Chapter 1

WebNN Report

Authors: Aaron Goin, Ron Cotton

Abstract: Machine learning has come to the browser in TensorFlow.js, but as yet there is no existing solution for serving and training deep neural networks over the internet. WebNN seeks to make it easy to configure, distribute, and train neural networks asynchronously over the internet. In this paper we explain the WebNN approach to distributed training, describe it's architecture, and test several peer-based weight merging algorithms against each other. Our results show that a peer-based weighted average merging algorithm performs better than naive averaging, and show that WebNN is a good solution for distributed training of models using pre-labeled batches, or client-provided data.

1.1 Introduction

There are a myriad of tools and frameworks available for training neural networks, but Google's TensorFlow is rapidly becoming the leader in the DNN space. Recently, Google added TensorFlow.js to it's framework–bringing machine learning to the browser. TensorFlow.js is inspired by TensorFlow's Keras in it's API, and it leverages WebGL to accelerate execution of neural networks on the client's GPU. Should the client's browser not support WebGL, TensorFlow.js will fallback to executing the model on the CPU.

While TensorFlow.js does run in the browser, it's a client-side technology that requires additional services on the back-end to distribute trained models to clients, or to train a central model that all clients can execute, train, and share.

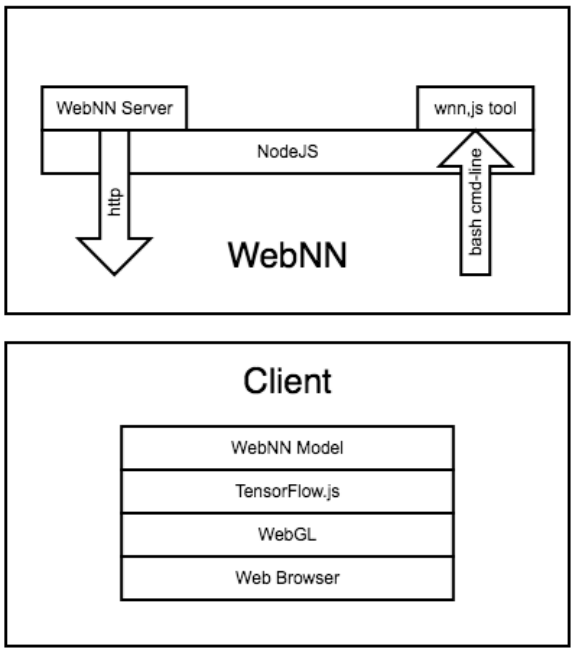
To this end, weve created a system called WebNN to facilitate distributing and training models in the web browser. WebNN also allows users to employ their models in real web applications, and have their users train the model with their own private data, eliminating the need for creating large, pre-labeled training data.

Because we wanted WebNN to applicable in web applications along with general research, we have to distribute the full model to each client, or else no single client could derive utility from the model. WebNN uses a single master node (the Server) to manage clients.

WebNN source resides at `https://github.com/aarongoin/WebNN`.

1.2 WebNN Architecture

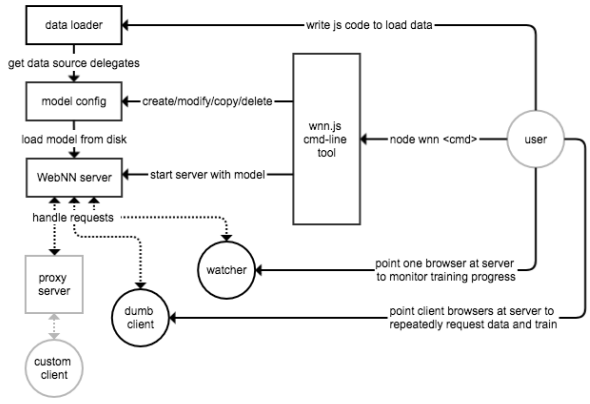
To make it easy to interface with the front-end, we employ JavaScript (ES6+) through NodeJS. As seen in Figure 2.2.1, the WebNN Server builds off of NodeJS to serve the model over http. Also supported by NodeJS is a small tool "wnn.js" which provides some convenient command-line controls for WebNN.



[2.2.1] Solutions Stack

1.3 WebNN Configuration

WebNN can be employed as a standalone server, or applied to an existing application as a service. Figure 2.2.2 gives a high level overview of how WebNN is used.



[2.2.2] Usage Overview

Users create their models in a JSON format, configure it training and validation properties, and create a JavaScript module which the server will use to get training and validation data. The server will hand the model off to clients for training, along with a set of weights, and training data upon request. Clients can send back their modified weights to the server, and receive a new set of weights (from another client) to merge into their own.

Users can employ the watcher to watch the model validation and training metrics in a live dashboard. If the user has pre-labeled data, they can employ the provided dumb client to train their model. Both the watcher and the dumb client are created when starting the server.

WebNN can be configured to validate the model every N seconds, where N is defined within the model "config.js" file. When validation is triggered, the next client to check in is told to validate. They request validation data (and receive a mini-batch from the validation data delegate the user defined), validate their current weights, and then return the accuracy and loss to the server. This validation is stochastic, and is represents validation of a single sample in the pool of clients. As a result, validation results are more valuable when taken as a moving average.

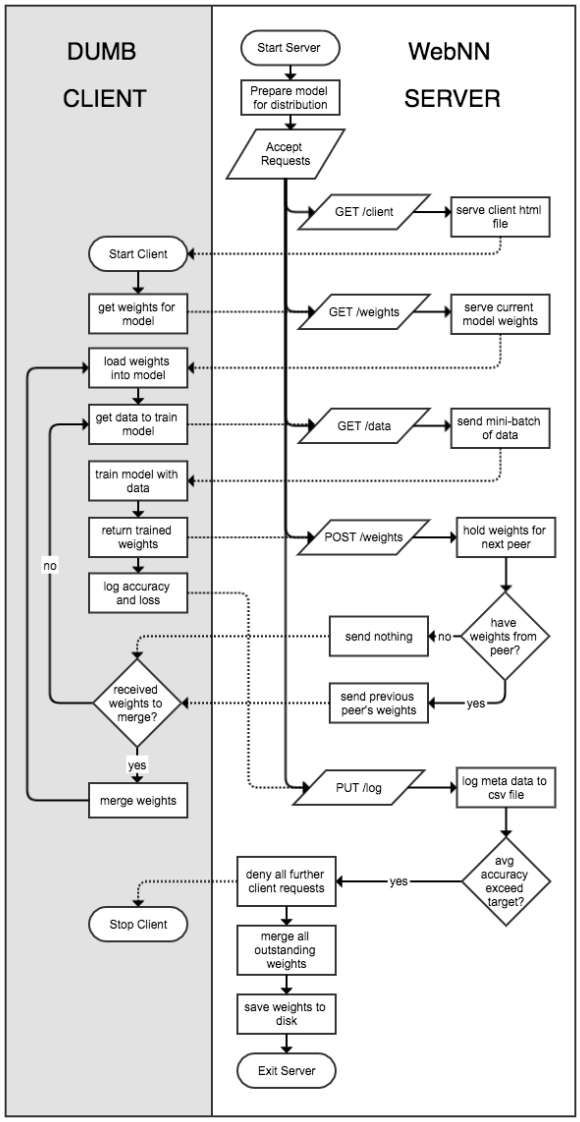
WebNN supports a centralized form of automatic weight decay, though this is optional. The user provides a starting learning rate, r. Then, given the moving average accuracy, A, loss, L, and number of validations, t, the server updates the learning rate as follows:

$$r_{t+1} = r_t^2(1 - A^{(L*t)}) \tag{1.1}$$

This has the effect of naturally scaling the learning rate down as the model gets more accurate. Note that if the outcome of the equation results in a higher learning rate, the learning rate remains the lower value.

1.4 WebNN System

Upon starting the WebNN server with a given model, it will validate the model and prepare it for distribution. The server then creates the dumb client and the watcher client, before waiting for requests. Figure 2.4.1 shows the WebNN training process using the dumb client in a flowchart.



[2.4.1] Server w/ Dumb Client

1.5 Merging Weights

Our first approach was to have the master node merge client's checked-in weights with a central set of weights. The inherently asynchronous nature of communication over the internet along with the variance of compute power each client might possess presents challenges for training centralized weights, as an incoming set of weights could be several iterations behind the central set, and a naive merging could revert prior training.

Distributed TensorFlow handles these so-called stale weights by simply dropping them, and not merging them at all. This is an acceptable strategy when training over a central dataset, but this is unacceptable when clients are potentially training with their own data that a central model cannot redistribute and train on. Dropping stale weights in WebNN could result in a true loss of learning, and so we turned first to a method proposed by [1] to scale stale weights based on how stale they were.

Our initial architecture relied on the server to merge all weights, forcing clients to wait until their weights had been merged into the central set before they could continue. This architecture worked well for our initial tests, but we quickly discovered merging weights on the server creates a bottleneck and prevents the server from supporting web-scale applications.

To enable web-scale applications, we offload merging to our clients, and the server simply facilitates swapping weights between peers. The server still retains a copy of all outstanding weights (weights that have been sent to a client who has not yet updated the server), so weights are never in jeopardy of being lost. And the server can be made to merge all weights together before sending them to be validated, should the user wish to do so.

Swapping weights is done simply: when client A posts their weights, the server sends A a set of weights from client B for A to merge into their own. The weights A sent are held by the server to pass to the next client to post weights, perhaps client B, or C, etc.

We implemented three different methods a client could use to merge the incoming weights with their own. The first is to simply average the two sets together, the second is to perform a weighted average based on the number of training iterations each set had undergone, and the last which we call mimic merge, is a sort of weighted average with an additional correction based on the difference in accuracy for each set of weights.

1.5.1 Average Merge

This is the naive approach to merging weights. Weights are simply averaged together regardless of potential staleness. The upside here is that this is the fastest merging method.

$$W_n = (W_{n-1} + W_p)/2. \tag{1.2}$$

1.5.2 Weighted Merge

The weighted merge is also an average, but it takes potential staleness into account. Every time a client trains, it's time step variable is incremented. This step variable is always kept with the weights, and weighted merge scales all weights by their time step to favor less stale weights.

$$W_n = (W_{n-1} * t_{n-1} + W_p * t_p)/(t_{n-1} + t_p) \quad (1.3)$$

1.5.3 Mimic Merge

The mimic merge uses the same information as the weighted merge, but handles it differently. Any average is going to be bounded between the values themselves. So rather than averaging the weights, the mimic merge determines which set of weights is less stale, determines the difference between those weights and the other set, and then moves the current weights in the same direction. To avoid erratic oscillations, the weight adjustment is scaled according to how far apart the two weights are.

$$d = t_{n-1} - t_p \quad (1.4)$$

if d > 0:

$$W_D = W_p - W_{n-1} \quad (1.5)$$

else:

$$W_D = W_{n-1} - W_p \quad (1.6)$$

$$W_n = (W_{n-1} + W_D)/(|d| + 1). \quad (1.7)$$

1.6 Experiment Setup

For our tests we use the MNIST data-set, and employ the convolutional architecture as described at `https://js.tensorflow.org/tutorials/mnist.html`.

We tested our merge functions by training on the model until the 5-second running average of the validation accuracy was greater than or equal to 98%. We set our model to validate once every second, set our mini-batch size to 64 samples, and set our learning rate to 0.2.

To compare between merge functions, we trained our model with each merge function and 4 clients. We did this 4 times for each merge function.

We also trained this model 4 times with 1 client, 2 clients, 4 clients, and 8 clients, to observe scaling behavior, and look for a speed-up in training–if any.

Note that WebNN does not merge weights when only a single client is training, as such a step is unnecessary, and so we use our averaged sample data from training with 1 client as the baseline to judge all other data.

1.6.1 Test Hardware

Both the server and the clients are run on Mid 2011 iMacs with the following specs:

CPU: 2.5 GHz Intel Core i5

RAM: 8 GB 1333 MHz DDR3

GPU: AMD Radeon HD 6750M 512 MB

All test devices were on a shared local network, so network latency is fairly low.

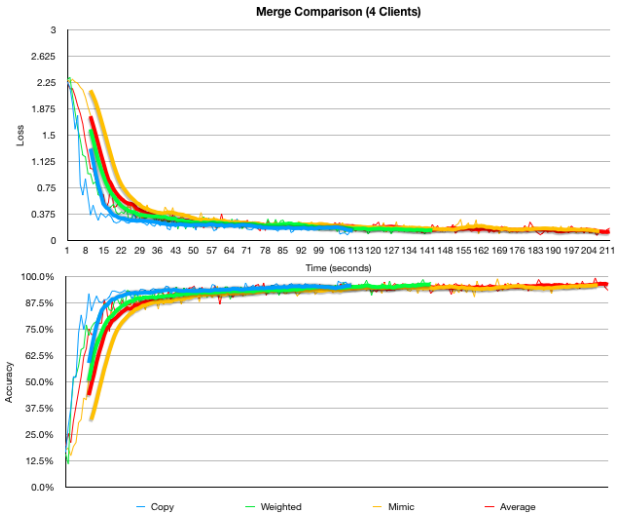
1.7 Experiment Results

We averaged the loss and accuracy over the 4 runs for each, and then plotted the 10-second running averages of each curve for easy comparison. More samples would be better to iron out variance, but even with 4 runs trends are readily apparent. Because we are averaging: the plotted lines end with the shortest iteration of that set. We compare average time to train for the average and weighted average merge functions in Section 2.7.2.

We first give results for comparing the merge functions, and then give results of scaling the number of clients.

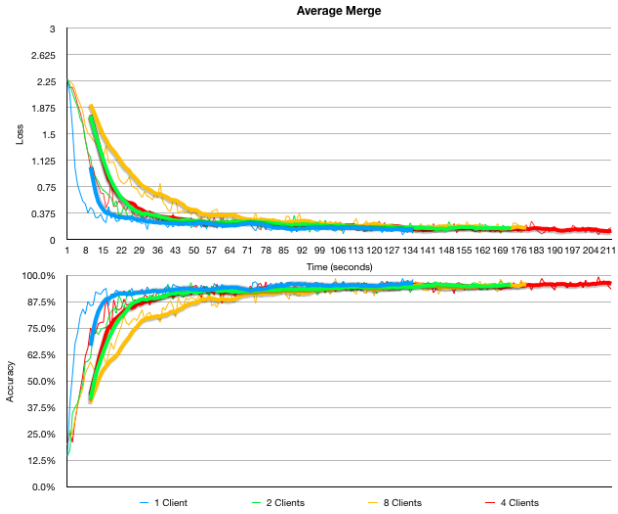
1.7.1 Merge Comparison

Our results show that the weighted merge results in more accurate training in less time than the naive merge. The mimic merge is the worst performer in both speed and duration of training. The Copy merge function is effectively equivalent to training with a single client and is used as a benchmark. This merging method has the client pick whichever set is most accurate, and discards the other. When used in an environment where clients provide their own data: clients using copy merge would discard weights and lose valuable trained weights.

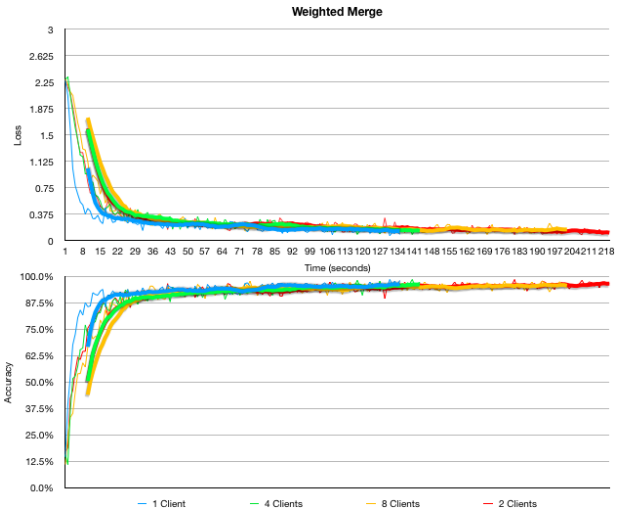


1.7.2 Training Scaling

We use one client training as the baseline, which appears faster than every method. This should be taken with a grain of salt, as the server is optimized to skip merging weights when it does not need to. Note that we did not test mimic merge on 8 clients due to it's poor performace.



While the average merge and weighted average merge appear similar with 4 clients, the difference emerges more clearly when training with 8 clients. Indiscriminately averaging the weights results in slowed training when the number of clients increases. As stated by Zhang et. al.: given N clients training, the average weights will be stale by N/2 iterations. [1] Thus staleness increases along with clients, and stale weights negatively impact training.



The negative impact of staleness all but disappears when merging with the weighted average method as training speed is relatively unaffected by an increased numbers of clients.

To better compare the average time to train, we calculated the average and standard deviation for runs from each sample.

Average Merge - Train Time (Seconds)

	2 Clients	4 Clients	8 Clients
Average	227.92	244.2	260.04
STD DEV	38.06	39.16	114.18

Weighted Merge - Train Time (Seconds)

	2 Clients	4 Clients	8 Clients
Average	253.88	178.86	267.3
STD DEV	24.2	42.24	62.48

We can see that as the number of clients increases, the variance of training time does increase, but the increase is much more drastic when using the naive average merge. No clear pattern of decreasing training time is apparent from the data. 4 Clients using weighted merging seemed to do particularly well, but this may disappear with a larger sample size.

1.8 Conclusion

WebNN is a viable platform for distributing and training a centralized neural network in the browser. From our data we seen that a peer-based weight merge system works best with a weighted average favoring weights with more training iterations behind them.

More extensive testing needs to be done on benchmarking how the system scales with clients, as our testing indicates increasing training times with increasing clients. The peer-based merging we implemented can still be improved to promote less variance between clients, and hopefully further decrease training time. One promising method called online distillation has been proposed by researchers from Google Brain.

While our testing has used mini-batches, WebNN works just fine with single-sample SGD though training is naturally slower. WebNN helps open the doors for smart web applications that can learn and adapt to users, while maintaining user data privacy.

Bibliography

[1] J. L. X. L. Suyog Gupta Wei Zhang, "Staleness-aware async-sgd for distributed deep learning," *Proceedings of the Twenty-Fifth International Joint Conference on Artificial Intelligence*, vol. 25, pp. 2350–2356, 2016.