastien et al., 2012; Jia et al., 2014; Collobert et al., 2011). 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 computational 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 (Shvachko et al., 2010)). It has long been a dream to use volunteer computing to achieve real massive scale computing. Successes include Seti@Home (Anderson et al., 2002) and protein folding (Lane et al., 2013). 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 connected to a power source, 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 (McNutt, 2014). 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 (Stodden et al., 2013b). Furthermore, merely replicating other work, despite its importance, can be given low publication priority (Casadevall and Fang, 2010) 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 further increases the 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 (Stodden et al., 2013a), we support "setting the default to reproducible."

For ML disciplines, this means other researchers should not only be able to 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 sub-discipline 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 those built with Caffe (Jia et al., 2014) make this collaboration explicit and easy to access for researchers. However, there remains a significant time investment to run any particular deep learning 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 (Lin et al., 2013) into JSON format and it can now be used on any browser with minimal effort.[1]

In a nod to the concept of closures concept common in functional programming, our approach treats a learning problem as a *research closure*: a single object containing model and algorithm configuration plus code, along with model parameters that can be executed (and therefore tested and analyzed) by other researchers.

3 MLITB: PROTOTYPE

The MLitB project and its accompanying software (application programming interfaces (APIs), libraries, etc.) are built entirely in JavaScript. We have taken a pragmatic software development approach to achieve as much of our vision as possible. 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 many of our objectives, but is as technologically future proof as possible. To demonstrate MLitB on a meaningful ML problem, we have similarly incorporated an existing JavaScript implementation of a Deep Neural Network into MLitB. The full implementation of the MLitB prototype can be found on GitHub[2].

3.1 Why JavaScript?

JavaScript is a pervasive web programming language, embedded in approximately 90% of web-sites (W3Techs, 2014). This pervasiveness means it is highly supported (Can I Use, 2014), and is actively developed for efficiency and functionality (JavaScript V8, 2014; asm.js, 2014). As a result, JavaScript is the most popular programming language on GitHub and its popularity is continuing to grow (Ray et al., 2014).

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. A recent set of benchmarks showed that numerical JavaScript code can be competitive with native C (Khan et al., 2014).

3.2 General Architecture and Design

Design Considerations

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.

Although we are agnostic to the specific technologies used to fulfill the *vision* of MLitB, in practice we are guided by both the requirements of MLitB and our development constraints. Therefore, as a first step towards implementing our vision, we chose technology pragmatically. Our choices also follow closely the design principles for web-based big data applications (Begoli and Horey, 2012), which recommend popular standards and light-weight architectures. As we will see, some of our choices may be limiting at large scale, but they have permitted a successful small-scale MLitB implementation (with up to 100 clients).

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 (W3C, 2014) for parallelizing program execution with threads and Web Sockets (IETF, 2011) for fast bi-directional communication channels to exchange messages more quickly between server and browser. To maintain compatibility across browser vendors, there is little choice for alternatives to Web Workers and Web Sockets. These same choices are also used in another browser-based distributed computing platform (Cushing et al., 2013).

[1] JavaScript Object Notation `json.org/`

[2] `https://github.com/software-engineering-amsterdam/MLitB`

On the server-side, there are many choices that can be made based on scalability, memory management, etc. However, we chose Node.js for the server application.[3] Node.js provides several useful features for our application: it is lightweight, written in JavaScript, handles events asynchronously, and can serve many clients concurrently (Tilkov and Vinoski, 2010). Asynchronous events occur naturally in MLitB as clients join/leave the network, client computations are received by the server, users add new models and otherwise interact with the server. Since the main computational load is carried by the clients, and not the server, a light-weight server that can handle many clients concurrently is all that is required by MLitB.

Design Overview

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 a (bespoke) map-reduce procedure used for computation. When a browser (i.e. a heterogeneous device) makes an initial connection to the master server, a user-interface (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[4]. 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.

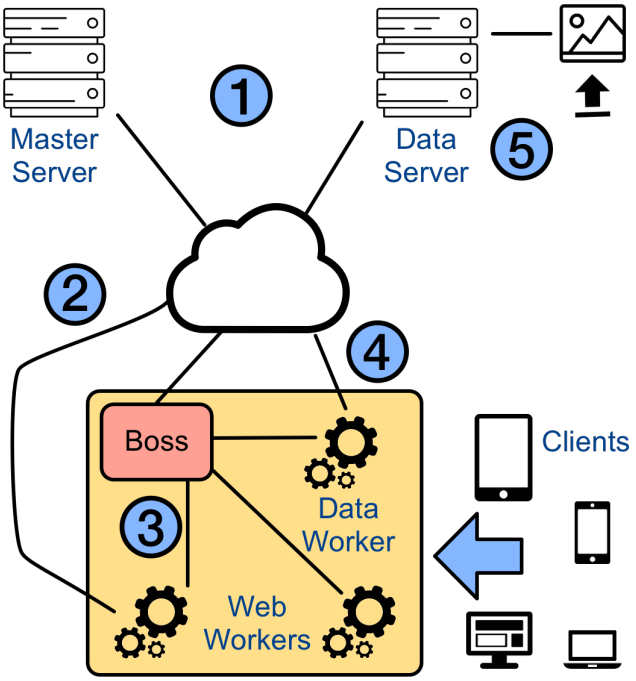


Figure 2. MLitB architecture and technologies. (1) Servers are *Node.js* applications. The *master server* is the main server controlling communication between clients and hosts ML projects. **(2)** Communication between the master server and clients occurs over Web Sockets. **(3)** When heterogeneous 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 details. **(4)** A special data worker on the client communicates with the data server using XHR. **(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.

[3]Node.js: `http://nodejs.org`.
[4]XMLHttpRequest `www.w3.org/TR/XMLHttpRequest`

Master Server

The master node (server) is implemented in Node.js with communication between the master and slave nodes 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*.

Data Server

The data server is a bespoke application intended to work with our neural network use-case model and can be thought of a lightweight replacement for a proper image database. The data server is an independent Node.js application that can, but does not necessarily live on the same machine. Users upload data in zip files before training begins; currently, the data server handles zipped image classification datasets (where sub-directory names define class labels). Data is then downloaded from the data server and zipped files are sent to clients using XHR and unzipped and processed locally. XHR is used instead of WebSockets because they communicate large zip-files more efficiently. 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. The data server uses specialized JavaScript APIs *unzip.js* and *redis-server*.

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). The boss is the main worker running in a client's browser. It manages the slave and image download worker and functions as a bridge between the downloader and slaves. A simple wrapper handles UI interactions, and provides input/output to the boss. Client bosses use a *data worker* to download data from the data server using XHR. 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. 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). Most tasks are run in a separate Web Worker thread (including the boss), 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.

Workers

In Fig. 3 the tasks implemented using Web Worker 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). Each slave worker communicates directly to the master server using Web Sockets. 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. 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*, which requires receiving model parameters from the master, and allows users to monitor statistics of the model on a dataset (e.g. classification error) or to execute the model (e.g. classify an image on a mobile device). Each slave worker communicates directly to the master server using Web Sockets.

Figure 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 Web Worker called the *data worker* is used. Users can add slaves through the UI worker; each slave performs a separate task using a Web Worker.

3.3 Events and Software Behavior

The MLitB network is constructed as a *master-slave* relationship, with one server and multiple slave nodes (clients). The setup for computation is similar to a MapReduce network (Dean and Ghemawat, 2008); however, the master server performs many tasks during an *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 as 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 (Boyd et al., 2006). We are currently developing *asynchronous* algorithms to improve the scalability of MLitB.

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:

a) New data uploading and allocation.

b) New client trainer initialization and data allocation.

c) Training workers reduce step.

d) Latency monitoring and data allocation adjustment.

e) Master broadcasts parameters.

a) 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 the 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.

b) 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.[5] Because of this, it can manage the newly unallocated data (that were allocated to the lost client).

c) 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.

d) 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 longer delays 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*.

e) 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.

3.4 ML use-case: 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

[5] If a user closes a client tab, the master will know immediately and take action. In the current implementation, if a user closes the master tab, all current connections are lost.

vision (Krizhevsky et al., 2012; Lin et al., 2013), speech recognition (Hinton et al., 2012), and natural language processing and machine translation (Liu et al., 2014; Bahdanau et al., 2014; Sutskever et al., 2014). Our implementation only required *superficial* modifications to an existing JavaScript implementation (Karpathy, 2014) to fit into our network design.

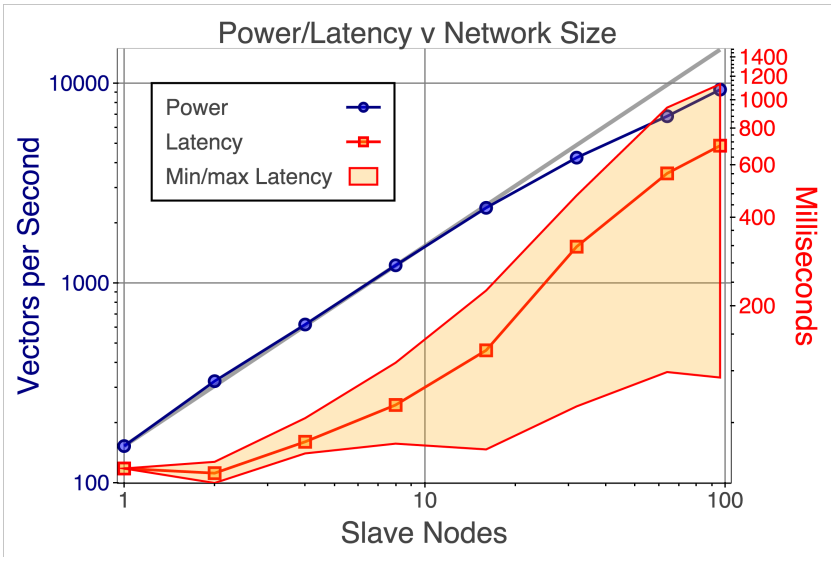


Figure 4. Effects of scaling on power and latency. Power—measured as the number of data vectors processed per second—scales linearly until 64 nodes, when the increase in latency jumps. The ideal linear scaling is shown in grey.

3.5 Scaling Behavior of MLitB

We performed an experiment to study the scaling behavior of MLitB prototype. Using up to 32 4-core workstation machines connected on a local area network using a single router, we trained a simple convolutional NN on the MNIST dataset for 100 iterations (with 4 seconds per iteration/synchronization event).[6] The number of slave nodes doubled from one experiment to the next (i.e. $1,2,4,\ldots,96$). We are interested in the scaling behavior of two performance indicators: 1) power, measured in data vectors processed per second, and 2) latency in milliseconds between slaves and master node. Of secondary interest is the generalization performance on the MNIST test set. As a feasibility study of a distributed ML framework, we are most interested scaling power while minimizing latency effects during training, but we also want to ensure the correctness of the training algorithm. Since optimization using compiled JS and/or GPUs of the ML JavaScript library possible, but not our focus, we are less concerned with the power performance of a single slave node.

Results for power and latency are shown in Fig. 4. Power increases linearly up to 64 slave nodes, at which point a large increase in latency limits additional power gains for new nodes. This is due to a single server reaching the limit of its capacity to process incoming gradients synchronously. Solutions include using multiple server processes, asynchronous updates, and partial gradient communication. Test error, as a function of the number of nodes is shown in Fig. 5 after 50 iterations (200 seconds) and 100 iterations (400 seconds); i.e. each point represents the same wall-clock computation time. This demonstrates the correctness of MLitB for a given model architecture and learning hyperparameters.

Due to the data allocation policy that limits the data vector capacity of each node to 3000 vectors, experiments with more nodes process more of the training set during the training procedure. For example, using only 1 slave node trains on 3/60 of the full training set. With 20 nodes, the network is training on the full dataset. This policy could easily be modified to include data refreshment when running with unallocated data.

[6] Slave node specifications (32 units): Intel Core i3-2120 3.3GHz (dual-core); 4GB RAM; Windows 7 Enterprise x64; Google Chrome 35. Master node specifications (1 unit): Intel Xeon E5620 2.4GHz (quad-core); 24 GB RAM; Ubuntu 10.04 LTS. NodeJS version: v0.10.28. The NN has a 28×28 input layer connected to 16 convolution filters (with pooling), followed by a fully connected output layer.

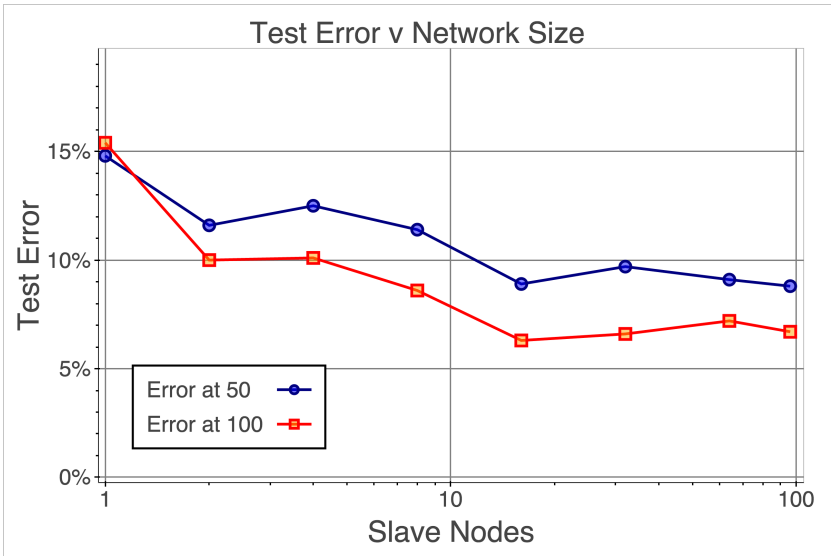


Figure 5. Effects of scaling on optimization. Convergence of the NN is measured in terms of test error after 50 and 100 iterations. Each point represents approximately the same wall-clock time (200/400 seconds for 50 and 100 iterations, respectively).

The primary latency issue is due to all clients simultaneously sending gradients to the server at the end of each iteration. Three simple scaling solutions are 1) increasing the number of master node processes that receive gradients 2) using asynchronous update rules (each slave computes for a random amount of time, then sends updates), reducing the load of any one master node process, and 3) partial communication of gradients (decreasing bandwidth).

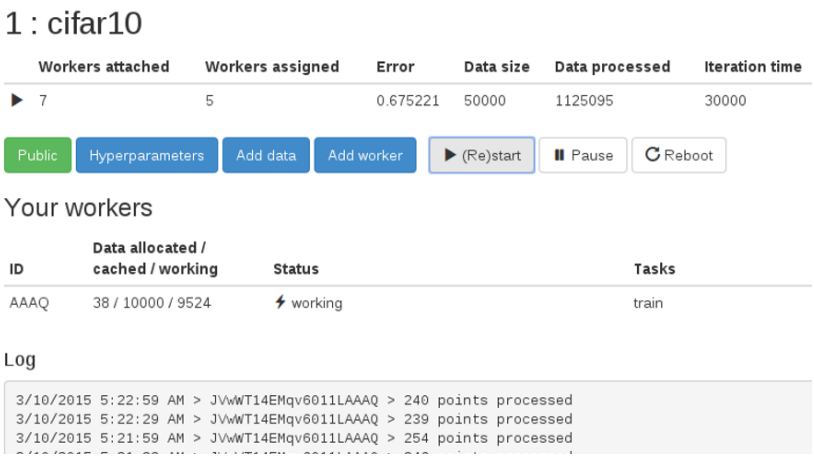


Figure 6. CIFAR-10 project loaded in MLitB.

3.6 Walk-through of MLitB Prototype

We briefly describe how MLitB works from a researcher's point of view.

Specification of Neural Network and Training Parameters

Using a minimalist UI (not shown), the researcher can specify their neural network, for example they can add/remove layers of different types, and adjust regularization parameters (L1/L2/dropout) and learning rates. Alternatively, the researcher can load a previously saved neural network in JSON format (that may or may not have already been trained). Once a NN is specified (or loaded), it appears in the display, along

with other neural networks also managed by the master node. By selecting a specific neural network, the researcher can then add workers and data (e.g. project *cifar10* in Fig. 6).

Specification of Training Data

Image classification data is simple to upload using named directory structures for image labels. For example, for CIFAR10 all files in the "apple" subdirectory will be given label "apple" once loaded (e.g. the image file `/cifar10/apple/apple_apple_s_000022.png`). The entire "cifar10" directory can be zipped and uploaded. MLitB processes JPEG and PNG formats. A test set can be uploaded in *tracker* mode.

Training Mode

In the *training* mode, a training worker performs as many gradient computations as possible within the iteration duration T (i.e. during the *map* step of the main event loop). 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 (Duchi et al., 2011). At the end of the main event loop, new neural network weights are sent via Web Sockets to both trainer workers (for the next gradient computations) and to tracker workers (for computing statistics and executing the latest model).

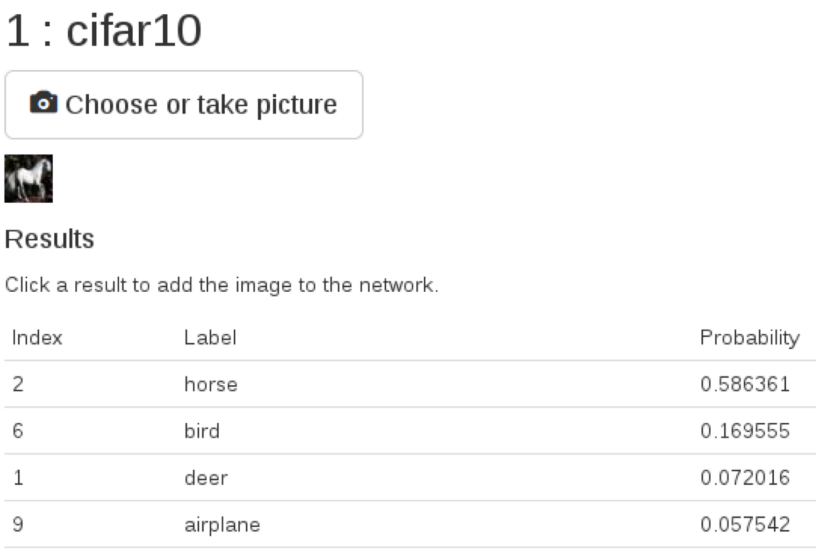


Figure 7. Tracking model (model execution): The label of a test image is predicted using the latest NN parameters. Users can execute a NN prediction using an image stored on their device or using their device's camera. In this example, an image of a horse is correctly predicted with probability 0.687 (the class-conditional predictive probability).

Tracking Mode

There are two possible functions in *tracking* mode: 1) executing the neural network on test data, and 2) monitoring classification error on an independent data set. For 1, users can predict class labels for images taken with a device's camera or locally stored images. Users can also 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. 7 shows a test image being classified by the *cifar10* trained neural network. For 2, users create a statistics worker and can upload test images and track their error over time; after each complete evaluation of the test images, the latest neural network received from the master is used. Fig. 8 shows the error for *cifar10* using a small test set for the first 600 parameter updates.

Archiving Trained Neural Network Model

The prototype does not include a research closure specification. However, it does provide easy archiving functionality. At any moment, users can download the entire model specification and current parameter values in JSON format. Users can then share or initialize a new training session with the JSON object

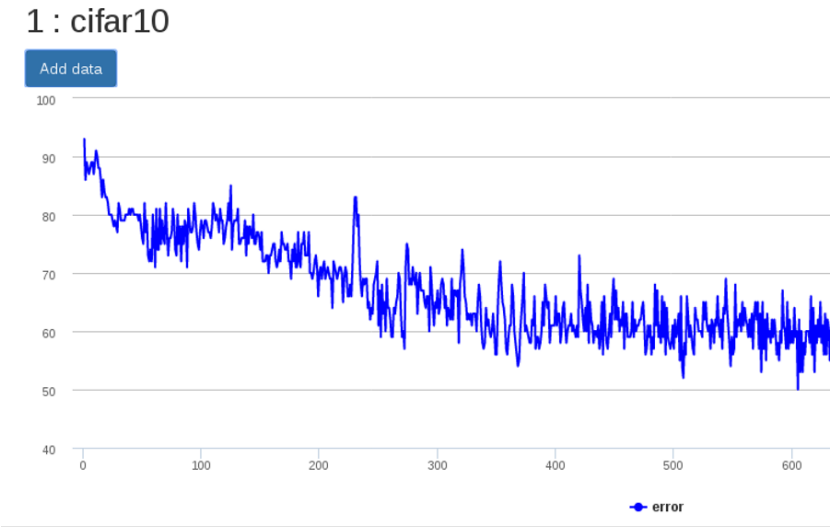


Figure 8. Tracking mode (classification error). A test dataset can be loaded and its classification error rate tracked over iterations; here using a NN trained on CIFAR-10.

by uploading it during the model specification phase, which represents a high-level of reproducibility. Although the JSON object fully specifies the model, it does not include training or testing code. Despite this shortcoming, using a standard protocol is simple way of providing a lightweight archiving system.

3.7 Limitations of MLitB Prototype

In this section we briefly discuss the limitations of the current prototype; later in Section 5 we will discuss the challenges we face in scaling MLitB to a massive level.

Our scaling experiment demonstrates that the MLitB prototype can accommodate up to 64 clients before latency significantly degrades its performance. Latency, however, is primarily affected by the length of an iteration and by size of the neural network. For longer iterations, latency will become a smaller portion of the main event loop. For very large neural networks, latency will increase due to bandwidth pressure.

As discussed previously, the main computational efficiency loss is due to the synchronization requirement of the master event loop. This requirement causes the master server to be idle while the clients are computing and the clients to wait while the master processes all the gradients. As the size of the full gradients can be large (at least $> 1MB$ for small neural networks), the network bandwidth is quickly saturated at the end of a computation iteration and during the parameter broadcast. By changing to an asynchronous model, the master can continuously process gradients and the bandwidth can be maximally utilized. By communicating partial gradients, further efficiency can be attained. We leave this for future work.

There is a theoretical limit of 500MB data storage per client (the viable memory of a web-browser). In our experience, the practical limit is closer to 100MB at which point performance is lost due to memory management issues. We found that 1MB/sec bandwidth was achievable on a local network, which meant that it could handle images on MNIST and CIFAR-10 easily, but would stall for larger images. With respect to Deep Neural Networks, the data processing ability of a single node was limited (especially is one compared to sophisticated GPU enables libraries (Bastien et al., 2012)). Although we were most interested in the scaling performance, we note that naive convolution implementations significantly slow performance. We found that reasonable sized images, up to $100 \times 100 \times 3$ pixels, can be processed on mobile devices in less than a second without convolutions, but can take several seconds with convolutions, limiting its usefulness. In the future, near native or better implementations will be required for the convolutional layers.

4 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.

4.1 Volunteer Computing

BOINC (Anderson, 2004) 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 in computation; this is called *public resource computing*. Public resource or volunteer computing was popularized by SETI@Home (Anderson et al., 2002), 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 (Lane et al., 2013). Hadoop (Shvachko et al., 2010) is an open-source software system for storing very large datasets and executing user application tasks on large networks of computers. MapReduce (Dean and Ghemawat, 2008) is a general solution for performing computation on large datasets using computer clusters.

4.2 JavaScript Applications

In (Cushing et al., 2013) a network of distributed web-browsers called *WeevilScout* is used for complex computation (regular expression matching and binary tree modifications) using a JavaScript engine. It uses similar technology (Web Workers and Web Sockets) as MLitB. ConvNetJS (Karpathy, 2014) is a JavaScript implementation of a convolutional neural-network, developed primarily for educational purposes, which is capable of building diverse neural networks to run in a single web browser and trained using stochastic gradient descent; it can be seen as the non-distributed predecessor of MLitB.

4.3 Distributed Machine Learning

The most performant deep neural network models are trained with sophisticated scientific libraries written for GPUs (Bergstra et al., 2010; Jia et al., 2014; Collobert et al., 2011) that provide orders of magnitude computational speed-ups compared to CPUs. Each implements some form of stochastic gradient descent (SGD) (Bottou, 2010) as the training algorithm. Most implementations are limited to running on the cores of a single machine and by extension the memory limitations of the GPU. Exceptionally, there are distributed deep learning algorithms that use a farm of GPUs (e.g. *Downpour SGD* (Dean et al., 2012)) and farms of commodity servers (e.g. *COTS-HPS* (Coates et al., 2013)). Other distributed ML algorithm research includes the parameter server model (Li et al., 2014), parallelized SGD (Zinkevich et al., 2010), and distributed SGD (Ahn et al., 2014). MLitB could potentially 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.

5 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 first focus on the engineering and research challenges we have discovered during the development of our prototype, along with some we expect as the project grows. Second, we look at the opportunities MLitB provides, not only based on the research directions the challenges uncovered, but also novel application areas that are perfect fits for MLitB. In Section 6 we preview the next concrete steps in MLitB development.

5.1 Challenges

We have identified three keys engineering and research challenges that must be overcome for MLitB to achieve its vision of learning models a global scale.

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 or use smaller neural networks. Smaller NN models have shown promise on image classification performance, in particular the Network in Network (Lin et al., 2013) model from the Caffe model zoo,

is 16MB, and outperforms AlexNet which is 256MB (Jia et al., 2014). It is also possible to first train a deep neural network then use it to train a much smaller, shallow neural network (Ba and Caruana, 2014). Another solution is to distribute the NN (during training and prediction) across clients. An example of this approach is Downpour SGD (Dean et al., 2012).

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, given a fixed bandwidth budget, we want to maximize the information transferred per iteration.

Performance Efficiency

Perhaps the biggest argument against scientific computing with JavaScript is its computation performance. 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 (JavaScript V8, 2014; asm.js, 2014) and GPU kernels are becoming part of existing web engines (e.g. WebCL[7]) and they can be seamlessly incorporated into existing JavaScript libraries, though they have yet to be written for ML.

5.2 Opportunities

Massively Distributed Learning Algorithms

The challenges just presented are obvious areas of future distributed machine learning research (and are currently being developed for the next version of MLitB). Perhaps more interesting is, at a higher level, that the MLitB vision raises novel questions about what it means to train models on a global scale. For instance, what does it mean for a model to be trained across a global internet of heterogeneous and unreliable devices? Is there a single model or a continuum of models that are consistent locally, but different from one region to another? How should a model adapt over long periods of time? These are largely untapped research areas for ML.

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) (Quinn et al., 2011). Projects like these could leverage publicly available, state-of-the-art computer vision models to bootstrap their field research.

Privacy Preserving Computing and Mobile Health

Our MLitB framework provides a natural platform for the development of real privacy-preserving application (Dwork, 2008) 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 (Han et al., 2010), 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.

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 (nrd, 2014).

[7] WebCL by Kronos: `www.khronos.org/webcl`.

6 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.

6.1 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 of a neural network 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 (Szegedy et al., 2014). 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.

6.2 Machine Learning Library

We currently have built a prototype around an existing JavaScript implementation of DNNs (Karpathy, 2014). In the near future we plan on implementing other models (e.g. latent Dirichlet allocation) and algorithms (e.g. distributed MCMC (Ahn et al., 2014)). MLitB is agnostic to learning algorithms and therefore is a great platform for researching novel distributed learning algorithms. To do this, however, MLitB will need to completely separate machine learning model components from the MLitB network. At the moment, the prototype is closely tied to its neural network use-case. Once separated, it will be possible for external modules to be added by the open-source community.

6.3 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 (Bergstra et al., 2010; Bastien et al., 2012) and Caffe (Jia et al., 2014), 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 phones 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.

6.4 Design of 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).

7 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. The MLitB **vision** is based upon the overarching objectives that provide *ubiquitous ML* capability to every computing device, *cheap distributed computing*, and *reproducible research*. The MLitB **prototype** 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. We demonstrated the potential of MLitB on a ML use-case: Deep Neural Networks trained with distributed 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 use-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 and used by other researchers immediately.

REFERENCES

(2014). Data center efficiency assessment. `http://www.nrdc.org/energy/files/data-center-efficiency-assessment-IP.pdf`. Issue Paper IP:14-08-A, August 2014.

Ahn, S., Shahbaba, B., and Welling, M. (2014). Distributed Stochastic Gradient MCMC. *ICML*.

Anderson, D. P. (2004). BOINC: A system for public-resource computing and storage. In *Proceedings of the Workshop on Grid Computing, Nov. 2004*, pages 4–10. IEEE.

Anderson, D. P., Cobb, J., Korpela, E., Lebofsky, M., and Werthimer, D. (2002). SETI@Home: an experiment in public-resource computing. *Communications of the ACM*, 45(11):56–61.

asm.js, M. (2014). `http://asmjs.org`. Accessed: 2014-11-26.

Ba, J. and Caruana, R. (2014). Do deep nets really need to be deep? In *Advances in Neural Information Processing Systems*, pages 2654–2662.

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

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

Begoli, E. and Horey, J. (2012). Design principles for effective knowledge discovery from big data. In *Software Architecture (WICSA) and European Conference on Software Architecture (ECSA), 2012 Joint Working IEEE/IFIP Conference on*, pages 215–218.

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

Bottou, L. (2010). Large-scale machine learning with stochastic gradient descent. In *Proceedings of COMPSTAT'2010*, pages 177–186. Springer.

Boyd, S., Ghosh, A., Prabhakar, B., and Shah, D. (2006). Randomized gossip algorithms. *Information Theory, IEEE Transactions on*, 52(6):2508–2530.

Can I Use (2014). Javascript API support comparison. `http://caniuse.com/#cats=JS_API`. Accessed: 2014-11-26.

Casadevall, A. and Fang, F. C. (2010). Reproducible science. *Infection and immunity*, 78(12):4972–4975.

Coates, A., Huval, B., Wang, T., Wu, D., Catanzaro, B., and Andrew, N. (2013). Deep learning with cots hpc systems. In *Proceedings of The 30th International Conference on Machine Learning*, pages 1337–1345.

Collobert, R., Kavukcuoglu, K., and Farabet, C. (2011). Torch7: A matlab-like environment for machine learning. In *BigLearn, NIPS Workshop*, number EPFL-CONF-192376.

Cushing, R., Putra, G. H. H., Koulouzis, S., Belloum, A., Bubak, M., and De Laat, C. (2013). Distributed computing on an ensemble of browsers. *Internet Computing, IEEE*, 17(5):54–61.

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

Dean, J. and Ghemawat, S. (2008). MapReduce: simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113.

Duchi, J., Hazan, E., and Singer, Y. (2011). Adaptive subgradient methods for online learning and stochastic optimization. *The Journal of Machine Learning Research*, 12:2121–2159.

Dwork, C. (2008). Differential privacy: A survey of results. In *Theory and Applications of Models of Computation*, pages 1–19. Springer.

Han, S., Ng, W. K., Wan, L., and Lee, V. (2010). Privacy-preserving gradient-descent methods. *Knowledge and Data Engineering, IEEE Transactions on*, 22(6):884–899.

Hinton, G., Deng, L., Yu, D., Dahl, G. E., Mohamed, A.-r., Jaitly, N., Senior, A., Vanhoucke, V., Nguyen, P., Sainath, T. N., et al. (2012). Deep neural networks for acoustic modeling in speech recognition: The shared views of four research groups. *Signal Processing Magazine, IEEE*, 29(6):82–97.

IETF (2011). The websocket protocol. http://tools.ietf.org/html/rfc6455. Accessed: 2014-12-04.

JavaScript V8, C. (2014). https://developers.google.com/v8. Accessed: 2014-11-26.

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

Karpathy, A. (2014). ConvNetJS Deep Learning in the browser. http://www.convnetjs.com. Accessed: 2014-07-09.

Khan, F., Foley-Bourgon, V., Kathrotia, S., Lavoie, E., and Hendren, L. (2014). Using javascript and webcl for numerical computations: A comparative study of native and web technologies. In *Proceedings of the 10th ACM Symposium on Dynamic Languages*, pages 91–102. ACM.

Krizhevsky, A., Sutskever, I., and Hinton, G. E. (2012). Imagenet classification with deep convolutional neural networks. In *Advances in neural information processing systems*, pages 1097–1105.

Lane, T. J., Shukla, D., Beauchamp, K. A., and Pande, V. S. (2013). To milliseconds and beyond: challenges in the simulation of protein folding. *Current opinion in structural biology*, 23(1):58–65.

Li, M., Andersen, D. G., Park, J. W., Smola, A. J., Ahmed, A., Josifovski, V., Long, J., Shekita, E. J., and Su, B.-Y. (2014). Scaling distributed machine learning with the parameter server. In *Proc. OSDI*, pages 583–598.

Lin, M., Chen, Q., and Yan, S. (2013). Network in network. *arXiv preprint arXiv:1312.4400*.

Liu, S., Yang, N., Li, M., and Zhou, M. (2014). A recursive recurrent neural network for statistical machine translation. In *Proceedings of ACL*, pages 1491–1500.

McNutt, M. (2014). Reproducibility. *Science*, 343(6168).

Quinn, J. A., Leyton-Brown, K., and Mwebaze, E. (2011). Modeling and monitoring crop disease in developing countries. In *AAAI*.

Ray, B., Posnett, D., Filkov, V., and Devanbu, P. (2014). A large scale study of programming languages and code quality in github. In *Proceedings of the 22Nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE 2014, pages 155–165. ACM.

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

Stodden, V., Borwein, J., and Bailey, D. H. (2013a). "setting the default to reproducible" in computational science research. *Computational Science Research*, 46(5).

Stodden, V., Guo, P., and Ma, Z. (2013b). Toward reproducible computational research: an empirical analysis of data and code policy adoption by journals. *PloS one*, 8(6):e67111.

Sutskever, I., Vinyals, O., and Le, Q. V. (2014). Sequence to sequence learning with neural networks. *arXiv preprint arXiv:1409.3215*.

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

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

W3C (2014). Web workers, editor's draft 19 may 2014. http://dev.w3.org/html5/workers. Accessed: 2014-12-04.

W3Techs (2014). Usage of JavaScript for websites. http://w3techs.com/technologies/details/cp-javascript/all/all. Accessed: 2014-06-17.

Zinkevich, M., Weimer, M., Li, L., and Smola, A. J. (2010). Parallelized Stochastic Gradient Descent. In *Advances in Neural Information Processing Systems*, pages 2595–2603.

Figure 1(on next page)

Overview of MLitB.

{\bf (1)} A researcher sets up a learning problem in his/her browser. {\bf (2)} Through the internet, grid and desktop machines contribute computation to solve the problem. {\bf (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.
Icon made by Freepik from www.flaticon.com

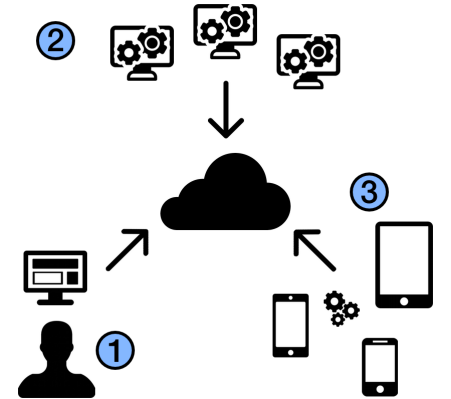

2
3
1

Figure 2(on next page)

MLitB architecture and technologies.

{\bf (1)} Servers are {\em Node.js} applications. The {\em master server} is the main server controlling communication between clients and hosts ML projects. {\bf (2)} Communication between the master server and clients occurs over Web Sockets. {\bf (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.~\ref{fig:mlitb_workers} for details. {\bf (4)} A special data worker on the client communicates with the data server using XHR. {\bf (5)} The {\em data server}, also a Node.js application, manages uploading of data in {\em zip} format and serves data vectors to the client data workers. Icon made by Freepik from www.flaticon.com

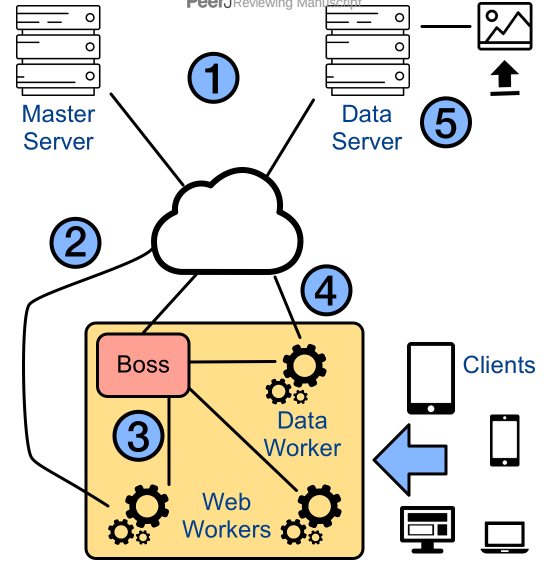

Master Server
Data Server
1
2
3
4
5
Boss
Data Worker
Web Workers
Clients

Figure 3(on next page)

MLitB Client Workers.

Each client connection to the master server initiates a {\em UI worker}, aka a {\em boss}. For uploading data from a client to the data server and for downloading data from the data server to a client, a separate Web Worker called the {\em data worker} is used. Users can add slaves through the UI worker; each slave performs a separate task using a Web Worker. Icon made by Freepik from www.flaticon.com

UI worker
(boss)

Data
Worker

Slave
Workers

Trainer
Worker

Statistics
Worker

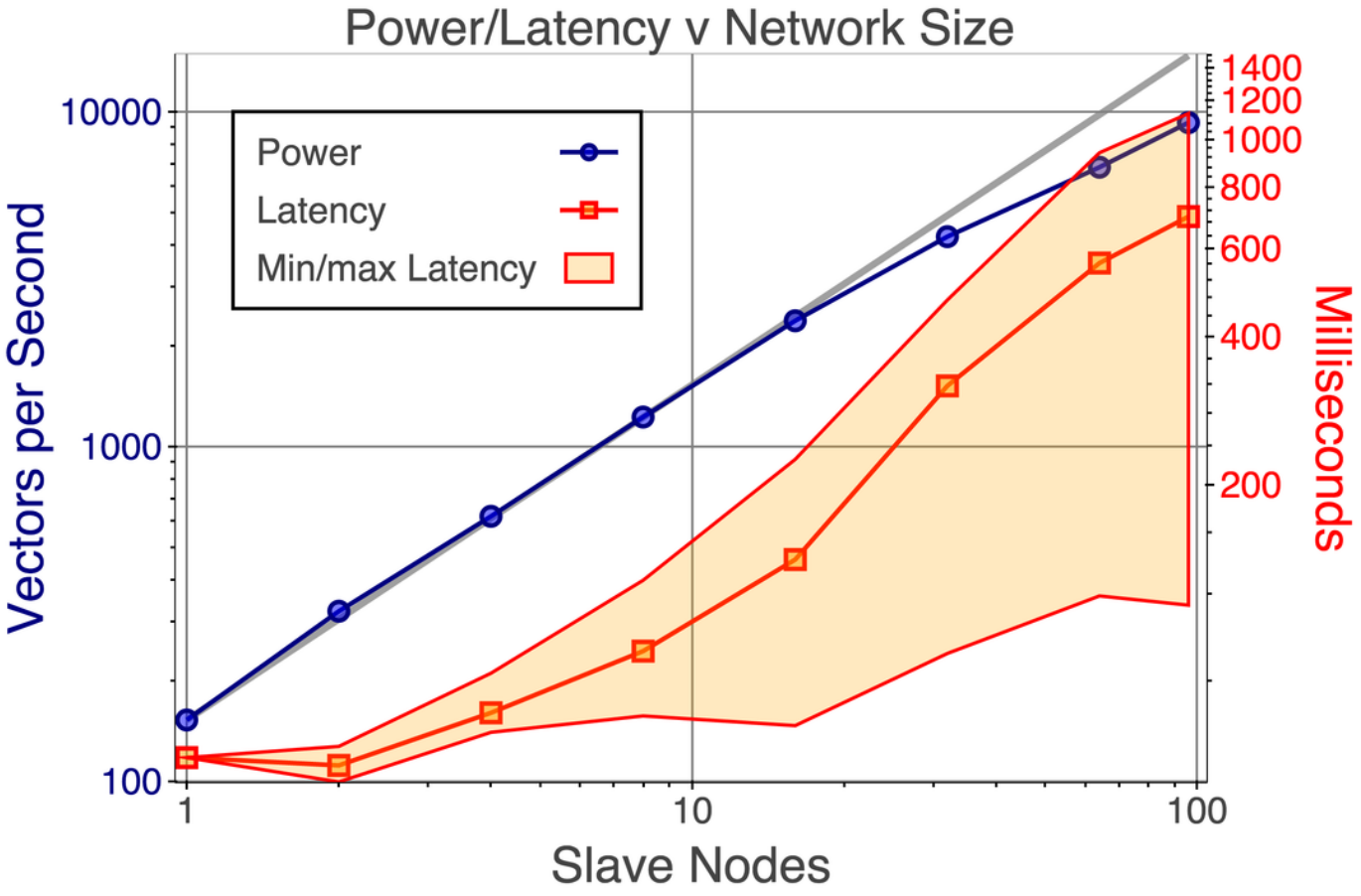
Take Picture
Worker

Download
Worker

4

Effects of scaling on power and latency.

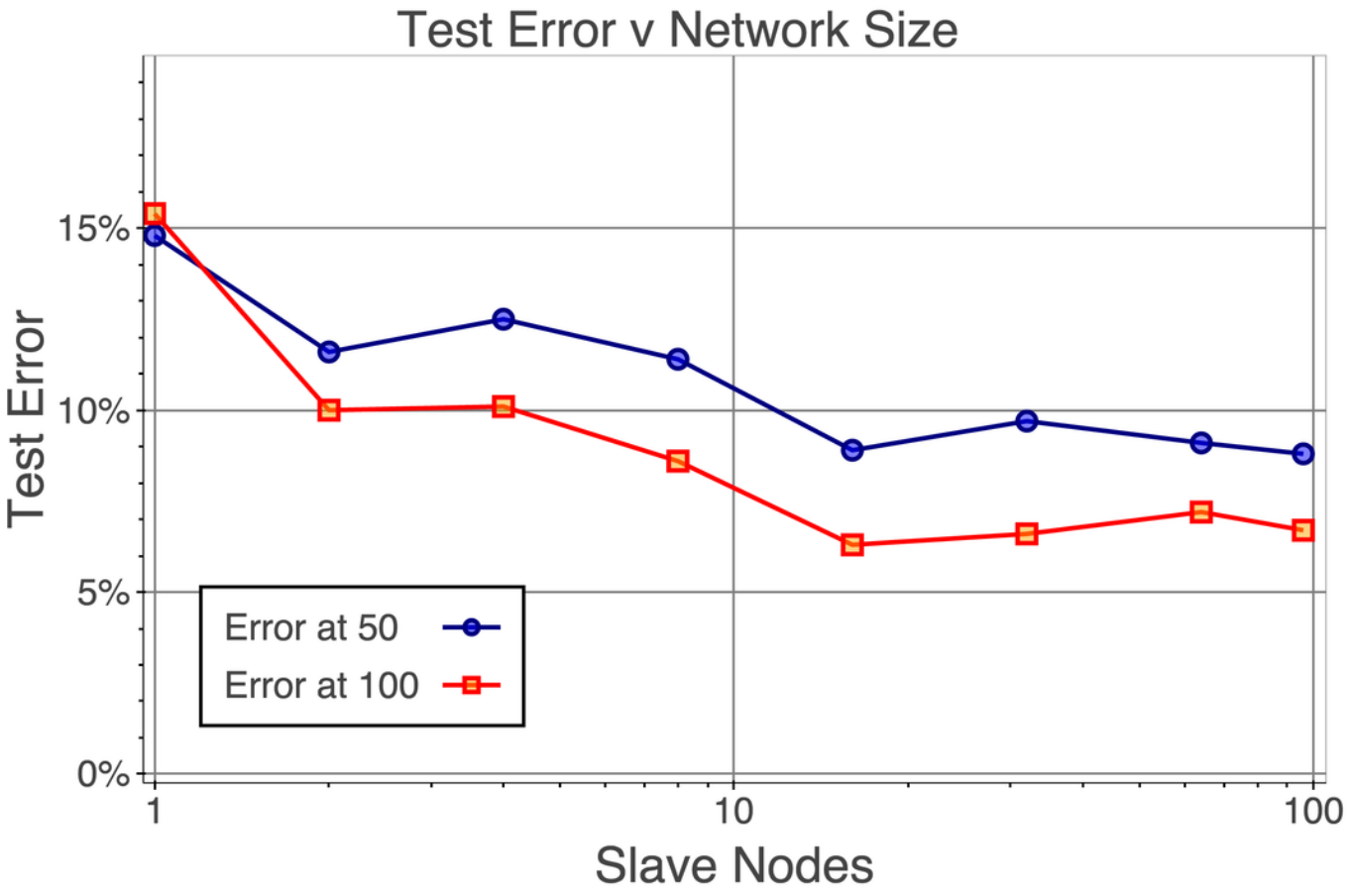
Power---measured as the number of data vectors processed per second---scales linearly until 64 nodes, when the increase in latency jumps. The ideal linear scaling is shown in grey.



5

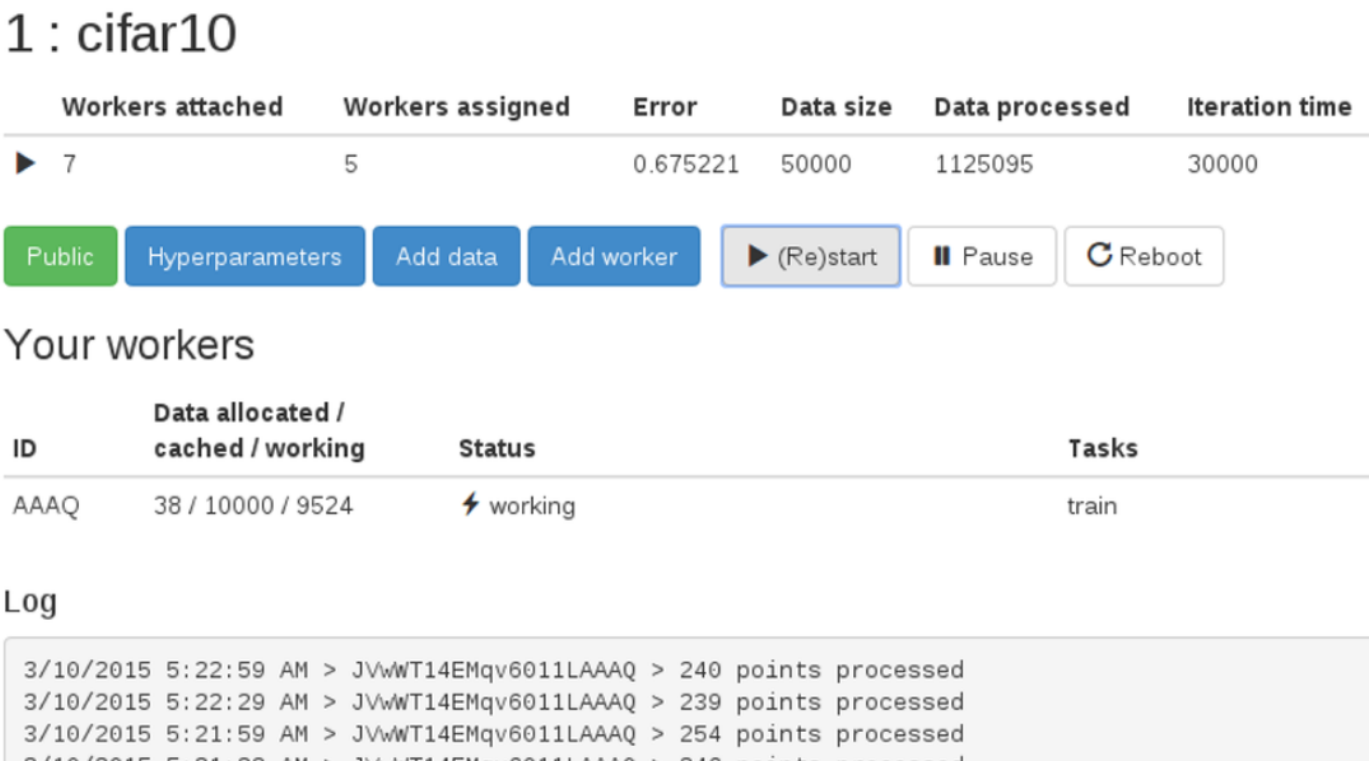
Effects of scaling on optimization.

Convergence of the NN is measured in terms of test error after 50 and 100 iterations. Each point represents approximately the same wall-clock time (200/400 seconds for 50 and 100 iterations, respectively).



6

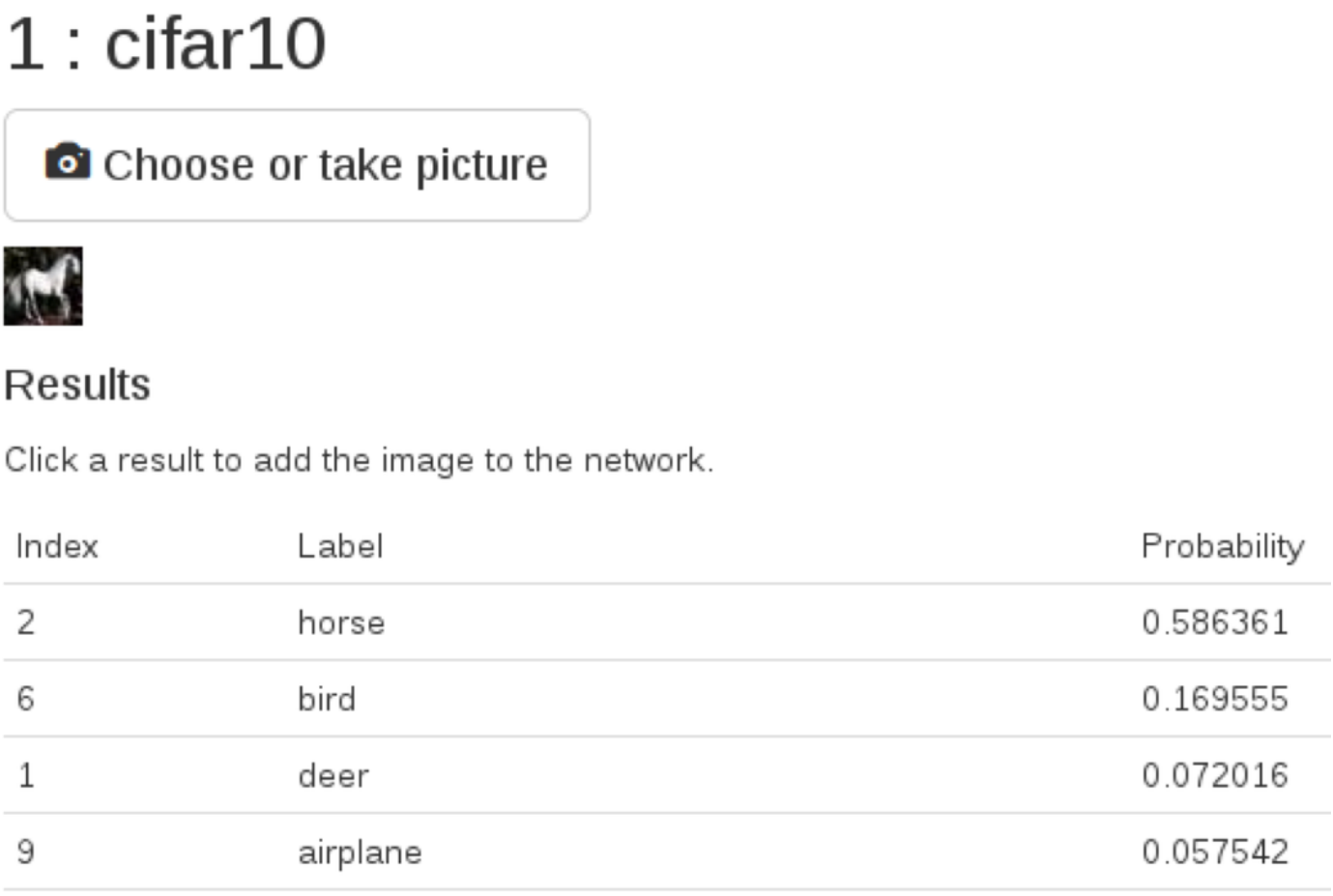
CIFAR-10 project loaded in MLitB.



7

Tracking model (model execution).

The label of a test image is predicted using the latest NN parameters. Users can execute a NN prediction using an image stored on their device or using their device's camera. In this example, an image of a horse is correctly predicted with probability 0.687 (the class-conditional predictive probability).



8

Tracking mode (classification error).

A test dataset can be loaded and its classification error rate tracked over iterations; here using a NN trained on CIFAR-10.