A HIGH PERFORMANCE NEURAL NETWORK JAVASCRIPT LIBRARY

A Project

Presented to the faculty of the Department of Computer Science

University of Alaska, Fairbanks

Submitted in partial satisfaction of
the requirements for the degree of

MASTER OF SCIENCE

in

Computer Science

by

Travis Michael Payton, B.A., B.S.

SPRING
2015

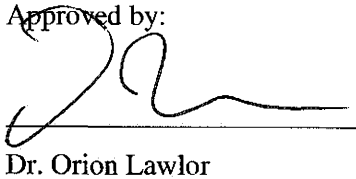
A HIGH PERFORMANCE NEURAL NETWORK JAVASCRIPT LIBRARY

A Project

by

Travis Michael Payton

Approved by:



_____, Committee Chair

Dr. Orion Lawlor

2015-04-21

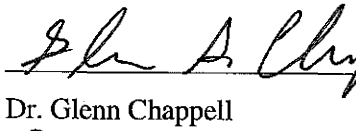
Date

_____, Committee Member

Dr. Jon Genetti

4/21/15

Date



_____, Committee Member

Dr. Glenn Chappell

2015-04-21

Date

ABSTRACT

of

A HIGH PERFORMANCE NEURAL NETWORK JAVASCRIPT LIBRARY

By

Travis Michael Payton

This report covers Intellect.js, a new high-performance Artificial Neural Network (ANN) library written in JavaScript and intended for use within a web browser. The library is designed to be easy to use, whilst remaining highly customizable and flexible.

A brief history of JavaScript and ANNs is presented, along with decisions made while developing Intellect.js. Lastly, performance benchmarks are provided, including comparisons with existing ANN libraries written in JavaScript.

Appendices include a code listing, usage examples, and complete performance data.

Intellect.js is available on GitHub under the MIT License.
https://github.com/sutekidayo/intellect.js

ACKNOWLEDGEMENTS

I would like to thank Dr. Jon Genetti for introducing me to the concept of Neural Networks and their application as an evolutionary game AI. The experience of creating an intelligence that learned to beat me at checkers was eye opening and powerful.

I would also like to thank Dr. Glenn Chappell for teaching me the fundamentals of Computer Science & OpenGL. Without which I would have never made it to where I am today.

I would also like to thank Dr. Lawlor for teaching me all about computer architecture, optimizations, and advanced computer graphics. His constant example of creativity and exploration was the inspiration for this project.

Lastly, I would like to dedicate this project to my wife and son.

TABLE OF CONTENTS

Title Page i
Abstract iii
Acknowledgements iv
Table of Contents v
List of Figures vii
1. INTRODUCTION 1
1.1 PROJECT BACKGROUND 1
1.1.1 JavaScript and Thin-Servers 1
1.1.2 Artificial Neural Networks 2
1.1.3 Previous Work 5
1.1.4 Limitations of Previous Work 5
1.2 Introducing Intellect.js 5
2. IMPLEMENTING INTELLECT.JS 6
2.1 JavaScript Best Practices 6
2.2 Performance Minded 7
2.2.1 Arrays 8
2.2.2 Activation Functions 10
2.3 Library Design 11
2.3.1 Initialization 11
2.3.2 Activation Functions 14
2.3.3 Error Functions 15
2.3.4 Computing Outputs 16
2.3.5 Training Methods 18
2.3.5.1 BackPropagate 18
2.3.5.2 Genetic 23
2.3.6 Creating New Workers 25
2.3.7 Saving and Loading Network Configuraitons 28
3. RESULTS 28
3.1 Working Examples 29
3.1.1 Computing XOR 29
3.1.2 Character Recognition 30

3.1.3 Genetic Algorithm XOR 32
3.2 Network Performance 33
3.2.1 Comparision with Native C++ 33
3.2.2 Comparison with Other JavaScript Libraries 34
4. CONCLUSION AND FUTURE WORK 37
Bibliography 38
APPENDIX A – CODE LISTING 40
Intellect.js 40
Examples / Demo index.html 52
C++ Neural Network Library NN2.h 62
C++ to JS Port of NN2.h 69
APPENDIX B – BENCHMARKS 73
Array Creation and Filling 73
1D vs 2D Array Access 74
Typed Array vs Normal Array 75
Array Comparison for Network Evaluation 76
C++ Benchmark 80
JavaScript vs C++ Performance 82
ANN Library Comparision – Initialization 83
ANN Library Comparision – Evaluation 85
ANN Library Comparison – Creation and XOR Training 87

LIST OF FIGURES

Figure 1 - Rosenblatt's perceptron 2
Figure 2 - Perceptron Algorithm 3
Figure 3 - Diagram of a Multilayer Perceptron or Feed Forward Network 3
Figure 4 - Sigmoid Activation Function 3
Figure 5 - Graph of Sigmoid Function 4
Figure 6 - JavaScript OOP: Declaring a NameSpace 6
Figure 7 - Classes in JavaScript 6
Figure 8 - Implicit variable declaration and scope leak 6
Figure 9 - JavaScript Prototype 7
Figure 10 - Benchmark Computer & Browser Configuration 7
Figure 11 - Array Allocation and Filling Performance 8
Figure 12 - 1D vs 2D Array Access 9
Figure 13 - Typed Array vs Normal Array 9
Figure 14 - Array comparison for Network Evaluation 9
Figure 15 - Activation Function Benchmarks 10
Figure 16 - Graph of Standard Activation functions 10
Figure 17 - Intellect Constructor Options 12
Figure 18 - Comparison of Box-Muller transform and optimized generator 13
Figure 19 - Optimized Normal Distribution Generator and Initialization Helper Functions 14
Figure 20 - ActivationFunctions and ActivationFunctionDerivatives Objects 14
Figure 21 – Converting a String Function Name to an Executable Function 15
Figure 22 - Labeled Network Topography 16
Figure 23 - ComputeOutputs Method 17
Figure 24 - Converting Network Evaluation to Standalone function 18
Figure 25 - Train method 18
Figure 26 - BackPropagate Function 22
Figure 27 - Fisher-Yates Shuffle Algorithm 22
Figure 28 - CalculateDeltas Function 23
Figure 29 - UpdateWeights Function 23
Figure 30 - Thomas Hunt Morgan's illustration of crossing over (1916) 24
Figure 31 - Web Worker Functions 26
Figure 32 - Passing Messages to Workers 26
Figure 33 - Worker's Message Processing Function 27
Figure 34 - Main Thread's Message Processing Function 28
Figure 35 - saveNetwork Function 28
Figure 36 - loadNetwork Function 28
Figure 37 - Learning XOR Backpropagate Example 30
Figure 38 - Sample MNIST Images 30
Figure 39 - Learning MNIST Backpropagate Example 31
Figure 40 - Accuracy of Network on MNIST Testing Set 31
Figure 41 - Genetic Algorithm XOR Example 32
Figure 42 - JavaScript versus compiled C++ performance 34
Figure 43 - JS ANN Library Comparison - Initialization 34
Figure 44 - JS ANN Comparison - Compute Output 35
Figure 45 - JS ANN Comparison - Learning XOR 35

1. INTRODUCTION

JavaScript has been an invaluable scripting language for web developers since its advent in 1995. In the last twenty years, it has grown from a simple scripting language for webpages into a powerful and robust language. Today it is the driving force behind many modern applications, databases, platforms, and frameworks.

Another area that has seen major strides in research and application in the last two decades has been Artificial Neural Networks (ANN). Their ability to learn and find optimal solutions to very complex problems makes them a popular field today.

This project marries these two fields together by creating a JavaScript Library for utilizing ANNs. The library, Intellect.js, was developed to be fast, expandable, and easy to use, in order to make ANNs accessible across a variety of platforms.

1.1 PROJECT BACKGROUND

In order to understand the significance of this project and reasoning behind it, a general history of JavaScript and Artificial Neural Networks is invaluable. A brief history of the two fields is presented in the following subsections.

1.1.1 JAVASCRIPT AND THIN-SERVERS

JavaScript was created by Netscape's Brendan Eich in 1995 and originally called Mocha. It was renamed to LiveScript, and then subsequently to JavaScript shortly thereafter because Java was extremely popular at the time. However, the similarities between Java and JavaScript end there. Java is a statically typed language, meaning a variable can only hold a single, declared type; JavaScript is a dynamically typed language meaning types are inferred from value, and variable names can be re-used for different types. Java is an Object Oriented Programing (OOP) language with classes; JavaScript is a prototype-based language with first-class functions making it a multi-paradigm language supporting OOP, imperative, and functional programming. Java is compiled to Java bytecode that is run on a Java Virtual Machine (JVM); JavaScript is compiled to machine code at run time using highly optimized Just in Time (JIT) compilers. Beyond these differences, JavaScript also has several features that are not available in Java: JavaScript supports closures; all functions are variadic; prototypes can be redefined at run time; object methods can be redefined independently of their prototype; and JavaScript supports anonymous functions.

From 1996 to 2000, ECMA International carved out a standard for JavaScript called ECMAScript that strove to make JavaScript behave the same in all web browsers. In 2005, Eich joined ECMA International and began working on the ECMAScript 4 standard. In 2007, technology giants Microsoft, Yahoo, and Google, teamed up to improve the outdated ECMAScript 3 and released version 3.1, which included many of the improvements from ECMAScript 4. In December of 2009 ECMAScript 5 was released and was quickly adopted by the major browsers.

ECMAScript is still being developed, with new features and capabilities being added every few years. The sixth edition will be released June 2015, and version 7 is currently in active development. With each edition, ECMAScript becomes a more powerful language pulling in the best aspects of other successful programming languages like Python, C++ 11, and C#.

The advances in JavaScript combined with the work the World Wide Web Consortium (W3C) have put into the HTML standard has paved the way for modern applications in the web browser. Everything from live video conferencing to 3D graphics are now possible in a browser. Even emulators of old video game consoles such as the Super Nintendo have been developed in JavaScript and are completely playable in a browser.

As the browser becomes more and more powerful, the concept of Thin Severs is starting to dominate the environment. Instead of paying for a powerful server with enough computing power to handle all of the application logic, the logic is instead offloaded to the client, and the server simply becomes an interface for querying stored data and other services. This has led to the development of new powerful platforms, such as Node.js, which uses Google's open source JavaScript Engine, called V8. Now full stack development can be done entirely in JavaScript, and many frameworks have been created to help accomplish it. For example, the MEAN stack uses MongoDB, Express.js, Angular.js, and Node.js to enable quick, agile development of fast, scalable, data intensive applications using only JavaScript.

Even though JavaScript is not as fast as C or C++, its growing dominance in modern development practices is the motivation behind choosing it as the language for this project.

1.1.2 ARTIFICIAL NEURAL NETWORKS

Inspired by the biological model of how brains function, ANNs are a flexible and powerful tool with a wide range of applications in Machine Learning and Artificial Intelligence.

In 1943, neurophysiologist Warren McCulloh and logician Walter Pitts published a paper describing a mathematical model for how neurons work called the Threshold Logic Unit [1]. A few years later psychologist Donald Hebb published a book outlining a hypothesis of learning based on the mechanism of neural plasticity known as Hebbian learning [2]. Alan Turing was already applying these ideas in 1948 on his B-Type machines, and in 1954, Farley and Wesley A. Clark simulated a Hebbian Network at MIT [3]. Shortly thereafter Frank Rosenblatt created the perceptron in 1958, which is the basis for modern neural networks [4].

Rosenblatt's perceptron was modeled after a neuron; it took several binary inputs and produced a single binary output. He introduced the concept of weights, by associating real numbers to the inputs based on

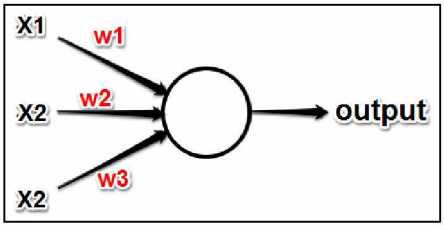


Figure 1 - Rosenblatt's perceptron

their importance to the output. This was a considerable improvement over McCulloh's and Pitt's Threshold Logic Unit. The output of the perceptron is 0 or 1, determined by whether the weighted sum $\sum_j w_i x_j$ exceeded a threshold value θ.

$$output = \begin{cases} 0 \; if \; \sum_j w_i x_j \leq \theta \\ 1 \; if \; \sum_j w_i x_j > \theta \end{cases}$$

Figure 2 - Perceptron Algorithm

In 1969, Marvin Minksy and Seymour Papert published *Perceptrons: an introduction to computational geometry*, highlighting two major issues surrounding the ANNs of the time [5]. The simple single-layer networks were incapable of processing the exclusive-or logical circuit, and computers were not powerful enough to handle the larger networks that were required to compute complex functions. This led to an abandonment of ANN research and related connectionism methods until improved methods and hardware were developed years later.

The problem with single-layer networks is that their output is only a relatively simple function of their input, and they are only capable of learning linearly separable patterns. For example, if neurons are related to logic gates in a circuit, then it would take at least two layers of neurons to create the exclusive-or circuit. This was understood by Minksy and Papert, however they were often misquoted as conjecturing that even multilayer networks would be similarly limited, leading to the decline in funding and stagnation of research until the 1980s.

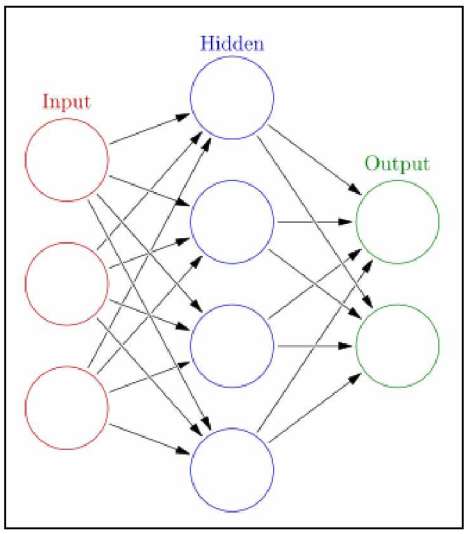


Figure 3 - Diagram of a Multilayer Perceptron or Feed Forward Network

In 1974, Paul Werbos published his PhD thesis, which included a backpropagation algorithm for multilayer perceptrons, or feed forward ANNs, effectively solving the exclusive-or problem [6]. This algorithm allowed for efficient supervised-learning where a desired output of a network was already known. This method requires a differentiable activation function be used by the neurons in order to calculate the partial derivative of the error with respects to the weights. This new artificial neuron's output is calculated by adding the weighted sum of the neuron's inputs to its bias, or threshold, and then passing that value to an activation function $\sigma(\sum_j w_i x_j + \theta)$. The result is the output of the neuron, and is subsequently used as the input to the next connected neuron, until it propagates through to the final output of the network. A commonly used activation function is the Sigmoid or Logistic Function as it is similar to

$$\sigma(z) = \frac{1}{1 + e^{-z}}$$

$$\frac{\partial \sigma}{\partial z} = \sigma(z)(1 - \sigma(z))$$

Figure 4 - Sigmoid Activation Function

the threshold step function, and has a nice derivative. The backpropagation algorithm is still very popular today for its ability to train extremely large ANNs.

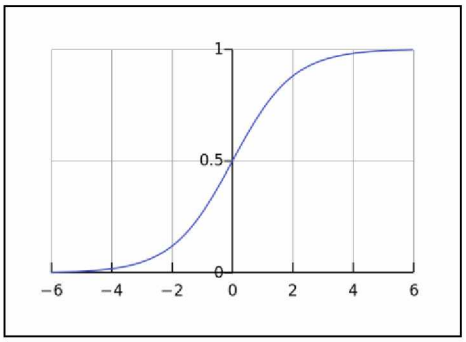


Figure 5 - Graph of Sigmoid Function

Despite a decline in popularity throughout the 1990s and 2000s, ANNs are now at the forefront of artificial intelligence and machine learning research. Almost all international competitions in pattern recognition and machine learning since 2009 have been won by ANNs. They are successfully used for many applications today, including pattern recognition, function approximation, data processing, robotics, and control. This rekindling of research and development in the field has led to the discovery of many new optimizations and training techniques in the last few years.

Some recent optimizations to the backpropagation algorithm include AdaDelta, an improved Gradient Decent optimization algorithm that removes the problem of choosing correct hyper-parameters [7]. Other research has been done on distributed neural networks capable of processing billions of parameters using thousands of processors [8]. Other unique training methods include utilizing genetic algorithms that mimic natural selection and Darwin's theory of evolution, by scoring a population of ANNs based on their performance, and then using the best ANNs to create the next generation by simulating chromosomal crossover.

Genetic Algorithms can be used to solve complex problems where a correct answer is not easily calculated or known. In 2001, David Fogel demonstrated that a Neural Network could learn to play checkers with this technique [9]. NASA used a genetic algorithm to design the ST5 spacecraft antenna.

Although there are an endless number of possible configurations for ANNs, the most common one is the feed-forward fully connected network, as in Figure 3. This is in part due to the universal approximation theorem for ANNs, which proves that multi-layer feed forward networks can approximate any continuous function on compact sets of $\mathbb{R}$.

Given the enormous amounts of data available on the internet, a high performance ANN that runs natively in a browser, the goal of this project, should be very useful. Some possible benefits could include being able to train on live user interaction and data, easily distributing the computation and training of large networks, or offloading computationally intensive ANNs to client computers, for example, facial recognition of photos for tagging users on Facebook or Google+.

1.1.3 PREVIOUS WORK

Neural Networks in JavaScript are not as original and new as initially thought when this project was proposed. Throughout the research for this project, it was discovered that a few JavaScript libraries for ANNs already existed.

Andrej Karpathy, a PhD student at Stanford created ConvNetJs [10]. This library specializes in Convolutional Deep Networks for image processing, and includes many possible optimization functions for backpropagation.

Juan Cazala, a Computer Engineering student, and JavaScript and .NET developer from Argentina, developed Synaptic.js. His library uses a strictly object oriented approach where connections between neurons and layers have to be explicitly mapped. It is described as an architecture-free neural network library, as any network configuration or layout is possible [11].

Heather Arthur, an engineer at Mozilla, created Brain.js [12]. This is a popular JavaScript Neural Network Library that has also seen some different variations of it, for example Einstein.js [13]. The library is extremely simple to use, as the only information needed is a training set of inputs and outputs, the library will automatically create and train a neural network with the correct number of inputs and outputs, and a hidden layer size based on the number of inputs and outputs of the training set.

All of these projects were developed by people with a good understanding of ANNs and JavaScript. Each library is distinctly different in its implementation and methods, and each has unique applications for which it is aptly suited. All of the libraries are also open-source, receiving contribution over the last few years from other developers as well.

1.1.4 LIMITATIONS OF PREVIOUS WORK

Although the existing JavaScript libraries of ANNs are excellent, they are all limited to a single supervised-learning training method using the backpropagation algorithm. In addition, it is unclear to what extent performance was considered in their implementations. All except Synaptic.js are limited to only being executed in the main thread. Synaptic.js has the option to create a web worker that will execute asynchronously in a separate thread.

1.2 INTRODUCING INTELLECT.JS

In order to gain a better understanding of Neural Networks, and learn how to develop a good JavaScript library, we developed Intellect.js. Initially developed as a simple neural network library optimized for Google's V8 JavaScript Engine, it also works on any modern standards compliant browser. It builds upon the existing libraries by adding a Genetic Algorithm for training, and utilizes the latest JavaScript features like multi-threading with web workers. Section 2 covers the design and implementation of Intellect.js, and Section 3 contains a comparison against the other libraries.

2. IMPLEMENTING INTELLECT.JS

Intellect.js started as a port of a previous C++ Neural Network library. This provided the benefit of already having a working C++ implementation to compare against for measuring V8's JavaScript performance in relation to compiled C++. As development continued, the JavaScript version quickly diverged as many enhancements and optimizations discovered throughout the course of the project were added to it.

2.1 JAVASCRIPT BEST PRACTICES

The design of Intellect.js draws from the standard practice of Object Oriented JavaScript development. First, the library's scope should be contained, and the library should be modular and not interfere with any other code or included libraries.

There are variety of methods for doing this in JavaScript [14], but all entail creating a global variable that will act as the namespace, or container, for the library. See Figure 6. This object can then be extended with methods, constructors, and classes.

```
// global namespace
var MYAPP = MYAPP || {};
```

Figure 6 - JavaScript OOP: Declaring a NameSpace

Since JavaScript is a prototype-based language, **class** does not exist as it does in other languages. Instead, functions assigned to variables are used to define classes as in Figure 7.

```
var Person = function () {};

var person1 = new Person();
var person2 = new Person();
```

Figure 7 - Classes in JavaScript

Everything that is declared within a JavaScript function is scoped to that function and will only be created and allocated when that function is called with the **new** keyword.

One feature of JavaScript however is that variables can be implicitly declared. Implicit variables are put in the Global Scope. Letting variables leak into the global scope is bad practice, and a number of tools exist to check for it. Scope leak can lead to memory and performance issues, as well as lead to unexpected behavior if existing global variables are overwritten unintentionally. See Figure 8.

```
var Person = function () {
     // Local Variable, deleted when
     // a person goes out of scope
     var variable1 = 0;

     // Implicit variable, not deleted
     // when it goes out of scope.
     variable2 = 0; // <-- variable2 is now Global!
};
```

Figure 8 - Implicit variable declaration and scope leak

Every JavaScript object has a prototype that is easily extended. This allows for the addition of methods and general information to any object. Everything from variables to functions can be added to a prototype as in Figure 9. This prototype method is also the fastest when compared to other methods such as module patterns with or without cached functions.

```
var Person = function(name) {
     this.name = name;
};
Person.prototype = {
     info: {
          height: 58,
          weight: 165,
          age: 29,
          gender: "male"
     }
     walk: function() {}
}
```

Figure 9 - JavaScript Prototype

The performance of the V8 JavaScript Engine is highly dependent on this idea of scoping and object properties. V8 is comprised of two independent compilers: Full-codegen and Crankshaft [15]. Full code-gen is the first compiler that produces slow and unoptimized code, and as V8 notices functions are called repeatedly, they are passed through Crankshaft and optimized during execution. However, Crankshaft code relies on assumptions regarding the execution and type of objects involved. If those are violated, then the optimized code is abandoned and execution is returned to the slower Full code-gen version. It is possible to crankshaft a function while it is executing, for example **for** loops, using a method called "on stack replacement".

Developing a high performance JavaScript library revolves around making sure it can be easily optimized by V8's Crankshaft compiler.

JavaScript's garbage collection process also relies on scoping. Memory is not reclaimed until it is no longer referenced. When a variable or object goes out of scope, the memory it was using gets recycled automatically. However, in Figure 8, Person objects would never be deleted, as variable2 will always be in scope unless manually dereferenced.

2.2 PERFORMANCE MINDED

The main goal of Intellect.js is to be fast and optimized for V8 Engine. In order to accomplish this, benchmarks of core JavaScript methods and data structures were measured on Google Chrome to determine the fastest and best techniques to use when developing the library. The tests were based around neural networks, and focused on how best to represent and use an ANN in JavaScript. All of the initial benchmarks were measured using a JavaScript performance analysis tool called jsPerf. JsPerf is powered by Benchmark.js, a robust JavaScript benchmarking library created by John-David Dalton that supports high-

Motherboard: ASUS P8Z77-V Pro

Processor: Intel i5 2500k

Memory: 16 GB DDR3

OS: Windows 8.1

Chrome: 41.0.2272.76

Figure 10 - Benchmark Computer & Browser Configuration

resolution timers, and returns statistically significant results [16]. Each benchmark suite was executed 10 times and the results were averaged together. The machine used was a common configuration to represent the most frequent use case for the library. The code used for the benchmarks can be found in Appendix B.

The first area tested was JavaScript array creation and access methods, to determine the best method for storing the weights and biases of an ANN in V8 Engine.

Next, the most frequently called function during the computation of a neural network, the activation function, was also evaluated. Performance analysis on the most common activation functions was performed to determine the fastest methods in V8 Engine.

2.2.1 ARRAYS

Neural Networks are most commonly represented by a collection of arrays or matrices containing the values of the weights and biases of each neuron. The first subject benchmarked and optimized is the method for storing and accessing the data of the network.

The first benchmark was for array access and filling. Interestingly there exists a rather opinionated stance towards best practice when it comes to JavaScript and Arrays. Standard convention recommends avoiding the **new** operator when instantiating arrays. However, the performance gain of pre-allocating the array is substantial.

It is apparent that V8 tries to optimize array allocation. Setting the length of an array to zero before filling it shows that V8 pays attention to the length field, as the performance was greatly impaired. This is most likely caused by the array being allocated twice, once when initialized with the **new** operator, and then again after resetting its length back to zero. Creating an array with the **new** operator was about three times faster than the conventional method. For large arrays, greater than 10,000 elements, V8 actually changes how it stores them internally in the object's hidden class, switching from an array of unsigned integers to a sparse array, or map. In order to force this optimization you can manually set the length property. The results of the benchmark are included in Figure 11.

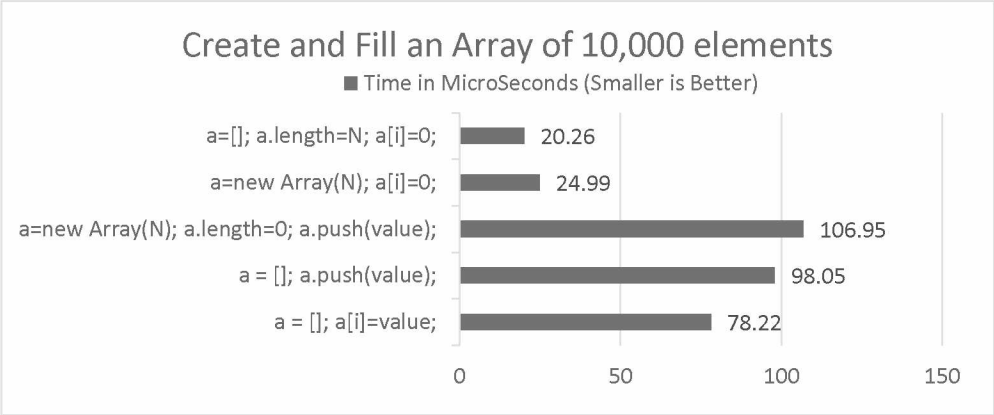


Figure 11 - Array Allocation and Filling Performance

The results are not surprising, as it requires fewer operations and time to allocate the space once, as opposed to multiple times throughout the loop.

When dealing with arrays, JavaScript is also capable of nesting arrays together to create multi-dimensional arrays. A common technique for improving performance is flattening the array and keeping track of rows and columns separately. This makes iterating

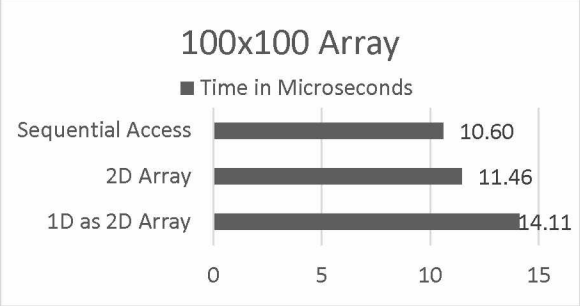


Figure 12 - 1D vs 2D Array Access

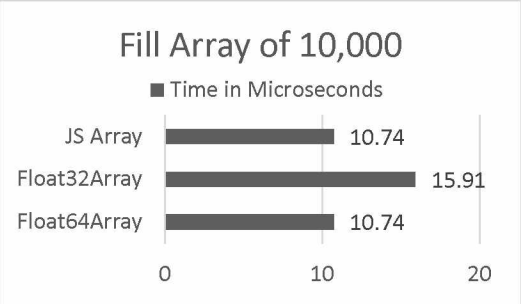


Figure 13 - Typed Array vs Normal Array

through the array a little more complicated, as accessing a specific element requires a calculation based on row and column size. The performance difference between one-dimensional and two-dimensional arrays was measured by creating a one-dimensional array of 100,000 elements, and a two-dimensional array of 100 by 100 elements, and then calculating the time it took to iterate through the rows and columns and set the values to zero. The result was that V8 was better at optimizing 2D array access than the row and column calculations for a 1D array. However, it was still slower than sequential array access. See the results in Figure 12.

A new feature added to JavaScript within the last few years is the ability to have typed arrays. These arrays are optimized for passing data to the GPU and to worker threads. A similar benchmark for array access was performed for typed arrays. Native JavaScript arrays had identical performance as Float64Arrays. This makes sense as JavaScript represents numbers internally as 64-bit floats, or doubles. [Figure 13] In order to test V8 Engine's ability to optimize typed array access in a function, a simple feed-forward network evaluation benchmark was performed. It tested the performance of multi-dimensional JavaScript arrays, versus one-dimensional typed arrays, versus one-dimensional typed array buffers accessed by multi-dimensional array views. A typed array buffer in JavaScript can have multiple views associated with it, allowing the creation of multi-dimensional arrays whose data is stored in a single buffer. This makes the calculations

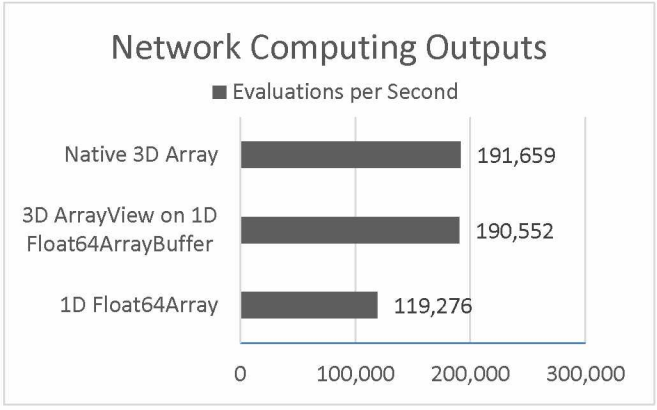


Figure 14 - Array comparison for Network Evaluation

for a network evaluation simpler, resulting in easier to optimize code. The testing results showed similar performance as the previous test, the native JavaScript arrays and array views on Float64Buffers were nearly identical[1].

These initial benchmarks on data structures showed that the optimizations built into V8 Engine work best when using standard JavaScript. Allowing the compiler to pick the best internal representations and optimize loops that access them leads to the best performance.

2.2.2 ACTIVATION FUNCTIONS

A number of activation functions were benchmarked to determine which functions performed the fastest in JavaScript. The results of the most common activations is found in Figure 15.

The most standard activation function, and the one used in all of the existing libraries, is the standard Logistic or Sigmoid function $\frac{1}{1+e^{-x}}$. This function is easily optimized for gradient descent, which is why it is the standard function for ANNs. However, the output of the standard logistic function is limited to 0 through 1.

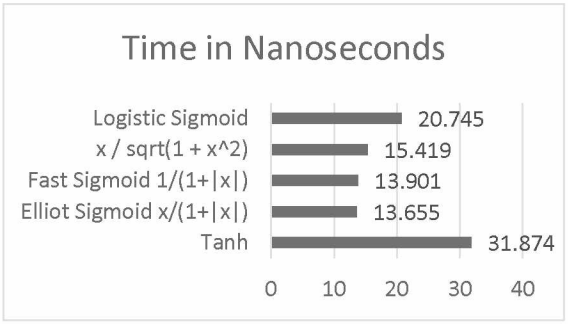


Figure 15 - Activation Function Benchmarks

Other symmetrical activation functions are commonly used in order to have an output that allows for negative values. Research also shows that symmetrical activation functions converge faster with backpropagation. The standard symmetrical activation function is the hyperbolic tangent, as it also has a derivative that lends itself nicely to gradient decent. The Elliot Sigmoid $\frac{x}{1+|x|}$ was published by David Elliot in 1993 as a faster activation function for ANNs as it required less computation [17].

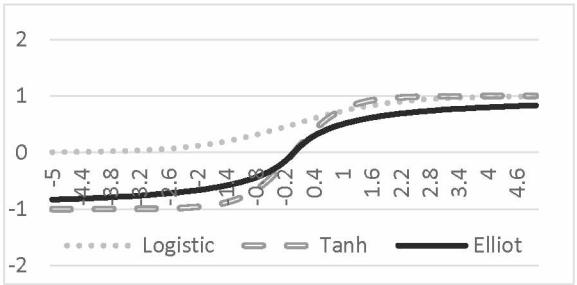


Figure 16 - Graph of Standard Activation functions

As demonstrated by the graphs of the activation functions in Figure 16, the Elliot Sigmoid shares many similar qualities with the hyperbolic tangent. In fact, for gradient

[1] The same benchmark was performed on a newer version of Chrome and Firefox. Both Chrome and Firefox improved the speed of native JavaScript arrays, while the array view remained about the same in Chrome. Array views perform the worst in Firefox.

decent it is possible to use the derivative of tanh for computing gradients, although more accuracy can be gained if using the actual derivative $\frac{1}{(1+|x|)^2}$.

The Elliot sigmoid function is much faster than the hyperbolic tangent function, and so it is the preferred symmetrical activation function in the library. In order to provide robustness and allow for networks trained outside of the library, the other activation functions are also included in the library.

2.3 LIBRARY DESIGN

Incorporating the results of the benchmarks, Intellect.js was designed to use multi-dimensional arrays to store the weights, biases, sums, and outputs. This not only provides increased performance, but also makes the calculations easier to write. The only downside is that multi-dimensional arrays are not transferable objects, meaning they cannot be quickly passed between worker threads without the overhead of structured cloning of the data [18].

Along with the weights and biases, the library also keeps track of a number of other values that are used during the training processes. For the genetic training method, each new Intellect object will have a fitness value that is updated based on the networks performance during training. For the backpropagation training algorithm, there are separate typed arrays for Errors, Previous Errors, and Previous Biases. These arrays, along with the sums and outputs, are not transferred to worker threads or saved when a network is exported as they are updated dynamically based on the weights and biases.

The following sections outline the methods and functions that are included in Intellect.js. The library is then compared against other libraries in section 3.2

2.3.1 INITIALIZATION

In order to keep the library simple to use, there is only a single method needed to create a network. All of the customization and options are available by passing a single object with only the keys and values of the options that need to be changed from the default behavior. This avoids the problem of having to remember specific parameter order when initializing a new Intellect, and is robust enough that erroneous parameters have no effect.

The auto-executing autonomous function that wraps the library checks to see if it is running in a worker thread, and defines the Boolean variable **is_worker** to reflect the execution scope of the library. If the library is called from the main thread, a built-in check makes sure that Intellect was invoked using the **new** keyword, thereby preventing unnecessary cluttering of the global window scope. If invoked without **new**, Intellect automatically calls itself with **new** and returns the result. The Intellect class variable is then added to the window scope so that it can be used outside the autonomous function. This allows a new Intellect to be created by either var net = **new** Intellect(), or var net = Intellect() in any subsequent JavaScript.

The autonomous function wrapper around the library also includes the messaging functions for communication between the worker and parent threads. The library defaults to using workers during training, but this can be disabled by setting the SingleThreaded option to **true** when creating an Intellect. There is a slight overhead involved with spawning a new worker thread and sending data between threads. However, single-threaded mode is blocking and so the web page will become unresponsive until training completes.

The default options for creating a new Intellect are displayed in Figure 17. The training method, activation function, and error functions can be either the name of a built-in method as a string, or an external function name. If an external function is used, web workers are disabled for training, as it is not possible to pass function references to workers.

The default network configuration is the smallest network required to calculate XOR. If a custom network configuration is provided, Intellect's default behavior is to create a hidden layer if one is not specified. The hidden layer is sized based on a rule-of-thumb method for determining the number of neurons in a hidden layer.

- The number of hidden neurons should be 2/3 the size of the input layer, plus the size of the output layer. [19]

```
var Intellect = function (options) {
  var defaultOptions = {
    Layers: [2, 2, 1],

    // Possible Options:
    // "BackPropagate"
    // "Genetic"
    // or provide your own function
    TrainingMethod: "BackPropagate",

    // Possible Options:
    // "ElliotSigmoid" = .5*x/(1+|x|)+.5
    // "Sigmoid" = 1/(1+e^-x)
    // "ElliotSymmetrical" = x/(1+|x|)
    // "HyperTan" = tanh(x)
    // or provide your own function
    ActivationFunction: "ElliotSigmoid",
    ActivationFunctionDerivative: "ElliotSigmoid",

    // Possible Options:
    // "MeanSquaredError"
    // "QuadradicCost"
    // "CrossEntropy"
    // or provide your own function
    ErrorFunction: "MeanSquaredError",

    // Disable Web Workers (only used for training)
    SingleThreaded:false,

    // Create a hidden layer if one is not provided
    AutomaticHiddenLayer:true

  };

  if (!is_worker) {
    if (window == this) {
      // Prevent Cluttering the Window Scope;
      return new Intellect(options);
    }
  }
        ...
}
// Main JavaScript
var myNetwork = new Intellect();
```

Figure 17 - Intellect Constructor Options

The constructor then associates the training method, and activation and error functions used by the various methods in the library with those specified at creation time.

Lastly, the network is initialized with a random set of weights and biases optimized for learning. In order to prevent neuron saturation, the weights are Gaussian random numbers

with a mean 0 and standard deviation $1/(\sqrt{n_{input}})$ [20]. This makes it so the weighted sum $\sum_j w_i x_j + \theta$ that is passed to the activation function for a neuron is also a Gaussian random number but with a tighter peak closer to 0 than a standard distribution. The biases are Gaussian random numbers with a mean 0 and standard deviation of 1.

Since JavaScript does not have a built-in normal distribution random number generator, Intellect.js includes its own optimized generator, rand_normal. The generator is quicker and more efficient than the Box-Muller transform, yet yields almost identical results. The

	Box-Muller Transform	rand_normal
Number of Samples	100,000	100,000
Histogram		
Average	0.002437297901841488	-0.0008916877223225311
Standard Deviation	1.001012637246901	1.0045330248825375
Performance	8,374,245 Ops/sec	975,283,349 Ops/sec

Figure 18 - Comparison of Box-Muller transform and optimized generator

major difference is that rand_normal will never return a value more than three away from the mean of the zero. This is actually ideal in its current use case, as the reason for the normal distribution is to cluster the random values closer to zero, such that the neurons do not become saturated by always having their outputs near the maximum of the activation function.

The rand_normal function is called by various helper functions that are used during network initialization. These helper functions are all defined in the prototype of Intellect, and are used by the initialize function.

```
rand_normal: function () {
  return (Math.random() * 2 - 1) + (Math.random() * 2 - 1) + (Math.random() * 2 - 1);
},

randWeights: function (size) {
    var weights = new Array(size);
    for (var i = 0; i < size; i++) {
        weights[i] = this.rand_normal() / Math.sqrt(size);
    }
    return weights;
},

randBias: function (size) {
    var biases = new Array(size);
    for (var i = 0; i < size; i++) {
```

```
        biases[i] = this.rand_normal();
    }
    return biases;
},

zeros: function (size) {
    var arr = new Array(size);
    for (var i = 0; i < size; i++) {
        arr[i] = 0.0;
    }
    return arr;
},
```

Figure 19 - Optimized Normal Distribution Generator and Initialization Helper Functions

2.3.2 ACTIVATION FUNCTIONS

Intellect.js includes four standard activation functions, but expanding to include more is trivial. All of the activation functions are stored within the Activation Functions object within the Intellect prototype. The prototype also contains an Activation Function Derivatives object that contains the partial derivatives of the various activation functions for use with the backpropagation algorithm.

```
ActivationFunctions: {
    Sigmoid: function (x) {
        return 1.0 / (1.0 + Math.exp(-x));
    },
    HyperTan: function (x) {
        return Math.tanh(x);
    },
    ElliotSigmoid: function (x) {
        return .5*x / (1 + Math.abs(x))+.5;
    },
    ElliotSymmetrical: function (x) {
        return x / (1 + Math.abs(x));
    }
},

ActivationFunctionDerivatives: {
    Sigmoid: function (output, input) {
        return (1 - output) * output;
    },
    HyperTan: function (output, input) {
        return (1 - output) * (1 + output);
    },
    ElliotSigmoid: function (output, input) {
        return 1 / ((1 + Math.abs(input)) * (1 + Math.abs(input)))
    },
    ElliotSymmetrical: function (output, input) {
        return .5 / ((1 + Math.abs(input)) * (1 + Math.abs(input)))
    }
},
```

Figure 20 - ActivationFunctions and ActivationFunctionDerivatives Objects

Storing the functions in an object makes it possible to convert from the name of an

activation function passed as a string in the initialization options, to an executable function. To improve performance the chosen activation function and its derivative are cached in the Intellect prototype by the constructor.

```
if (typeof(this.options.ActivationFunction) !== 'function') {
    this.ActivationFunction =
this.ActivationFunctions[this.options.ActivationFunction];
}
else {
    this.SingleThreaded = true;
    this.ActivationFunction = this.options.ActivationFunction;
}
```

Figure 21 – Converting a String Function Name to an Executable Function

This also allows the Activation Function option specified at creation time to be either the name of an internal function as a string, or an external function. If an external function is used, ActivationFunctionDerivative is an optional value that can also be passed for use in backpropagation algorithm.

The Elliot approximations are faster to compute, but have a downside of taking more iterations to train. When training a network to be a binary classifier (output of 0 or 1), a Sigmoid activation function is recommended. If negative outputs are required, then a symmetrical activation such as HyperTan or its Elliot approximation is recommended.

2.3.3 ERROR FUNCTIONS

Error functions are used for the backpropagation algorithm. Intellect.js includes three common error functions used for computing the error of a network based on the desired outputs, $\hat{Y}$, and the actual outputs, Y, over the number training items, n.

- Mean Squared Error

$$\frac{1}{n}\sum_{i=1}^{n}\left(\hat{Y}_i - Y_i\right)^2$$

- Quadratic Cost

$$\frac{1}{2n}\sum_{i=1}^{n}\left(\hat{Y}_i - Y_i\right)^2$$

- Cross Entropy

$$-\frac{1}{n}\sum_{i=1}^{n}\left(\hat{Y}_i \ln Y_i + \left(1-\hat{Y}_i\right)\ln(1-Y_i)\right)$$

Since each error function is different, the same error does not properly signify the accuracy of the network across different error function. Cross Entropy's error is about 10 times the Mean Squared Error for similarly performing networks. This means the error threshold value that determines when a network should stop training is different based on the Error function and size of training data.

Cross Entropy is the slowest error to compute, but works better for specific optimizations to the backpropagation algorithm. For example, mini-batch Stochastic Gradient Descent works well with Cross Entropy and it only needs to be calculated once for each batch, making its slowness more acceptable.

The default Error function is the Mean Squared Error. This is faster to compute than Cross Entropy, and still fast to train with. If a different Error function is chosen, it is also a good idea to provide a more appropriate error threshold value when training.

Error Functions are defined similarly to the activation functions and their derivatives. There is an Error Functions object in Intellect's prototype that contains the functions for computing the various errors. When a new Intellect is created, the appropriate Error function is cached from the object using the same method as the Activation Function.

2.3.4 COMPUTING OUTPUTS

The output for a specific neuron is defined by

$$a_j^l = \sigma(\sum_k w_{jk}^l a_k^{l-1} + b_j^k)$$

Were a_j^l is the activation for the j^{th} neuron in the l^{th} layer, w_{jk}^l is the weight from the k^{th} neuron in the $(l - 1)^{th}$ layer to the j^{th} neuron in the l^{th} layer. The input values to the networks are considered the activations for the first layer, in other words $a_j^1 = input_j$. In order to understand the equation better a graphical representation of a network is helpful.

If we represent the individual weights, biases, and activations as vectors, the above equation can be rewritten as simply

$$a^l = \sigma(w^l a^{l-1} + b^l)$$

Intellect.js represents the weights, activations, and biases of the network as arrays of vectors, or multi-dimensional arrays. This makes the activation calculation easier to write, and increases performance according to the earlier benchmarks (see Figure 14).

Instead of creating a Matrix class with functions to handle the computation of sums and dot products, nested for loops are used within the output function to

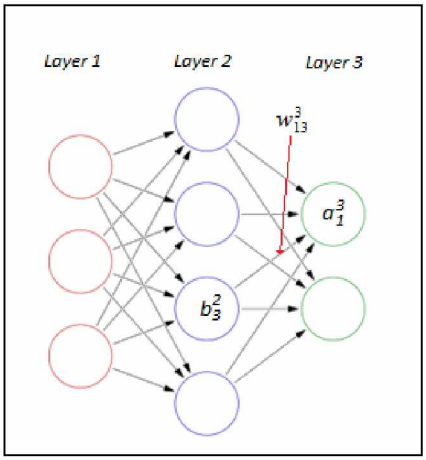


Figure 22 - Labeled Network Topography

perform the matrix calculations in place. Starting at the first layer of the network, neurons and their weights are iterated over, calculating $\sigma(\sum_j w_i x_j + \theta)$ for each neuron, where σ is the activation function specified for the network.

Sums are stored only because they are needed for the backpropagation algorithm. A possible optimization would be removing it from the computation if not training with the backpropagation algorithm. Performance is also gained by caching the length of an array or value at the beginning of the for-loop.

```
ComputeOutputs: function (input) {
    this.Outputs[0] = input;
    // Compute the Outputs of each Layer
    for (var layer = 1; layer <= this.outputLayer; layer++) {
        for (var neuron = 0, neurons = this.Layers[layer]; neuron < neurons; neuron++){
            var weights = this.Weights[layer][neuron];
            var sum = this.Biases[layer][neuron];
            for (var k = 0, length = weights.length; k < length; k++) {
                sum += weights[k] * input[k];
            }
            this.Sums[layer][neuron] = sum;
            this.Outputs[layer][neuron] = this.ActivationFunction(sum);
        }
        var output = input = this.Outputs[layer];
    }
    return output;
},
```

Figure 23 - ComputeOutputs Method

Another possible optimization would be converting the entire computation to a string that can then be evaluated and executed using the **eval()** function. Synaptic.js uses this approach in its standalone method, which converts the entire network evaluation into a standalone function that can be used anywhere. To test the performance a ComputeOutputsToString method was added to Network3D in the Array Comparison for Network Evaluation benchmark. The method converts the entire output computation into a single string that is then evaluated to a function and assigned to a method. When compared with the ComputeOutputs method, the evaled function is eleven times slower to evaluate for networks of any size. This is probably because a new output array needs to be allocated on every call, whereas the ComputeOutputs method operates in place on existing arrays. For large networks, the standalone function that is generated could be larger than the entire Intellect.js library, making it more efficient to include the library and import a saved network rather than embedding an extremely large function.

```
ComputeOutputsToString = function(input){
    var OutputFunction = "(function(_inputs){ var _input, _previous; _previous=_inputs;
_outputs=[];"
    for (var layer = 1; layer <= this.outputLayer; layer++) {
        for (var node = 0, nodes = this.Layers[layer]; node < nodes; node++) {
            var weights = this.Weights[layer][node];
            var sum = this.Biases[layer][node];
            OutputFunction += "_input="+sum;
            for (var k = 0, length = weights.length; k < length; k++) {
                OutputFunction += "+ (_previous["+k+"] * "+ weights[k]+")";
```

```
            }
            OutputFunction += ";_outputs["+node+"] = 1/(1+Math.exp(-_input));";
        }
        OutputFunction += "_previous = _outputs;"
    }
    OutputFunction += "return
_outputs.splice("+0+","+this.Layers[this.outputLayer]+");})";
    this.OutputFunction = eval(OutputFunction);
}
```

Figure 24 - Converting Network Evaluation to Standalone function

2.3.5 TRAINING METHODS

The library includes two different training methods, backpropagation for supervised-learning, and a genetic algorithm for unsupervised-learning. Training methods are defined in the Intellect prototype in a TrainingMethod object similar to the activation and error function objects. At creation time, the specified training method function is cached using the same method as the activation and error functions. An external function name can also be used in place of one of the internal methods.

The Train method is used to train the network using whatever method is specified. It acts as a wrapper, simply calls the specified training algorithm, calls the callback function if one was provided, and then returns the result. Since each training function makes different uses of web workers, the creation and management of workers are done in the specific training methods. Web Workers are discussed in more detail in the next section.

```
Train: function (data, options, callback) {
    options = options || {};
    options.callback = options.callback || function () {};
    callback = callback || function () {};
    this.Trained = false; // Used to know when a worker thread is done training
    var result = this.TrainingMethod(data, options, callback);
    callback(result);
    return result;
},
```

Figure 25 - Train method

Since all JavaScript functions are variadic, all the parameters are optional. The method checks for undefined parameters with the logical OR operator, and assigns a default value if it is missing, just in case it is referenced or used elsewhere. However, no training can be done without proper input data, or a fitness measuring function for the Genetic algorithm.

2.3.5.1 BACKPROPAGATE

Backpropagation is the most common supervised-learning technique. It consists of three steps:

1. Do a forward pass, by calculating the output of a training pattern's input.

2. Do a backward pass, by propagating the error between the output and training pattern target backwards through the network and storing the deltas.
3. Update the weights by a ratio of the gradient of the error.

It utilizes gradient descent to minimize some cost function, or in the case of this library, an Error Function. The partial derivative of the chosen error function with respect to the weights of the network is required in order to calculate the deltas of the neurons and update the weights and biases. The three error functions included in the library also each contain their own delta function, which will calculate the appropriate delta of the output neurons for the specified Error function.

To demonstrate the math behind the backpropagation algorithm, it is beneficial to recall that the output of a neuron is defined as $a_j^l = \sigma\left(\sum_k w_{jk}^l a_k^{l-1} + b_j^k\right)$. Since $\sum_k w_{jk}^l a_k^{l-1} + b_j^k$ is such a common term, it is often denoted as z_j^l, thus the output becomes simply $a_j^l = \sigma(z_j^l)$. Also, recall that w_{jk}^l denotes the weight going to the j^{th} neuron in the l^{th} layer from the k^{th} neuron in the $(l\text{-}1)^{th}$ layer.

The partial derivative of the error with respect to a weight w_{jk} is calculated using the chain rule twice.

$$\frac{\partial E}{\partial w_{jk}^l} = \frac{\partial E}{\partial a_j^l}\frac{\partial a_j^l}{\partial z_j^l}\frac{\partial z_j^l}{\partial w_{jk}^l}$$

For simplicity, the partial derivative of the Quadratic Error function would be as follows.

Starting at the far right term, there is only one term in z that depends on w_{jk}^l

$$\frac{\partial z_j^l}{\partial w_{jk}} = \frac{\partial}{\partial z_j^l}\left(\sum_k w_{jk}^l a_k^{l-1} + b_j^k\right) = a_k^{l-1}$$

The derivative of the output of the k^{th} neuron with respect to its inputs is simply the partial derivative of the activation function, which is why the activation function must be differentiable for backpropagation to work. For this example, we will assume that the sigmoid function is used. Then the partial derivative becomes

$$\frac{\partial a_j^l}{\partial z_j^l} = \frac{\partial}{\partial z_j^l}\sigma(z_j^l) = \sigma(z_j^l)\left(1 - \sigma(z_j^l)\right) = a_j^l(1 - a_j^l)$$

The first partial derivative in the chain is simple if computing an output neuron, as a_j^L is equal to the neuron's output Yj.

$$\frac{\partial E}{\partial a_j^L} = \frac{\partial E}{\partial Y_j} = \frac{\partial}{\partial Y_j}\frac{1}{2}\left(\widehat{Y}_j - Y_j\right)^2 = Y_j - \widehat{Y}_j = a_j^L - \widehat{Y}_j$$

The term is more complicated if the neuron is in a hidden layer, as E is a function of all neurons receiving input from neuron k.

$$\frac{\partial E}{\partial a_j^l} = \frac{\partial E(a_j^l)}{\partial a_j^l} = \frac{\partial E(z_u^{l+1}, z_v^{l+1}, \ldots, z_w^{l+1})}{\partial a_j^l}$$

Taking a total derivative with respect to a_j^l, the following expression is obtained.

$$\frac{\partial E}{\partial a_k^l} = \sum_{l \in L} \left(\frac{\partial E}{\partial z_l} \frac{\partial z_l}{\partial a_{l+1}} \right) = \sum_{l \in L} \left(\frac{\partial E}{\partial a_l} \frac{\partial a_l}{\partial z_l} w_{kl} \right)$$

Meaning that derivatives must be calculated in reverse, hence the term backpropagation.

The result is

$$\frac{\partial E}{\partial w_{jk}^l} = \delta_j a_k^{l-1}$$

$$\delta_j = \frac{\partial E}{\partial a_j^l} \frac{\partial a_j^l}{\partial z_j^l} = \begin{cases} \left(a_j^L - \widehat{Y}_j\right) a_j^l \left(1 - a_j^l\right) \ \textit{if } j \textit{ is an output neuron} \\ \sum_{l \in L} \left(\frac{\partial E}{\partial a_l} \frac{\partial a_l}{\partial z_l} w_{kl} \right) a_j^L \left(1 - a_j^L\right) \ \textit{if } j \textit{ is a hidden neuron} \end{cases}$$

To update a weight, a learning rate α is multiplied against the weight's error to determine the amount the weight should change in order to fix the error.

$$\Delta w_{jk} = \alpha \frac{\partial E}{\partial w_{jk}^l}$$

This whole value is either subtracted from the weight, or added to it after multiplying it by -1. Without the subtraction, it would be moving up the gradient and increasing the error.

Since the output error is scaled by the gradient, output neurons that are already saturated have a very small gradient as they are at the part of the activation function that is leveling off. This small gradient means very small changes are made, and training can take a long time to move in any direction. This stagnation due to small gradients is a common cause for learning slowdown.

The Cross Entropy error function actually fixes this type of learning slowdown for the sigmoid function, as taking its derivative cancels out the derivative of sigmoid, removing the gradient of the output from the equation. Thus, the output error of a neuron is simply $a_j^L - \widehat{Y}_j$.

In practice, however, using the more effective $a_j^L - \widehat{Y}_j$ delta works for any Error function, as long as the learning rate, and error threshold values are also adjusted. This means that the calculation of the deltas can be computed the same way regardless of which error function is selected.

The backpropagation algorithm in Intellect features a small improvement over the standard backpropagation algorithm, by using the concept of momentum. Allowing larger changes to be made each step if the direction of the gradient has not changed since the last step. In order to utilize momentum, previous deltas are also kept along with current deltas.

The BackPropagate method takes an array of training data, and an optional object that contains any specific training options, and an optional callback function to execute when training is finished. It will then perform the backpropagation process until either the error threshold is met, or the max number of epochs are performed. For each epoch, it will iterate over the training data, and perform the backpropagation algorithm. Once it finishes it will return the error, the number of epochs, and the number of iterations of the backpropagation algorithm it performed.

It is also possible to have the current training status logged or another function called at specified intervals in order to monitor the training. Since Backpropagation is a serial process, only one web worker is created to handle the training.

```
BackPropagate: function (data, options, callback) {
    var epochs = options.epochs || 20000;
    var shuffle = options.shuffle || false;
    var errorThresh = options.errorThresh || 0.01;
    var log = options.log || false;
    var logPeriod = options.logPeriod || 500;
    options.learningRate = options.learningRate || 0.5;
    options.momentum = options.momentum || .2;
    var callbackPeriod = options.callbackPeriod || 100;
    var dataLength = data.length;
    options.learningRate /= dataLength;

    // Create a WebWorker to handle the training
    if (!is_worker || !this.SingleThreaded) {
        // Create a Worker and have it train
    }
    else {
        var error = 999999999999;

        for (var epoch = 0; epoch < epochs && error > errorThresh; epoch++) {
            var sum = 0;
            if(shuffle){this.shuffle(data);}
            for (var n = 0; n < dataLength; n += batchSize) {
                // Feed Forward
                var actual = this.ComputeOutputs(data[n].input);
                // Compute the Deltas & Errors
                var expected = data[n].output;
                sum += this.ErrorFunction.error(expected, actual, dataLength);
                this.CalculateDeltas(expected);
                // Update Weights and Biases
```

```
                this.UpdateWeights(options);
            }

            error = sum;
            if (log && (epoch % logPeriod == 0)) {
                console.log("epoch:", epoch, "training error:", error);
            }

            if (callbackPeriod && epoch % callbackPeriod == 0) {
                if (is_worker) {
                    // Worker Callback function
                }
                else {
                    options.callback({epoch: epoch, error: error})
                }
            }
        }
        return {
            error: error,
            epoch: epoch,
            iterations: dataLength * epoch
        };
    }
}
```

Figure 26 - BackPropagate Function

At the beginning of each epoch, the training set can be shuffled to help prevent overfitting on specific values in the training set. The shuffle is a simple in-place Fisher-Yates shuffle. Then for each item in the training set, ComputeOutputs is called, and then the deltas are computed based on the expected values. Lastly, the weights and biases of the network are updated.

```
// In place Fisher-Yates shuffle.
shuffle: function (array) {
    var m = array.length, t, i;
    // While there remain elements to shuffle...
    while (m) {
        // Pick a remaining element...
        i = Math.floor(Math.random() * m--);
        // And swap it with the current element.
        t = array[m];
        array[m] = array[i];
        array[i] = t;
    }
    return array;
},
```

Figure 27 - Fisher-Yates Shuffle Algorithm

The calculation of the deltas is a simple backwards pass through the network, propagating the error backwards given the expected outputs.

```
CalculateDeltas: function (tValues) {
// Calculate Deltas, must be done backwards
for (var layer = this.outputLayer; layer >= 0; layer--) {
    for (var node = 0; node < this.Layers[layer]; node++) {
```

```
        var output = this.Outputs[layer][node];
        var input = this.Sums[layer][node];
        var error = 0;
        if (layer == this.outputLayer) {
            error = output - tValues[node];
        }
        else {
            var deltas = this.Deltas[layer + 1];
            for (var k = 0; k < deltas.length; k++) {
                error += deltas[k] * this.Weights[layer + 1][k][node];
            }
        }
        this.Deltas[layer][node] = error * this.ActivationFunctionDerivative(output,
input);
    }
}
},
```

Figure 28 - CalculateDeltas Function

Updating weights and biases can be done in any order, so a simple forward pass through the network is done. The weights and biases are then adjusted using the learning rate and momentum values provided, along with the previously calculated deltas.

```
UpdateWeights: function (options) {
    // Adjust Weights
    for (var layer = 1; layer <= this.outputLayer; layer++) {
        var incoming = this.Outputs[layer - 1];
        for (var node = 0, nodes = this.Layers[layer]; node < nodes; node++) {
            var delta = this.Deltas[layer][node];
            for (var k = 0, length = incoming.length; k < length; k++) {
                var prevWeightDelta = (options.learningRate * delta * incoming[k]) +
(options.momentum * this.prevWeightsDelta[layer][node][k]);
                this.prevWeightsDelta[layer][node][k] = prevWeightDelta;
                this.Weights[layer][node][k] -= prevWeightDelta
            }
            this.Biases[layer][node] -= options.learningRate * delta;
        }
    }

},
```

Figure 29 - UpdateWeights Function

2.3.5.2 GENETIC

The unsupervised learning algorithm provided in Intellect.js is a simple Genetic Algorithm. This method creates a population of Intellects and evolves a new generation every epoch by simulating mating of the best performing Intellects and adding a little mutation to the offspring.

The first part of the genetic algorithm is determining the fitness of the population. A provided function will be called and passed the population for the fitness calculation. If no function is provided, it will default to an internal fitness function that determines the

fitness based on subtracting the error of the network on the provided training data from zero. For this reason, the genetic method can take a training set, although for practical purposes backpropagation is always recommended when a training set is available. Usual use cases for a genetic algorithm are simulations where certain behaviors are rewarded, and others punished.

After the population's fitness has been calculated, a genetic epoch is performed which creates the next generation. First, the population is sorted by fitness, and different metrics for the population are determined such as total fitness, average fitness, best fitness, and worst fitness. Next, in order to preserve the best performing networks in the next generation, a certain number are copied to the new population. Then until the new population is the same size as the old population, the best performing networks in the old population are randomly mated together, and their children are randomly mutated and added to the new population.

The mating function in the genetic algorithm is a simple chromosomal crossover, where the chromosomes for an Intellect are its Weights and Biases. A random split point in the chromosome is calculated, and then the Weights and Biases of the offspring networks are determined by subsections of the parent's chromosome, alternating between mom and dad based on the split point.

The mutation function then goes through the network and scales the weights and biases by a random number within a specified mutation rate. This adds a little variety to the population that might make the offspring perform better than their parents did.

After the new population is created, another genetic epoch is performed on the new population. This process repeats until a specified fitness is reached, or the max number of epochs have been performed.

In order to increase performance, the process makes use of web. Similar to how the Backpropagate method can work in a single web worker, the Genetic Algorithm will also spawn off a web worker. The web worker will perform the crossover and mutation, and then return the new offspring. This process can be parallelized making the creation of multiple workers ideal.

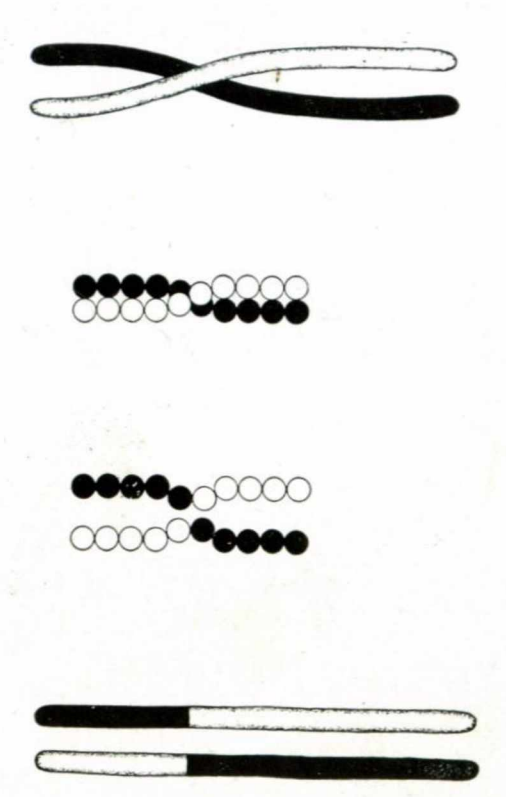


Figure 30 - Thomas Hunt Morgan's illustration of crossing over (1916)

Multi-threading can also be disabled by setting SingleThreaded to true when initializing the Intellect. The library will also default to running

in the main thread if workers are unsupported.

2.3.6 CREATING NEW WORKERS

Web Workers allow multi-threading in JavaScript. However, there are some limitations to them. For example, you cannot directly manipulate the DOM from a worker thread, or access the window object [21]. Web workers can do almost everything else though, including creating more web workers. Communication between worker threads and the main thread is accomplished via message passing.

There are four different types of web workers.

- Dedicated Worker: the default web worker, only accessible by the thread that created it.
- Shared Worker: can be used by multiple scripts running in different windows, iframes, tabs, etc., as long as they are in the same domain. Communication with shared workers is done via an active port.
- Service Worker: acts as a proxy server that sits between a web applications and the browser and network. Intended to improve offline experiences by intercepting network requests and taking appropriate actions based on network connectivity.
- Audio Worker: provides the ability for direct scripted audio processing to be done inside a web worker context.

A dedicated worker is created by creating a new worker and passing the URI of a script to execute in the worker thread.

```
var myWorker = new Worker("worker.js");
```

In order to create Web Workers without the need of another JavaScript file, the entire library is wrapped in a self-executing autonomous function that executes the moment the library is included on a web page. This allows the script to find its path relative to the webpage and store it for creating web workers later.

```
(function (global) {
    var is_worker = !this.document;

    var script_path = is_worker ? null : (function () {
        // append random number and time to ID
        var id = (Math.random() + '' + (+new Date)).substring(2);
        document.write('<script id="wts' + id + '"></script>');
        return document.getElementById('wts' + id).
            previousSibling.src;
    })();

    function msgFromParent(e) {
        // event handler for parent -> worker messages
    }

    function msgFromWorker(e) {
        // event handler for worker -> parent messages
    }

    function new_worker() {
```

```
        var w = new Worker(script_path);
        w.addEventListener('message', msgFromWorker, false);
        return w;
    }

    if (is_worker) {
        var workerIntellect; // Holds the network for worker threads
        global.addEventListener('message', msgFromParent, false);
    }
    // … Rest of library code here
})(this);
```

Figure 31 - Web Worker Functions

Calling new_worker() will create a new dedicated worker that has access to the entire Intellect.js library. This allows a web worker to create new Intellect objects and access all of the methods. The autonomous function also provides the added benefit of preventing scope creep, as any variables declared within the function will not be accessible unless manually added to the window scope. For example, at the end of the autonomous function, the Intellect object is added to the window scope.

```
if (!is_worker) {
        window.Intellect = Intellect;
}
```

In order to get data between workers and the main thread, JavaScript implements a message passing technique. A worker includes a postMessage() function that sends information between threads, be that a command, data, or both. Binding functions to the **message** event for both the main thread and the worker thread allows those messages to be processed when received.

```
this.worker = new_worker();
this.worker.postMessage({
    command: "Train",
    options: this.options,
    weights: this.Weights,
    biases: this.Biases,
    data:data,
    trainingOptions: options
});
```

Figure 32 - Passing Messages to Workers

The only time consuming operation in Intellect is training, so that is currently the only functionality built into a worker's message event function.

```
function msgFromParent(e) {
    // event handler for parent -> worker messages
    switch (e.data.command) {
        case "Train":
            workerIntellect = new Intellect(e.data.options);
            workerIntellect.Weights = e.data.weights;
            workerIntellect.Biases = e.data.biases;
            var result = workerIntellect.Train(e.data.data, e.data.trainingOptions);
            self.postMessage({
                command:"Trained",
                result:result,
```

```
                weights:workerIntellect.Weights,
                biases:workerIntellect.Biases
            });
            self.close();
            break;
        case "Stop":
            self.postMessage({
                command:"Trained",
                result:workerIntellect.error,
                weights:workerIntellect.Weights,
                biases:workerIntellect.Biases
            });
            self.close();
            break;
        default:
            console.log("Worker received",e.data);
            break;
    }
}
```

Figure 33 - Worker's Message Processing Function

The main thread also listens for messages from a worker. When a when worker finishes training, or is stopped, it will return the new weights and biases by posting a message to the main thread. In order for the main thread's messaging function to reference the parent Intellect that called it, as well as call any callback functions provided to the original training function, the worker prototype is extended at creation time and those values are stored so they can be referenced when a worker sends a message.

```
var IntellectThis = this;
this.worker.prototype = {
    _this:IntellectThis,
    Result: false,
    trainingCallback:options.callback,
    callback:callback
};
this.worker.postMessage(...);
```

The main thread then replaces the old weights and biases in the parent Intellect object with the new ones from the worker.

```
function msgFromWorker(e) {
    // event handler for worker -> parent messages
    switch(e.data.command){
        case "Trained":
            this.prototype._this.Weights = e.data.weights;
            this.prototype._this.Biases = e.data.biases;
            this.prototype._this.Trained = true;
            this.prototype.callback(e.data.result);
            break;
        case "TrainingCallback":
            this.prototype.trainingCallback(e.data.result);
            break;
        default:
            console.log("Message from Worker",e.data);
    }
}
```

Figure 34 - Main Thread's Message Processing Function

2.3.7 SAVING AND LOADING NETWORK CONFIGURAITONS

After a network is trained, it is important to be able to save it and load it again later. JavaScript has a built-in method for converting objects into human readable text. JavaScript Object Notation or JSON is a very common format that is available in many different programming languages and platforms. Using this format will make networks more portable and easy to use, enabling networks to be trained outside of the browser. This makes saving an Intellect very easy and quick. The saveNetwork function creates a simple object that contains the Networks configuration options, weighs, and biases, and then converts that object into JSON. A JavaScript Blob is created with the JSON string and a mime-type of octet/stream. An object URL is then created that links to the blob. To save the file to the client's computer, the object URL is added as a hidden link to the DOM and the click event is triggered on it.

```
saveNetwork: function () {
    var data = {options: this.options, weights: this.Weights, biases: this.Biases};
    var fileName = "Network - " + new Date() + ".json";
    var a = document.createElement("a");
    document.body.appendChild(a);
    a.style = "display: none";
    var json = JSON.stringify(data),
        blob = new Blob([json], {type: "octet/stream"}),
        url = window.URL.createObjectURL(blob);
    a.href = url;
    a.download = fileName;
    a.click();
    window.URL.revokeObjectURL(url);
}
```

Figure 35 - saveNetwork Function

Loading a Network is even easier. A JSON string is converted back into an object, and then the network is re-initialized with the new options. The Weights and Biases are then updated with the weights and biases from the JSON object.

```
loadNetwork: function (data) {
    data = JSON.parse(data);
    this.options = data.options;
    this.Initialize(this.options.Layers);
    this.Biases = data.biases;
    this.Weights = data.weights;
}
```

Figure 36 - loadNetwork Function

3. RESULTS

Intellect.js achieved its main goal of being a fast and flexible library for training neural networks. In order to test and demonstrate this a number of examples were developed to utilize the library. To test the performance of the library, it was also compared against the original C++ code that was the reference for the initial creation. In addition, the finished

library was also benchmarked against the existing neural network libraries to compare performance.

3.1 WORKING EXAMPLES

We developed three standalone examples that demonstrate Intellect's abilities. The first demonstrates a network's ability to learn to compute the exclusive-or function using the backpropagation algorithm. The second example demonstrates a network's ability to learn to recognize hand written characters. The last example demonstrates the genetic algorithm by training a network to compute XOR.

3.1.1 COMPUTING XOR

The simplest test to determine if a network is training correctly is verifying that it can learn the XOR function. It takes about four milliseconds on average to train an Intellect network to compute XOR. The result is that the network can compute values that are less than 0.1 away from the desired values. This error will continue to shrink if more iterations of the training algorithm are performed. Due to the nature of the activation functions, it is impossible to get exact outputs of 0 or 1. If 0 or 1 is required, a binary classifier can be used to set outputs to 0 or 1 based on some threshold value.

Since the network required for calculating XOR is so small, the training takes very little time. It actually takes longer to create a worker thread than it does to train the network on the main thread; therefore web workers are turned off in the example. The creation and training on an Intellect is as follows.

```
var XORNetwork = new Intellect({SingleThreaded:true});
XORNetwork.Train([
    {input: [0, 0], output: [0]},
    {input: [1, 0], output: [1]},
    {input: [0, 1], output: [1]},
    {input: [1, 1], output: [0]}]);
// takes about 2 - 10ms
// returns Object {error: 0.039989342751308776, epoch: 4703, iterations: 18812}
XORNetwork.ComputeOutputs([0,0]) // [0.09728578243214275]
XORNetwork.ComputeOutputs([1,0]) // [0.886647359112916]
XORNetwork.ComputeOutputs([0,1]) // [0.9042454652385032]
XORNetwork.ComputeOutputs([1,1]) // [0.09196626561919297]
```

The actual output of the demo page can be seen below.

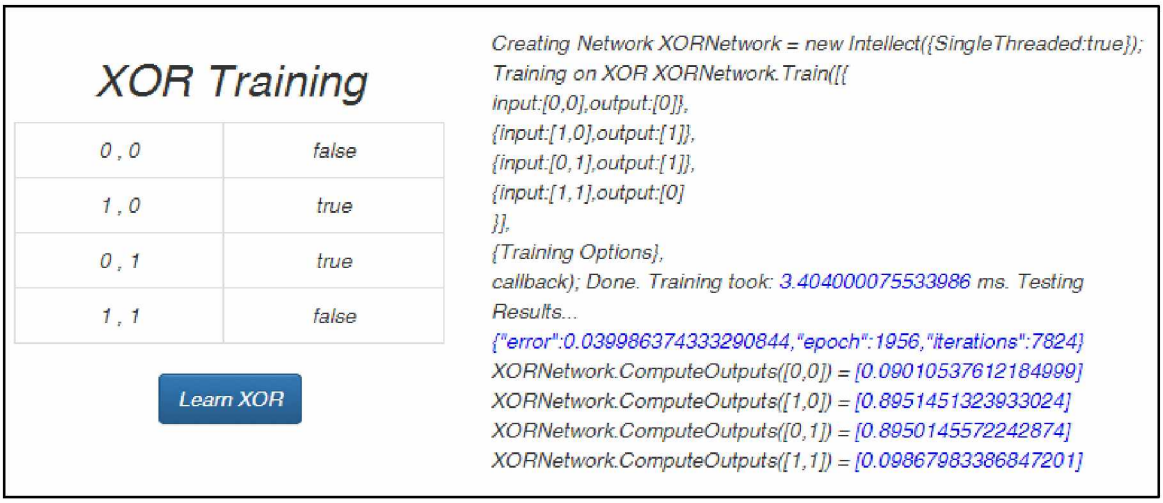


Figure 37 - Learning XOR Backpropagate Example

3.1.2 CHARACTER RECOGNITION

Character recognition is a task for which neural networks are particularly well suited. The MNIST (Mixed National Institute of Standards and Technology) database is a large database of handwritten digits from 0-9 that are commonly used for training various image-processing systems in the field of machine learning.

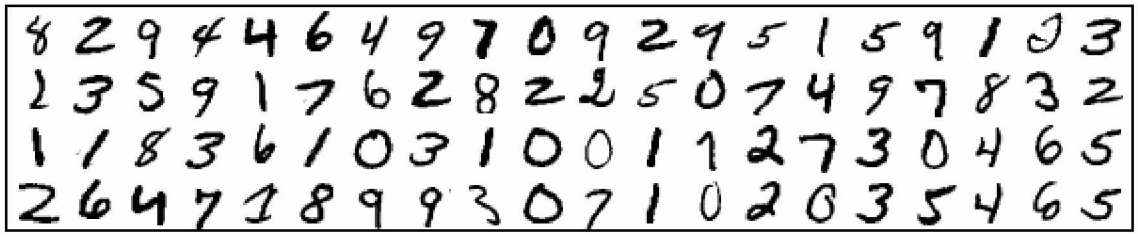


Figure 38 - Sample MNIST Images

The database contains 60,000 training images, and 10,000 testing images for measuring accuracy. As a test of the network, it was trained on 50,000 training images for 300 epochs. The trained Neural Network was able to recognize 91.2% of the characters in the MNIST testing set, as opposed to only 10% before it was trained.

Figure 39 - Learning MNIST Backpropagate Example

The MNIST images are 28 pixels wide by 28 pixels high. The network created for training on them was 784 inputs, 200 hidden neurons, and 10 output neurons. A single epoch of the backpropagation training algorithm on the 50,000 images took about 3.5 minutes, resulting an overall training time of about 17 hours.

The testing program also contained a canvas for handwriting new characters for testing the network. The network was not as good at recognizing characters drawn on the webpage as it was at recognizing the training set images. This is indicative of overfitting on the training data, and not getting the drawings on the webpage centered and scaled the same as the numbers in the training set. A possible

```
> checkAccuracy();
  Accuracy:  9192 /10000 or  91.92 %
```

Figure 40 - Accuracy of Network on MNIST Testing Set

solution to this problem would be to crop and downsample the training images and the webpage drawings before training, removing the requirement of centering and scaling an image precisely on the canvas. This would also increase the training speed of the network, as fewer inputs would be needed in order to process the images.

Another possible solution for increasing the network accuracy would be monitoring the accuracy of the network on the validation set every couple of epochs. This would greatly increase training time, but would allow for over-fitting to be monitored. If the accuracy quit improving, it would be time to stop training on the training set. If more training data is required, a common practice is to add more noise to the training images, effectively increasing the amount of training images available to the network. Adding effects such as blur, sharpening, skewing, rotation, would all provide the network with a better ability to generalize handwritten digits.

For the sake of this project, these further optimizations for MNIST recognition were not implemented, as the test was there only to demonstrate the ability of Intellect.js.

3.1.3 GENETIC ALGORITHM XOR

Intellect.js' genetic algorithm can be used with a training set, however the performance is much slower than backpropagation, and the error is considerably higher. Since the training time is much longer for the genetic algorithm, web workers are enabled for the demo in order to keep the page responsive. A population of 30 Intellects are created, and their fitness calculated by subtracting the mean squared error of their outputs from zero for the training set.

Since the entire goal of the genetic algorithm is to increase the overall fitness, the population quickly converges to always producing outputs that hover around 0.5. This not only creates a relatively small error, but it also makes it unlikely that new generations will be able to improve upon it randomly. Occasionally a new Intellect will be created with the right mutations to continue reducing the error by having the outputs diverge from 0.5.

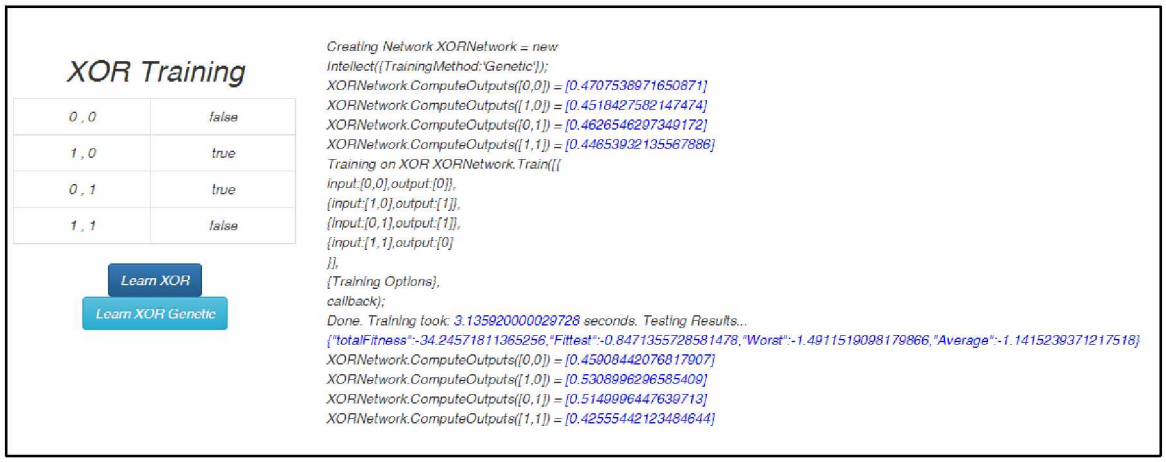


Figure 41 - Genetic Algorithm XOR Example

A good takeaway for the genetic algorithm is that the calculation of fitness plays a major role in what the Networks will evolve to do. In addition, randomness is the driving factor behind the evolution of a population; the more a population converges into the same behavior, the less likely the next generation will do something different without a good amount of random mutation.

3.2 NETWORK PERFORMANCE

To evaluate the performance Intellect.js, the main function of the network, ComputeOutputs, was compared against the same function of other libraries and implementations. In addition, the overall performance of Intellect.js was compared with other JavaScript libraries as far network creation, and computation speeds are concerned.

3.2.1 COMPARISION WITH NATIVE C++

The first benchmark compared a native C++ version of the library with an almost identical JavaScript implementation to measure how well V8's JIT compiler was able to optimize the code compared to a C++ compiler. The benchmark measured the time it took to compute the output of a large network, with 500 inputs, 200 hidden neurons, and 10 output neurons. In order to get reliable benchmarks, the C++ version was averaged across a thousand iterations, while jsPerf was used for the JavaScript versions.

The C++ code was compiled in Microsoft Visual Studio Professional 2013 in Release Mode to enable compiler optimizations. The result was that the optimized binary took about 240 microseconds to compute the output. The first implementation of the JavaScript library was a direct port of the C++ library, except for a few minor syntax and semantic changes to make it work in JavaScript. V8 was very good at optimizing during run time to get an average time of 310 microseconds to compute the output, or about 25% slower than the compiled binary.

For comparison purposes, the final version of Intellect.js was also timed. The result was that all of the optimizations and changes performed to the JavaScript code made it execute 41% faster than the compiled C++ code. If the same algorithm optimizations were applied to the C++ library, it is probable that the new compiled binary's speed would be faster than Intellect.js due to the way JavaScript is optimized during runtime.

Lastly, newer browser versions were also tested to see if performance would continue to increase as the JIT compilers improved. Chrome's alpha build of 44.0.2364 saw an increase in both Intellect.JS and the C++ to JS port. Mozilla Firefox's SpiderMonkey JavaScript engine was also benchmarked; Intellect.js saw a large performance increase while the direct C++ to JS version was slower. This shows that JavaScript performance is continually improving and that the optimizations made in Intellect.js work across different JIT compilers.

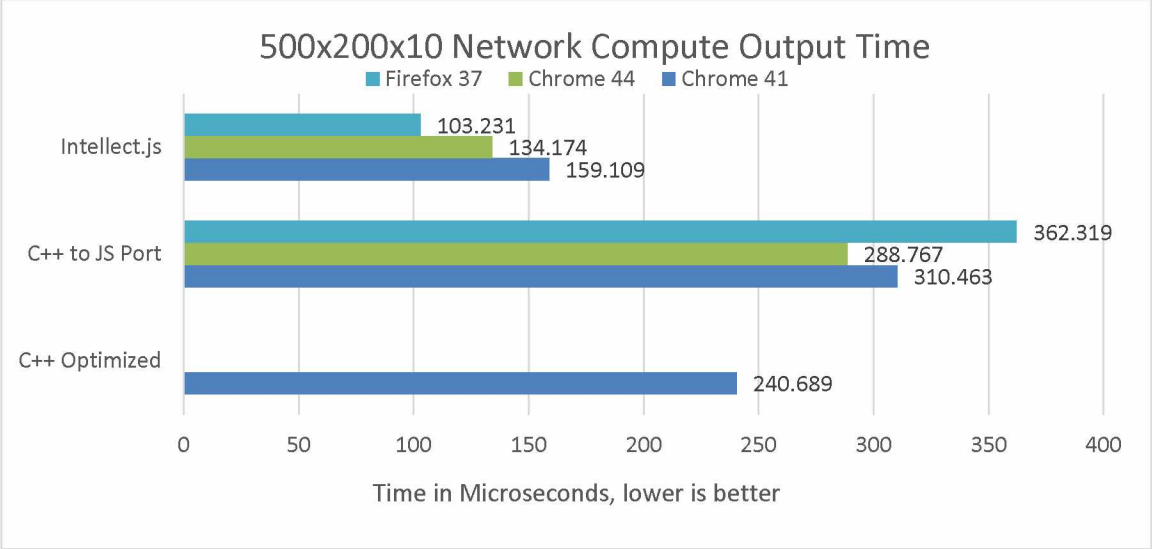


Figure 42 - JavaScript versus compiled C++ performance

3.2.2 COMPARISON WITH OTHER JAVASCRIPT LIBRARIES

In order to compare the performance of Intellect.js with the existing libraries, two series of benchmarks were performed. The first was the time it took to create a new network of 500 inputs, 200 hidden neurons, and 10 outputs. The second measured the output computation time similar to the one performed in the last section.

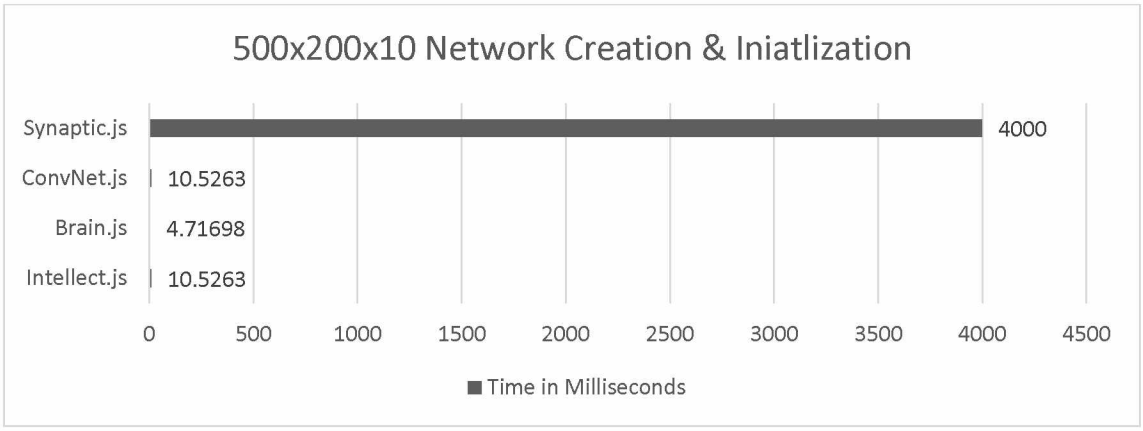


Figure 43 - JS ANN Library Comparison - Initialization

Brain.js performed the fastest at creation time. However, it does not have many options when it comes to activation functions, or training algorithms. Intellect.js and ConvNet.js were second with identical times. However, ConvNet.js's layer definitions are much more complex, allowing for any number of network configurations. Synaptic.js was the slowest, with a time averaging around 4 seconds. Synaptic also takes the most memory out of all

The next benchmark was the time required to evaluate an input.

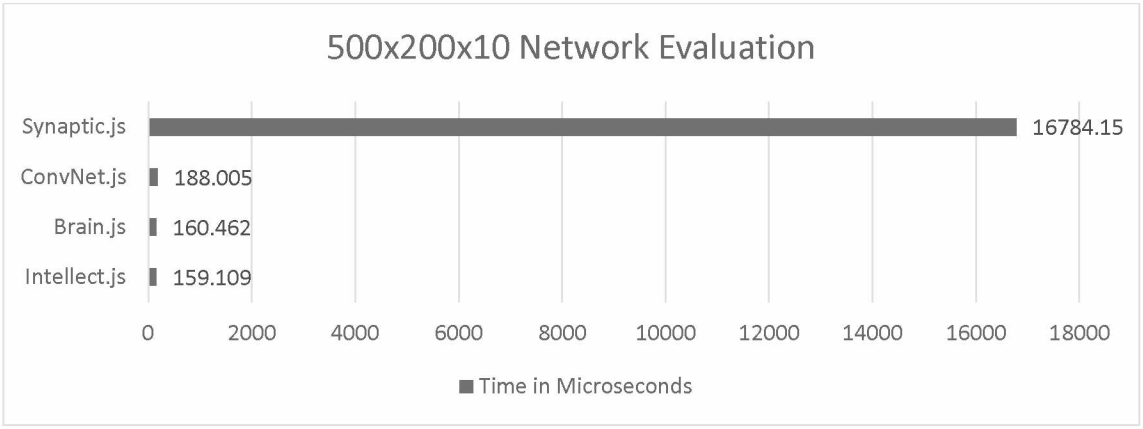


Figure 44 - JS ANN Comparison - Compute Output

Synaptic was under a second, but it was still extremely slow in comparison. Brain.js and Intellect.js were almost identical, with ConvNet.js coming in in a close third. Intellect.js mirrors Brain.js closely in the storage of Weights and Biases, and the compute outputs function, so the similarities are expected. The main difference being that Intellect.js uses an ElliotSigmoid function as compared to the standard Sigmoid function.

A training benchmark was also performed, but it only involved Synaptic, Brain.js, and Intellect.js. ConvNet.js was not included because its training methods were not standalone, meaning the iterations and error checking would have to be added manually. Doing so would not adequately reflect the library's performance as the recommended training examples for ConvNet.js have networks training continually by calling a training step using the setTimeout function in JavaScript. As part of this test, the network was both created and trained, largely because Brain.js initializes the Weights and Biases when train is called. Since each library uses its own cost function, maximum epochs, and different error thresholds for XOR, the results are inconclusive. Synaptic.js actually performed quite well for this test, implying that the compute output time can be improved by whatever caching method its training algorithm uses.

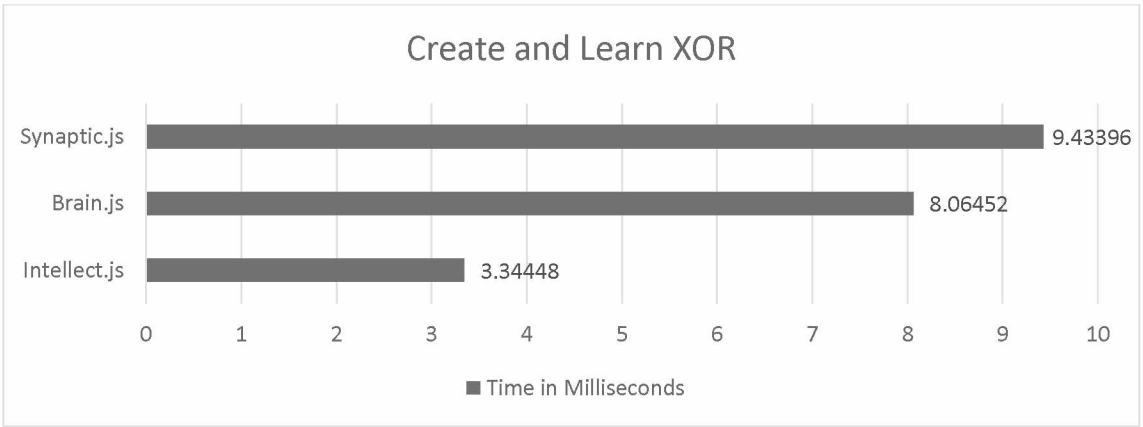


Figure 45 - JS ANN Comparison - Learning XOR

Some of the major difference between Intellect.js and Brain.js is the optimized weight initialization, the activation function, and some other optimizations to the backpropagation algorithm. These might account for the speed up visible in Intellect.js when compared with the other libraries. However, another reason these results are inconclusive is that the training time can be shorter or longer depending on the initial weights and biases of the network. Since each library was initialized with random weights and biases, the amount of training each network needed to do in order to meet its error threshold was different.

4. CONCLUSION AND FUTURE WORK

Artificial Neural Networks in JavaScript are very useful. The Just In Time compilers are able to optimize the code efficiently so that the execution times are comparable to compiled C++ binaries. It is unlikely that JavaScript will ever be faster than optimized C++ code, as both would run up against the same limits of the physical hardware upon which they run. However, if considering build time as part of the execution, then JavaScript will always be faster.

Intellect.js is comparable to the other Neural Network libraries available in JavaScript, and has some key features that are missing in the others. However, some libraries are more robust such as ConvNet.js and Synaptic. This downside can easily be fixed by continuing to build up Intellect.js, and improving its optimizations. The Intellect.js code is also much smaller and more lightweight than the other libraries, coming in at only 9KB minified, while Brain.js is about 63KB, and ConvNet.js and Synaptic.js are both 32KB.

Intellect.js is available under the MIT License online at
https://github.com/sutekidayo/intellect.js

Future work for Intellect.js involves adding more training optimizations for the backpropagation algorithm, namely AdaDelta, as it removes the need for picking good hyper-parameters like learning rate and momentum. In addition, adding hybrid-training methods that use backpropagation and simulated annealing would also improve performance on prediction-based data sets.

The portability and performance of JavaScript opens the doors to many possible applications of ANNs in the browser, such as distributed networks the size of the internet, the ability to train on live data sources instead of compiled training sets, and being able to solve complex problems directly in the browser. The possibilities are endless.

BIBLIOGRAPHY

[1] Warren S McCulloh and Walter Pitts, "A logicial calculus of the ideas immanent in nervous activity," *The bulletin of mathematical biophysics 5.4*, pp. 115-133, 1943.

[2] Donald Hebb, *The Organization of Behaviour: A Neuropsychological Theory*. New York: Wiley, 1949.

[3] B G Farley and Wesley A Clark, "Simulation of self-organizing systems by digital computer," *Information Theory, Transactions of the IRE Professional Group on*, vol. 4, no. 4, pp. 76-84, 1954.

[4] Frank Rosenblatt, "The perceptron: a probabilistic model for information storage and organization in the brain.," *Pysochological review*, vol. 65, no. 6, p. 386, 1958.

[5] Marvin Minksy and Seymour Papert, *Perceptron: an introduction to computational geometry*. Cambridge: The MIT Press, 1969.

[6] Paul J Werbos, "Beyond Regression: New Tools for Prediction and Analysis in the Behavioral Sciences," 1974.

[7] Matthew D Zeiler, "Adadelta: An Adaptive Learning Rate Method," *arXiv preprint*, vol. 1212, no. 5701, 2012.

[8] Jeffrey Dean et al., "Large Scale Distributed Deep Networks," *Advances in Neural Information Processing Systems*, vol. 25, pp. 1223-1231, 2012.

[9] David B Fogel, *Blondie24: Playing at the Edge of AI.*: Morgan Kaufmann, 2001.

[10] Andrej Karpathy. ConvNetJS: Deep Learning in your browser. [Online]. http://cs.stanford.edu/people/karpathy/convnetjs/index.html

[11] Juan Cazala. Synaptic - The javascript neural network library. [Online]. http://synaptic.juancazala.com/

[12] Heather Arthur. harthur/brain - GitHub. [Online]. https://github.com/harthur/brain

[13] Nikki Moreaux. Einstein.js. [Online]. https://github.com/nikkimoreaux/Einstein.js

[14] Mozilla Corporation. Introduction to Object-Oriented Javascript. [Online]. https://developer.mozilla.org/en-US/docs/Web/JavaScript/Introduction_to_Object-Oriented_JavaScript

[15] Marja Hölttä. (2013, Aug) Crankshafting from the ground up. [Online]. https://docs.google.com/document/d/1hOaE7vbwdLLXWj3C8hTnnkpE0qSa2P--dtDvwXXEeD0/pub

[16] John-David Dalton and Mathias Bynens. Benchmark.js. [Online]. http://benchmarkjs.com/

[17] David L Elliot, "A Better Activation Function for Artificial Neural Networks," 1993.

[18] W3C. (2014, Oct) HTML5 A vocabulary and associated APIs for HTML and XHTML W3C Recommendation. [Online]. http://www.w3.org/TR/html/infrastructure.html#transferable-objects

[19] Jeff Heaton, *Introduction to Neural Networks for C#*. Chesterfield: Heaton Research, Inc, 2008.

[20] Michael Nielsen, "Weight Initialization," in *Neural networks and Deep Learning*., 2014.

[21] Mozilla Corporation. (2015, March) Web Workers API. Web Page. [Online]. https://developer.mozilla.org/en-US/docs/Web/API/Web_Workers_API/

[22] Travis M Payton. (2015, January) Preallocating Array - jsPerf. [Online]. http://jsperf.com/preallocating-array/9

APPENDIX A – CODE LISTING

INTELLECT.JS

```
/**
 * Created by tpayton1 on 11/7/2014.
 */
(function (global) {
    var is_worker = !this.document;

    var script_path = is_worker ? null : (function () {
        // append random number and time to ID
        var id = (Math.random() + '' + (+new Date)).substring(2);
        document.write('<script id="wts' + id + '"></script>');
        return document.getElementById('wts' + id).
            previousSibling.src;
    })();

    function msgFromParent(e) {
        // event handler for parent -> worker messages
        switch (e.data.command) {
            case "Train":
                workerIntellect = new Intellect(e.data.options);
                workerIntellect.Weights = e.data.weights;
                workerIntellect.Biases = e.data.biases;
                var result = workerIntellect.Train(e.data.data,
e.data.trainingOptions);
                self.postMessage({
                    command: "Trained",
                    result: result,
                    weights: workerIntellect.Weights,
                    biases: workerIntellect.Biases
                });

                self.close();
                break;
            case "Stop":
                self.postMessage({
                    command: "Trained",
                    result: result,
                    weights: workerIntellect.Weights,
                    biases: workerIntellect.Biases
                });
                self.close();
                break;
            default:
                console.log("Worker received", e.data);
                break;
        }
    }

    function msgFromWorker(e) {
        // event handler for worker -> parent messages
        switch (e.data.command) {
            case "Trained":
                this.prototype._this.Weights = e.data.weights;
                this.prototype._this.Biases = e.data.biases;
```

```
            this.prototype._this.Trained = true;
            this.prototype.callback(e.data.result);
            break;
        case "TrainingCallback":
            this.prototype.trainingCallback(e.data.result);
            break;
        default:
            console.log("Message from Worker", e.data);

    }
}

function new_worker() {
    var w = new Worker(script_path);
    w.addEventListener('message', msgFromWorker, false);
    return w;
}

if (is_worker) {
    var workerIntellect; // Holds the network for worker threads
    global.addEventListener('message', msgFromParent, false);
}
// Main Intellect Library Starts Here.
var Intellect = function (options) {
    var defaultOptions = {
        Layers: [2, 2, 1],

        // Possible Options:
        // "BackPropagate" = Gradient Descent
        // "Genetic" = Evolutionary Genetic Algorithm
        // or provide your own function
        TrainingMethod: "BackPropagate",

        // Possible Options:
        // "ElliotSigmoid" = .5*x/(1+|x|)+.5
        // "ElliotSymmetrical" = x/(1+|x|)
        // "Sigmoid" = 1/(1+e^-x)
        // "HyperTan" = tanh(x)
        // or provide your own function
        ActivationFunction: "ElliotSigmoid",
        ActivationFunctionDerivative:"Sigmoid",

        // Possible Options:
        // "MeanSquaredError"
        // "QuadradicCost"
        // "CrossEntropy"
        // or provide your own function
        ErrorFunction: "MeanSquaredError",

        // Disable WebWorkers (only used for training)
        SingleThreaded: false,

        // Create a hidden layer if one is not provided
        AutomaticHiddenLayer: true
    };

    if (!is_worker) {
        if (window == this) {
```

```
                // Prevent Cluttering the Window Scope;
                return new Intellect(options);
            }

        }


        var extend = function () {
            for (var i = 1; i < arguments.length; i++)
                for (var key in arguments[i])
                    if (arguments[i].hasOwnProperty(key))
                        arguments[0][key] = arguments[i][key];
            return arguments[0];
        };

        // Merge defaultOptions with any provided options
        this.options = extend({}, defaultOptions, options);
        this.SingleThreaded = this.options.SingleThreaded;


        this.Initialize(this.options.Layers);


    };

    Intellect.prototype = {

        // Helper Functions

        // returns a random number from a standard distribution (used for initializing
Biases for optimal learning speed)
        // Quicker than Box-Muller transform, credit goes to Protonfish
http://www.protonfish.com/random.html
        rand_normal: function () {
            return (Math.random() * 2 - 1) + (Math.random() * 2 - 1) + (Math.random() *
2 - 1);
        },


        // In place Fisher-Yates shuffle.
        shuffle: function (array) {
            var m = array.length, t, i;
            // While there remain elements to shuffle...
            while (m) {
                // Pick a remaining element...
                i = Math.floor(Math.random() * m--);
                // And swap it with the current element.
                t = array[m];
                array[m] = array[i];
                array[i] = t;
            }
            return array;
        },


        randWeights: function (size) {
            var weights = new Array(size);
            for (var i = 0; i < size; i++) {
```

```
            weights[i] = this.rand_normal() / Math.sqrt(size);
        }
        return weights;
    },

    randBias: function (size) {
        var biases = new Array(size);
        for (var i = 0; i < size; i++) {
            biases[i] = this.rand_normal();
        }
        return biases;
    },

    zeros: function (size) {
        var arr = new Array(size);
        for (var i = 0; i < size; i++) {
            arr[i] = 0.0;
        }
        return arr;
    },


    // Activation Functions http://jsperf.com/neural-net-activation-functions/2
    ActivationFunctions: {
        Sigmoid: function (x) {
            return 1.0 / (1.0 + Math.exp(-x));
        },
        HyperTan: function (x) {
            return Math.tanh(x);
        },
        ElliotSigmoid: function (x) {
            return (.5 * x / (1 + Math.abs(x))) + .5;
        },
        ElliotSymmetrical: function (x) {
            return x / (1 + Math.abs(x));
        }

    },

    // Partial Derivatives of Activation Functions for BackPropagation Training
Method
    ActivationFunctionDerivatives: {
        Sigmoid: function (output, input) {
            return (1 - output) * output;
        },
        HyperTan: function (output, input) {
            return (1 - output) * (1 + output);
        },
        ElliotSigmoid: function (output, input) {
            return 1 / ((1 + Math.abs(input)) * (1 + Math.abs(input)));
        },
        ElliotSymmetrical: function (output, input) {
            return (.5 / (1 + Math.abs(input) * (1 + Math.abs(input))));
        }
    },

    ErrorFunctions: {
        MeanSquaredError: {
```

```
                error: function (expected, actual) {
                    var sum = 0;
                    for (var i = 0; i < expected.length; i++) {
                        var error = actual[i] - expected[i]
                        sum += (error * error);
                    }
                    return sum / expected.length;
                },
                delta: function (expected, actual, gradient) {
                    return (expected - actual);
                }
            },
            Quadratic: {
                error: function (expected, actual) {
                    // mean squared error
                    var sum = 0;
                    for (var i = 0; i < expected.length; i++) {
                        var error = actual[i] - expected[i]
                        sum += (error * error)
                    }
                    return .5 * sum;
                },
                delta: function (expected, actual, gradient) {
                    return (expected - actual) * gradient;
                }

            },
            CrossEntropy: {
                error: function (expected, actual) {
                    var sum = 0;
                    for (var i = 0; i < expected.length; i++) {
                        var error = -expected[i] * Math.log(actual[i]) - (1 -
expected[i]) * Math.log(1 - actual[i]);
                        sum += isFinite(error) ? error : 0;
                    }
                    return sum / expected.length;
                },
                delta: function (expected, actual) {
                    return (expected - actual);
                }
            }

        },

        // Training Methods
        // Some require the Activation Functions so this is initialized after they are
pointing correctly.
        TrainingMethods: {
            // BackPropagate
            // This updates the weights of the NN using the backpropagation technique
            // Takes an array of expected outputs, eta: amount to change,
alpha:momentum based on previous weight delta
            BackPropagate: function (data, options, callback) {
                var batchSize = options.batchSize || 1;
                var epochs = options.epochs || 20000;
                var shuffle = options.shuffle || false;
                var errorThresh = options.errorThresh || 0.04;
                var log = options.log || false;
```

```
                var logPeriod = options.logPeriod || 500;
                options.learningRate = options.learningRate || .45;
                options.momentum = options.momentum || .3;
                var callbackPeriod = options.callbackPeriod || 0;
                var dataLength = data.length;
                //options.learningRate /= dataLength;

                // Create a WebWorker to handle the training
                if (!is_worker && !this.SingleThreaded) {
                    this.worker = new_worker();
                    var IntellectThis = this;
                    this.worker.prototype = {
                        _this: IntellectThis,
                        Result: false,
                        trainingCallback: options.callback,
                        callback: callback
                    };
                    options.callback = true;
                    this.worker.postMessage({
                        command: "Train",
                        options: this.options,
                        weights: this.Weights,
                        biases: this.Biases,
                        data: data,
                        trainingOptions: options
                    });
                }
                else {
                    var error = 9999;
                    var preverror = 99999999;
                    for (var epoch = 0; epoch < epochs && error > errorThresh; epoch++)
{
                        var sum = 0;
                        if (shuffle) {this.shuffle(data);}
                        for (var n = 0; n < dataLength; n += batchSize) {
                           // for (var batch = 0; batch < batchSize && n + batch <=
dataLength; batch++) {
                                // Feed Forward
                                var actual = this.ComputeOutputs(data[n].input);
                                // Compute the Deltas & Errors for this batch
                                var expected = data[n].output;
                                sum += this.ErrorFunction.error(expected, actual,
dataLength);
                                this.CalculateDeltas(expected);
                            //}
                            // Update Weights and Biases
                            this.UpdateWeights(options);
                        }

                        error = sum;
                        if (error > preverror){
                            options.learningRate *= .25;
                        }
                        preverror = error;

                        if (log && (epoch % logPeriod == 0)) {
                            console.log("epoch:", epoch, "training error:", error);
                        }
```

```
                        if (callbackPeriod && epoch % callbackPeriod == 0) {
                            if (is_worker) {
                                self.postMessage({
                                    command: "TrainingCallback",
                                    result: {epoch: epoch, error: error, learningRate:
options.learningRate}});
                            }
                            else {
                                options.callback({epoch: epoch, error: error})
                            }
                        }
                    }
                    var result = {
                        error: error,
                        epoch: epoch,
                        iterations: dataLength * epoch
                    };
                    callback(result);
                    return result;
                }
            },

            Genetic: function (data, options,callback) {
                // performs a single epoch
                var popSize = options.popSize || 5;
                var epochs = options.epochs || 100;
                var log = options.log || false;
                var logPeriod = options.logPeriod || 500;
                options.learningRate = options.learningRate || 0.5;
                var callback = options.callback;
                var callbackPeriod = options.callbackPeriod || 100;
                var fitnessFunction = options.testFitnessFunction ||
this.testFitnessFunction;
                var population = [];



                // Create the population
                for (var intellect = 0; intellect < popSize; intellect++) {
                    population.push(new Intellect(this.options));
                }

                var fitnesses = fitnessFunction(population, data);
                var generationInfo = this.CalculateFitnesses(fitnesses)
                // fitnesses are now sorted by CalculateFitnesses function
                // perform a geneticEpoch
                var epoch = 0;
                var _this = this;
                var nextEpoch = function(population){
                    epoch++;
                    if (epoch < epochs){
                        if (log && (epoch % logPeriod == 0)) {

                            console.log("epoch:", epoch, " Best Fitness:",
generationInfo.Fittest, ", Average Fitness:", generationInfo.Average, "Worst
Fitness:",generationInfo.Worst);
                        }
```

```
                    if (callback && i % callbackPeriod == 0) {
                        callback({
                            epoch: epoch,
                            fittest: fitnesses.Fittest,
                            worst: fitnesses.Worst,
                            average: fitnesses.Average
                        });
                    }
                    fitnesses = fitnessFunction(population, data);
                    generationInfo = _this.CalculateFitnesses(fitnesses);
                    _this.geneticEpoch(population,fitnesses,nextEpoch);

                }
                else {
                    this.Weights = population[fitnesses[0].index].Weights;
                    this.Biases = population[fitnesses[0].index].Biases;
                }
            };

            // perform the genetic algorithm!
            this.geneticEpoch(population,fitnesses,nextEpoch);




        }

    },

    CalculateDeltas: function (tValues) {
        // Calculate Deltas, must be done backwards
        for (var layer = this.outputLayer; layer >= 0; layer--) {
            for (var node = 0; node < this.Layers[layer]; node++) {
                var output = this.Outputs[layer][node];
                var input = this.Sums[layer][node];
                var error = 0;
                if (layer == this.outputLayer) {
                    error = output - tValues[node];
                }
                else {
                    var deltas = this.Deltas[layer + 1];
                    for (var k = 0; k < deltas.length; k++) {
                        error += deltas[k] * this.Weights[layer + 1][k][node];
                    }
                }
                this.Deltas[layer][node] = error *
this.ActivationFunctionDerivative(output, input);
            }
        }
    },
    UpdateWeights: function (options) {
        // Adjust Weights
        for (var layer = 1; layer <= this.outputLayer; layer++) {
            var incoming = this.Outputs[layer - 1];
            for (var node = 0, nodes = this.Layers[layer]; node < nodes; node++) {
```

```
                    var delta = this.Deltas[layer][node];
                    for (var k = 0, length = incoming.length; k < length; k++) {
                        var prevWeightDelta = (options.learningRate * delta *
incoming[k]) + (options.momentum * this.prevWeightsDelta[layer][node][k]);
                        this.prevWeightsDelta[layer][node][k] = prevWeightDelta;
                        this.Weights[layer][node][k] -= prevWeightDelta;
                    }
                    this.Biases[layer][node] -= options.learningRate * delta;
                }
            }

        },

        testFitnessFunction: function (data, population) {
            var fitnesses = new Array(population);
            // Use the training data to determine fitness
            for (var i = 0, popSize = population.length; i < popSize; i++) {
                var fitness = 0;
                for (var j = 0, length = data.length; j < length; j++) {
                    fitness -= this.ErrorFunction(data[j].output,
population[i].ComputeOutputs(data[j].input))
                }
                fitnesses[i]=({index: i, Fitness: fitness})
            }
            return fitnesses;

        },
        // Sorts Population in Descending Order based on Fitness
        FitnessSort: function (a, b) {
            return b.Fitness - a.Fitness;
        },

        CalculateFitnesses: function (fitnesses) {
            var totalFitness = 0;
            fitnesses.sort(this.FitnessSort);
            for (var i = 0, popSize = fitnesses.length; i < popSize; i++) {
                totalFitness += fitnesses[i].Fitness;
            }
            return {totalFitness: totalFitness, Fittest: fitnesses[popSize -
1].Fitness, Worst: fitnesses[0].Fitness, Average: totalFitness / fitnesses.length}
        },

        geneticEpoch: function(population,fitnesses,callback){


        },


        Trained: true,

        stop: function () {
            if (this.worker) {
                this.worker.postMessage({command: "Stop"});
            }
        },

        Train: function (data, options, callback) {
            options = options || {};
```

```
            options.callback = options.callback || function () {};
            callback = callback || function () {};
            this.Trained = false; // Used to know when a worker thread is done training
            var result = this.TrainingMethod(data, options,callback);
            return result;
        },


        // Functions
        ComputeOutputs: function (input) {

            this.Outputs[0] = input;
            // Compute the Outputs of each Layer
            for (var layer = 1; layer <= this.outputLayer; layer++) {
                for (var node = 0, nodes = this.Layers[layer]; node < nodes; node++) {
                    var weights = this.Weights[layer][node];
                    var sum = this.Biases[layer][node];
                    for (var k = 0, length = weights.length; k < length; k++) {
                        sum += weights[k] * input[k];
                    }
                    this.Sums[layer][node] = sum;
                    this.Outputs[layer][node] = this.ActivationFunction(sum);
                }
                var output = input = this.Outputs[layer];
            }

            return output;

        },


        Initialize: function (NetworkConfiguration) {
            if (this.options.Layers.length == 2 && this.options.AutomaticHiddenLayer) {
                // Add a hidden layer using the rule that it should 2/3 thei size of
the input plus the size of the output
                Math.floor((2 / 3) * this.options.Layers[0] + this.options.Layers[1])

            }
            // Check to see if Workers are supported
            if (!is_worker && !this.SingleThreaded) {
                if (!window.Worker) {
                    this.SingleThreaded = true;
                }
                else{
                    this.cores = navigator.hardwareConcurrency;
                    this.cores = this.cores || 1; // can be improved with a polyfill to
estimate
                }
            }
            // Check to see if a function was provided, if not update the string to the
actual function
            if (typeof(this.options.ActivationFunction) !== 'function') {
                this.ActivationFunction =
this.ActivationFunctions[this.options.ActivationFunction];
            }
            else {
                this.SingleThreaded = true;
                this.ActivationFunction = this.options.ActivationFunction;
```

```
            }
            this.options.ActivationFunctionDerivative =
this.options.ActivationFunctionDerivative || this.options.ActivationFunction;
            if (typeof(this.options.ActivationFunctionDerivative) !== 'function') {
                this.ActivationFunctionDerivative =
this.ActivationFunctionDerivatives[this.options.ActivationFunctionDerivative];
            }
            else {
                this.SingleThreaded = true;
                this.ActivationFunctionDerivative =
this.options.ActivationFunctionDerivative;
            }


            if (typeof(this.options.TrainingMethod) !== 'function') {
                this.TrainingMethod =
this.TrainingMethods[this.options.TrainingMethod];
            }
            else {
                this.TrainingMethod = this.options.TrainingMethod;
            }

            if (typeof(this.options.ErrorFunction) !== 'function') {
                this.ErrorFunction = this.ErrorFunctions[this.options.ErrorFunction];
            }
            else {
                this.SingleThreaded = true;
                this.ErrorFunction = this.options.ErrorFunction;
            }


            // Setup Network Matrices
            // Sums, Biases, Weights, and Outputs, Deltas, Deltas
            this.Fitness = 0;
            this.Layers = NetworkConfiguration;
            this.outputLayer = this.Layers.length - 1;
            this.Sums = []; // input to node output function
            this.Biases = [];
            this.Weights = [];
            this.Outputs = [];
            this.prevWeightsDelta = [];
            this.prevBiasesDelta = [];
            this.Deltas = [];

            for (var layer = 0; layer <= this.outputLayer; layer++) {
                var size = this.Layers[layer];
                this.prevBiasesDelta[layer] = this.zeros(size);
                this.Outputs[layer] = this.zeros(size);
                this.Deltas[layer] = this.zeros(size);
                this.Sums[layer] = this.zeros(size);

                if (layer > 0) {
                    this.Biases[layer] = this.randBias(size);
                    this.Weights[layer] = new Array(size);
                    this.prevWeightsDelta[layer] = new Array(size);

                    for (var node = 0; node < size; node++) {
                        var prevSize = this.Layers[layer - 1];
```

```
                        this.Weights[layer][node] = this.randWeights(prevSize);
                        this.prevWeightsDelta[layer][node] = this.zeros(prevSize);
                    }
                }
            }
        },

        saveNetwork: function () {
            var data = {options: this.options, weights: this.Weights, biases:
this.Biases};
            var fileName = "Network - " + new Date() + ".json";
            var a = document.createElement("a");
            document.body.appendChild(a);
            a.style = "display: none";

            var json = JSON.stringify(data),
                blob = new Blob([json], {type: "octet/stream"}),
                url = window.URL.createObjectURL(blob);
            a.href = url;
            a.download = fileName;
            a.click();
            window.URL.revokeObjectURL(url);

        },

        loadNetwork: function (data) {
            data = JSON.parse(data);
            this.options = data.options;
            this.Initialize(this.options.Layers);
            this.Biases = data.biases;
            this.Weights = data.weights;
        }


    }; // end Intellect prototype declaration

    if (!is_worker) {
        window.Intellect = Intellect;
    }
})(this);
```

EXAMPLES / DEMO INDEX.HTML

```
<!doctype html>
<html lang="en">
<head>
    <meta charset="utf-8">
    <meta http-equiv="X-UA-Compatible" content="IE=edge,chrome=1">
    <meta name="viewport" content="width=device-width, initial-scale=1">
    <title>Javascript Neural Network</title>
    <link href="CSS/bootstrap.min.css" rel="stylesheet"/>
    <link href="CSS/bootstrap-theme.min.css" rel="stylesheet"/>
</head>
<body>
<div class="container-fluid">
    <div class="jumbotron">
        <h1 class="text-center">Neural Network Demo</h1>
    </div>
    <div class="row">

        <div class="col-sm-2 col-xs-offset-0 text-center">
            <h2>XOR Training</h2>
            <table class="table table-bordered">
                <tr>
                    <td>0 , 0</td>
                    <td>false</td>

                </tr>
                <tr>
                    <td>1 , 0</td>
                    <td>true</td>
                </tr>
                <tr>
                    <td>0 , 1</td>
                    <td>true</td>
                </tr>
                <tr>
                    <td>1 , 1</td>
                    <td>false</td>
                </tr>
            </table>
            <button class="btn btn-primary" id="learnXOR">Learn XOR</button>
        </div>
        <div class="col-lg-3" id="xorLogger">
        </div>
        <div class="col-lg-3 col-sm-offset-0">
            <h1>MNIST Training</h1>

            <label for="training_images">Load Images & Labels</label>
            <input type="file" id="training_images" name="files[]" multiple/>

            <label for="mnistIndex">Digit to Check: </label>
            <input type="number" id="mnistIndex" class="input-sm"/>
            <button class="btn btn-info" id="checkDigit">Check Digit</button>
            <label for="digitHidden">Hidden Layers:</label>
            <input type="number" id="digitHidden" class="input-sm" value="100"/><br>
            <label for="trainingSize">Training Data Size:</label>
            <input type="number" id="trainingSize" class="input-sm" value="1000"/><br>
            <label for="batchSize">Batch Size:</label>
```

```
            <input type="number" id="batchSize" class="input-sm" value="1"/><br>
            <label for="logPeriod">Log Period:</label>
            <input type="number" id="logPeriod" class="input-sm" value="1"/><br>
            <label for="learningRate">Learning Rate:</label>
            <input type="number" id="learningRate" class="input-sm" value=".5"/><br>
            <label for="momentum">Momentum:</label>
            <input type="number" id="momentum" class="input-sm" value=".1"/><br>
            <label for="epochs">Epochs:</label>
            <input type="number" id="epochs" class="input-sm" value="30"/><br>
            <label for="errorThreshold">Error Threshold:</label>
            <input type="number" id="errorThreshold" class="input-sm" value=".15"/><br>
            <button class="btn btn-primary" id="learnMNIST"><i class="glyphicon
glyphicon-education"></i>Learn MNIST
            </button>
            <br>
            <button class="btn btn-primary" id="saveNetwork"><i class="glyphicon
glyphicon-save-file"></i> Save Network
            </button>

            <button class="btn btn-primary" id="loadNetworkButton"><i class="glyphicon
glyphicon-open-file"></i> Load
                Network
            </button>

            <input id="loadNetwork" type="file" style="display:none;"/>

            <h2>Hand Drawing Test</h2>
            <button class="btn btn-default" id="clear"><i class="glyphicon glyphicon-
erase"></i> Clear</button>
            <canvas id="digitDrawer" width="140" height="140" style="border: 1px solid
#000;"></canvas>
            <button class="btn btn-primary" id="checkDrawing">Calculate Number</button>
        </div>
        <div class="col-lg-3" id="MNISTLogger">

        </div>


    </div>
    <div>

    </div>

</div>
<script src="JS/Intellect.js"></script>
<script src="JS/jquery-2.1.3.min.js"></script>
<script src="JS/bootstrap.min.js"></script>
<script>

    // Keep everything in anonymous function, called on window load.
    if (window.addEventListener) {
        window.addEventListener('load', function () {
            var canvas, context, tool;

            function init() {
                // Find the canvas element.
                canvas = document.getElementById('digitDrawer');
                if (!canvas) {
```

```
            alert('Error: I cannot find the canvas element!');
            return;
        }

        if (!canvas.getContext) {
            alert('Error: no canvas.getContext!');
            return;
        }

        // Get the 2D canvas context.
        context = canvas.getContext('2d');
        if (!context) {
            alert('Error: failed to getContext!');
            return;
        }

        // Pencil tool instance.
        tool = new tool_pencil();

        // Attach the mousedown, mousemove and mouseup event listeners.
        canvas.addEventListener('mousedown', ev_canvas, false);
        canvas.addEventListener('mousemove', ev_canvas, false);
        canvas.addEventListener('mouseup', ev_canvas, false);
    }

    // This painting tool works like a drawing pencil which tracks the mouse
    // movements.
    function tool_pencil() {
        var tool = this;
        this.started = false;

        // This is called when you start holding down the mouse button.
        // This starts the pencil drawing.
        this.mousedown = function (ev) {
            context.beginPath();
            context.moveTo(ev._x, ev._y);
            tool.started = true;
        };

        // This function is called every time you move the mouse. Obviously, it
only
        // draws if the tool.started state is set to true (when you are holding

down
        // the mouse button).
        this.mousemove = function (ev) {
            if (tool.started) {
                context.lineTo(ev._x, ev._y);
                context.lineWidth = 10;
                context.stroke();
            }
        };

        // This is called when you release the mouse button.
        this.mouseup = function (ev) {
            if (tool.started) {
                tool.mousemove(ev);
                tool.started = false;
            }
```

```
            };
        }

        // The general-purpose event handler. This function just determines the
mouse
        // position relative to the canvas element.
        function ev_canvas(ev) {
            if (ev.layerX || ev.layerX == 0) { // Firefox
                ev._x = ev.layerX;
                ev._y = ev.layerY;
            } else if (ev.offsetX || ev.offsetX == 0) { // Opera
                ev._x = ev.offsetX;
                ev._y = ev.offsetY;
            }

            // Call the event handler of the tool.
            var func = tool[ev.type];
            if (func) {
                func(ev);
            }
        }

        init();

    }, false);
}
</script>
<script>
    var labels;
    var images;
    var imagesBuffer;
    var width;
    var height;
    var currentLog = null;
    var MNISTNet = null;
    var stop = false;
    var MNISTOptions = {
        Layers: [28 * 28, 100, 10],
        TrainingMethod: "BackPropagate", // Possible Options: "BackPropagate" =
Recursive Descent to minimize Cost Function, "Genetic" = A Genetic Mutation based on
criteria, or provide your own function
        ActivationFunction: "ElliotSigmoid",
        ActivationFunctionDerivative: "Sigmoid",
        ErrorFunction: "MeanSquaredError" // Possible Options: "MeanSquaredError",
"QuadradicCost", and "CrossEntropy"

    };

    var BuildBridgedWorker = function (workerFunction, workerExportNames,
mainExportNames, mainExportHandles) {

        var baseWorkerStr =
workerFunction.toString().match(/^\s*function\s*\(\s*\)\s*\{([\s\S](?!\}$))*[\s\S])/)[
1];
        var extraWorkerStr = [];

        // build a string for the worker end of the worker-calls-funciton-in-main-
thread operation
```

```
        extraWorkerStr.push("var main = {};\n");
        for (var i = 0; i < mainExportNames.length; i++) {
            var name = mainExportNames[i];
            if (name.charAt(name.length - 1) == "*") {
                name = name.substr(0, name.length - 1);
                mainExportNames[i] = name;//we need this trimmed version back in main
                extraWorkerStr.push("main." + name + " = function(/* arguments */){\n
var args = Array.prototype.slice.call(arguments); var buffers = args.pop(); \n
self.postMessage({foo:'" + name + "', args:args},buffers)\n}; \n");
            } else {
                extraWorkerStr.push("main." + name + " = function(/* arguments */){\n
var args = Array.prototype.slice.call(arguments); \n self.postMessage({foo:'" + name +
"', args:args})\n}; \n");
            }
        }

        // build a string for the worker end of the main-thread-calls-function-in-
worker operation
        var tmpStr = [];
        for (var i = 0; i < workerExportNames.length; i++) {
            var name = workerExportNames[i];
            name = name.charAt(name.length - 1) == "*" ? name.substr(0, name.length -
1) : name;
            tmpStr.push(name + ": " + name);
        }
        extraWorkerStr.push("var foos={" + tmpStr.join(",") + "};\n");
        extraWorkerStr.push("self.onmessage = function(e){\n");
        extraWorkerStr.push("if(e.data.foo in foos) \n  foos[e.data.foo].apply(null,
e.data.args); \n else \n throw(new Error('Main thread requested function ' + e.data.foo
+ '. But it is not available.'));\n");
        extraWorkerStr.push("\n};\n");

        var fullWorkerStr = baseWorkerStr + "\n\n/*==== STUFF ADDED BY
BuildBridgeWorker ==== */\n\n" + extraWorkerStr.join("");

        // create the worker
        var url = window.URL.createObjectURL(new Blob([fullWorkerStr], {type:
'text/javascript'}));
        var theWorker = new Worker(url);

        // buid a funcion for the main part of worker-calls-function-in-main-thread
operation
        theWorker.onmessage = function (e) {
            var fooInd = mainExportNames.indexOf(e.data.foo);
            if (fooInd != -1)
                mainExportHandles[fooInd].apply(null, e.data.args);
            else
                throw(new Error("Worker requested function " + e.data.foo + ". But it
is not available."));
        }

        // build an array of functions for the main part of main-thread-calls-function-
in-worker operation
        var ret = {blobURL: url};//this is useful to know for debugging if you have
loads of bridged workers in blobs with random names
        var makePostMessageForFunction = function (name, hasBuffers) {
            if (hasBuffers)
                return function (/*args...,[ArrayBuffer,..]*/) {
```

```
                    var args = Array.prototype.slice.call(arguments);
                    var buffers = args.pop();
                    theWorker.postMessage({foo: name, args: args}, buffers);
                }
            else
                return function (/*args...*/) {
                    var args = Array.prototype.slice.call(arguments);
                    theWorker.postMessage({foo: name, args: args});
                };
        }

        for (var i = 0; i < workerExportNames.length; i++) {
            var name = workerExportNames[i];
            if (name.charAt(name.length - 1) == "*") {
                name = name.substr(0, name.length - 1);
                ret[name] = makePostMessageForFunction(name, true);
            } else {
                ret[name] = makePostMessageForFunction(name, false);
            }
        }

        return ret; //we return an object which lets the main thread call the worker.
The object will take care of the communication in the other direction.
    }

    function customLogger() {
        var msg = "";
        for (var i = 0; i < arguments.length; i++) {
            if (typeof arguments[i] !== 'string') {
                msg += "<span style='color:blue;'>" + JSON.stringify(arguments[i]) +
"</span> ";
            }
            else {
                msg += arguments[i] + " ";
            }
        }

        currentLog.append(msg);
    }

    $("#learnXOR").click(function () {
        $("#xorLogger").html("");

        var old_console = console.log;
        currentLog = $("#xorLogger");

        customLogger("Creating Network");
        customLogger("XORNetwork = new Intellect({SingleThreaded:true});");
        var XORNetwork = new Intellect({SingleThreaded: true});
        customLogger("Training on XOR");

customLogger("XORNetwork.Train([{<br>input:[0,0],output:[0]},<br>{input:[1,0],output:[1
]},<br>{input:[0,1],output:[1]},<br>{input:[1,1],output:[0]<br>}],<br>{Training
Options},<br>callback);");
        var start = window.performance.now();
        XORNetwork.Train([{input: [0, 0], output: [0]}, {input: [1, 0], output: [1]}, {
            input: [0, 1],
            output: [1]
```

```
        }, {input: [1, 1], output: [0]}], {}, function (result) {
            var end = window.performance.now() - start;
            customLogger("Done. Training took:", end, "ms. Testing Results...");
            customLogger(result);
            //XORNetwork.ActivationFunction =
XORNetwork.ActivationFunctions["ElliotSymmetrical"];
            customLogger("XORNetwork.ComputeOutputs([0,0]) = ",
XORNetwork.ComputeOutputs([0, 0]));
            customLogger("XORNetwork.ComputeOutputs([1,0]) = ",
XORNetwork.ComputeOutputs([1, 0]));
            customLogger("XORNetwork.ComputeOutputs([0,1]) = ",
XORNetwork.ComputeOutputs([0, 1]));
            customLogger("XORNetwork.ComputeOutputs([1,1]) = ",
XORNetwork.ComputeOutputs([1, 1]));
        });

    });

    $("#loadNetworkButton").click(function () {
        $("#loadNetwork").click()
    });

    var ImageLoaderWorker = function () {
        "use strict"; // This will become the first line of the worker

        function parseTrainingImages(buff) {
            var dv = new DataView(buff)
            var magicNumber = dv.getUint32(0);
            switch (magicNumber) {
                case 2049: // MNIST LABEL FILE
                    var vector_length = dv.getUint32(4);
                    var labels = new Uint8Array(vector_length);
                    var offset = 8;
                    for (var i = 0; i < vector_length; i++) {
                        labels[i] = dv.getUint8(offset++);
                    }
                    main.updateImageData({type: "labels", labels: labels.buffer},
[labels.buffer])
                    break;
                case 2051: // MNIST IMAGE FILE
                    var vector_length = dv.getUint32(4);
                    var width = dv.getUint32(8); // 0+uint8 = 1 bytes offset
                    var height = dv.getUint32(12); // 0+uint8+uint16 = 3 bytes offset
                    var length = width * height * vector_length;
                    var images = new Uint8Array(length);
                    var offset = 13;
                    for (var i = 0; i < length; i++) {
                        images[i] = dv.getUint8(offset++);
                    }
                    main.updateImageData({
                        type: "images",
                        numImages: vector_length,
                        width: width,
                        height: height,
                        images: images.buffer
                    }, [images.buffer]);
                    break;
                default:
```

```
                    console.error("Unknown File Type")
                }
            }
        };

        var updateImageData = function (buffer) {

            if (buffer.type == "labels") {
                labels = new Uint8Array(buffer.labels);
                console.log("Labels Loaded!");
            }
            else if (buffer.type == "images") {
                imagesBuffer = new Uint8Array(buffer.images);
                width = buffer.width;
                height = buffer.height;
                var singleimage = width * height;
                images = new Array(buffer.numImages);
                for (var i = 0; i < buffer.numImages; i++) {

                    images[i] = imagesBuffer.subarray(i * singleimage, (i + 1) *
singleimage);
                }
                console.log("Images Loaded!");
            }
        }

        var imageWorker = BuildBridgedWorker(ImageLoaderWorker, ["parseTrainingImages*"],
["updateImageData*"], [updateImageData]);

        function loadTrainingImages(evt) {
            var files = evt.target.files; // FileList object

            // files is a FileList of File objects. List some properties.
            for (var i = 0, f; f = files[i]; i++) {
                var reader = new FileReader();

                // Closure to capture the file information.
                reader.onload = (function (theFile) {
                    return function (e) {
                        // parse data
                        var buffer = new Uint32Array(this.result);
                        imageWorker.parseTrainingImages(buffer.buffer, [buffer.buffer]);
                    };
                })(f);

                // Read in the image file as a data URL.
                reader.readAsArrayBuffer(f);
            }
        }

        function loadNetwork(evt) {
            var files = evt.target.files; // FileList object

            // files is a FileList of File objects. List some properties.
            for (var i = 0, f; f = files[i]; i++) {
                var reader = new FileReader();

                // Closure to capture the file information.
```

```
            reader.onload = (function (theFile) {
                return function (e) {
                    MNISTNet = new Intellect();
                    MNISTNet.loadNetwork(e.target.result);
                }
            })(f);

            // Read in the image file as a data URL.
            reader.readAsText(f);
        }
    }


    document.getElementById('training_images').addEventListener('change',
loadTrainingImages, false);
    document.getElementById('loadNetwork').addEventListener('change', loadNetwork,
false);

    $("#checkDigit").click(function () {
        var index = parseInt($("#mnistIndex").val());
        if (!MNISTNet) {
            createMNISTNetwork();
        }
        CheckOutput(index);
    });
    function drawDigit(index) {
        var canvas = document.createElement('canvas');
        canvas.width = width;
        canvas.height = height;
        var context = canvas.getContext('2d');
        var image = context.createImageData(width, height);
        for (var i = 0, j = 0; i < images[index].length; i++, j += 3) {
            image.data[i + j + 0] = 255 - images[index][i];
            image.data[i + j + 1] = 255 - images[index][i];
            image.data[i + j + 2] = 255 - images[index][i];
            image.data[i + j + 3] = 255;
        }
        context.putImageData(image, 0, 0);
        $("#MNISTLogger").append(canvas);
    }

    function drawDigit2(sampleImage) {
        var canvas = document.createElement('canvas');
        canvas.width = 28;
        canvas.height = 28;
        var context = canvas.getContext('2d');
        var image = context.createImageData(28, 28);
        for (var i = 0, j = 0; i < sampleImage.length; i++, j += 3) {
            image.data[i + j + 0] = 255 - sampleImage[i];
            image.data[i + j + 1] = 255 - sampleImage[i];
            image.data[i + j + 2] = 255 - sampleImage[i];
            image.data[i + j + 3] = 255;
        }
        context.putImageData(image, 0, 0);
        $("#MNISTLogger").append(canvas);
    }
```

```
    $("#learnMNIST").click(function () {

        var old_console = console.log;
        currentLog = $("#MNISTLogger");


        currentLog.html("");

        createMNISTNetwork();
        var TrainingSize = parseInt($("#trainingSize").val());
        var trainingOptions = {
            epochs: parseInt($("#epochs").val()),
            errorThresh: parseFloat($("#errorThreshold").val()),
            learningRate: parseFloat($("#learningRate").val()),
            momentum: parseFloat($("#momentum").val()),
            BatchSize: 1,
            callback: customLogger,
            callbackPeriod: parseInt($("#logPeriod").val())
            //logPeriod: parseInt($("#logPeriod").val()),
            //log:true
        };


        var trainingData = [];
        for (var i = 0; i < TrainingSize; i++) {
            var output = [0, 0, 0, 0, 0, 0, 0, 0, 0, 0];
            output[labels[i]] = 1;
            trainingData[i] = {input: images[i], output: output}
        }


        MNISTNet.Train(trainingData, trainingOptions, function (result) {
            customLogger(result);
        });

    });

    function createMNISTNetwork() {

        MNISTOptions.Layers[1] = parseInt($("#digitHidden").val());

        MNISTNet = new Intellect(MNISTOptions);
    }

    function checkAccuracy() {
        var accuracy = 0;
        for (var i = 0; i < images.length; i++) {
            var result = getMax(MNISTNet.ComputeOutputs(images[i]));
            if (result == labels[i]) {
                accuracy++;
            }
        }

        console.log("Accuracy: ", accuracy, "/10000 or ", (accuracy / images.length) *
100, "%");
    }

    function getMax(array) {
```

```
        var max = 0;
        for (var i = 1; i < array.length; i++) {
            if (array[i] > array[max]) {
                max = i;
            }
        }
        return max;
    }

    function CheckOutput(index) {
        drawDigit(index);
        var result = MNISTNet.ComputeOutputs(images[index]);
        result = getMax(result);
        $("#MNISTLogger").append("=" + result + "<br>");

    }

    $("#saveNetwork").click(function () {
        MNISTNet.saveNetwork();
    });

    $("#checkDrawing").click(function () {
        MNISTNet = MNISTNet || new Intellect({Layers: [784, 100, 10]});
        var canvas = document.getElementById("digitDrawer");
        var canvas2 = document.createElement("canvas");
        canvas2.width = 28;
        canvas2.height = 28;
        var context = canvas2.getContext('2d');
        context.drawImage(canvas, 0, 0, 28, 28);
        var imageData = context.getImageData(0, 0, canvas2.width, canvas2.height);

        var image = [];
        for (var i = 0, j = 3; i < 784; i++, j += 4) {
            image[i] = imageData.data[j] // just grab the opacity
        }

        drawDigit2(image);
        var result = MNISTNet.ComputeOutputs(image);
        $("#MNISTLogger").append("=" + getMax(result));
        console.log(result);
    });

    $("#clear").click(function () {
        var canvas = document.getElementById("digitDrawer");
        var context = canvas.getContext('2d');
        context.clearRect(0, 0, canvas.width, canvas.height);
    });

</script>
</body>
</html>
```

C++ NEURAL NETWORK LIBRARY NN2.H

```
// NN2.H
// Travis Payton
// Feb 25th 2013
// A new NeuralNetwork written for back propogation,
```

```
// and possible implementation via AMP

#ifndef NN2_H_INCLUDED
#define NN2_H_INCLUDED


#include <vector>
using std::vector;
#include <iostream>
using std::cin;
using std::cerr;
using std::cout;
#include <string>
using std::string;
#include <cmath>
using std::tanh;
using std::exp;
#include <sstream>
#include <fstream>

class NN2
{
private:
  int numInput;
  int numHidden;
  int numOutput;
  int numWeights;


  // Input to Hidden Layers collection of Weights
  vector<float> inputs;
  vector<vector <float>> ihWeights;
  vector<float> ihSums;
  vector<float> ihBiases;
  vector<float> ihOutputs;

  // Hidden to Output Layer collection of Weights

  vector<vector<float>> hoWeights;
  vector<float> hoSums;
  vector<float> hoBiases;
  vector<float> outputs;


  // Vectors for Backpropogation

  vector<float> oGrads; // output gradients
  vector<float> hGrads; // hidden gradients

  // Vectors that hold the deltas from the previous iteration
  vector<vector<float>> ihPrevWeightsDelta;
  vector<float> ihPrevBiasesDelta;
  vector<vector<float>> hoPrevWeightsDelta;
  vector<float> hoPrevBiasesDelta;

public:

  // Constructor, initialize the neural network
```

```
  NN2(int numI, int numH, int numO) {

        numInput=numI; numHidden=numH; numOutput=numO;
        numWeights = (numInput * numHidden) + (numHidden * numOutput) + numHidden +
numOutput;
        inputs.resize(numInput);
        ihSums.resize(numHidden);
        ihBiases.resize(numHidden);
        ihWeights = MakeMatrix(numInput,numHidden);
        ihOutputs.resize(numHidden);

        hoWeights = MakeMatrix(numHidden, numOutput);
        hoSums.resize(numOutput);
        hoBiases.resize(numOutput);
        outputs.resize(numOutput);


        oGrads.resize(numOutput);
        hGrads.resize(numHidden);

        ihPrevWeightsDelta = MakeMatrix(numInput,numHidden);
        ihPrevBiasesDelta.resize(numHidden);
        hoPrevWeightsDelta = MakeMatrix(numHidden,numOutput);
        hoPrevBiasesDelta.resize(numOutput);

  }

  int getNumWeights()
  {
     return numWeights;
  }

  void UpdateWeights(vector<float> tValues, float eta, float alpha) {
  // assumes that SetWeights and ComputeOutputs have been called and so all the
internal arrays and matrices have values (other than 0.0)

      // 1. compute output gradients
      for (int i = 0; i < oGrads.size(); i++)
      {
        float derivative = (1 - outputs[i]) * (1 + outputs[i]); // derivative of tanh
        oGrads[i] = derivative * (tValues[i] - outputs[i]);
      }

      // 2. compute hidden gradients
      for (int i = 0; i < hGrads.size(); i++)
      {
        float derivative = (1 - ihOutputs[i]) * ihOutputs[i]; // (1 / 1 + exp(-x))'  --
using output value of neuron
        float sum = 0.0;
        for (int j = 0; j < numOutput; ++j) // each hidden delta is the sum of
numOutput terms
          sum += oGrads[j] * hoWeights[i][j]; // each downstream gradient * outgoing
weight
        hGrads[i] = derivative * sum;
      }

      // 3. update input to hidden weights (gradients must be computed right-to-left
but weights can be updated in any order
```

```
      for (int i = 0; i < ihWeights.size(); i++) // 0..2 (3)
      {
        for (int j = 0; j < ihWeights[0].size(); ++j) // 0..3 (4)
        {
          float delta = eta * hGrads[j] * inputs[i]; // compute the new delta
          ihWeights[i][j] += delta; // update
          ihWeights[i][j] += alpha * ihPrevWeightsDelta[i][j]; // add momentum using
previous delta. on first pass old value will be 0.0 but that's OK.
          ihPrevWeightsDelta[i][j] = delta;
               }
      }

      // 3b. update input to hidden biases
      for (int i = 0; i < ihBiases.size(); i++)
      {
        float delta = eta * hGrads[i] * 1.0; // the 1.0 is the constant input for any
bias; could leave out
        ihBiases[i] += delta;
        ihBiases[i] += alpha * ihPrevBiasesDelta[i];
              ihPrevBiasesDelta[i] = delta;
      }

      // 4. update hidden to output weights
      for (int i = 0; i < hoWeights.size(); i++)  // 0..3 (4)
      {
        for (int j = 0; j < hoWeights[0].size(); ++j) // 0..1 (2)
        {
          float delta = eta * oGrads[j] * ihOutputs[i];  // see above: ihOutputs are
inputs to next layer
          hoWeights[i][j] += delta;
          hoWeights[i][j] += alpha * hoPrevWeightsDelta[i][j];
          hoPrevWeightsDelta[i][j] = delta;
        }
      }

      // 4b. update hidden to output biases
      for (int i = 0; i < hoBiases.size(); i++)
      {
        float delta = eta * oGrads[i] * 1.0;
        hoBiases[i] += delta;
        hoBiases[i] += alpha * hoPrevBiasesDelta[i];
        hoPrevBiasesDelta[i] = delta;
      }
    } // UpdateWeights

  void SetWeights(vector<float> weights) {
  // copy weights and biases in weights[] array to i-h weights, i-h biases, h-o
weights, h-o biases

      int k = 0; // points into weights param

      for (int i = 0; i < numInput; i++)
        for (int j = 0; j < numHidden; ++j)
          ihWeights[i][j] = weights[k++];

      for (int i = 0; i < numHidden; i++)
        ihBiases[i] = weights[k++];
```

```
      for (int i = 0; i < numHidden; i++)
        for (int j = 0; j < numOutput; ++j)
          hoWeights[i][j] = weights[k++];

      for (int i = 0; i < numOutput; i++)
        hoBiases[i] = weights[k++];
  }

  bool SetWeightsFromFile(string inputfile ) {
       //load the file
       std::ifstream file(inputfile.c_str());

       // file does not exist, or didn't open
       if (!file) {
              std::cerr<<"ERROR: Cannot read from file \n";
        return false;}
    // go until the end of the file is reached
       vector <float> tmpweights;
    while(!file.eof()) {
              std::string line;
        unsigned int float_bits;

        // if read error, not due to EOF
              if (!file && !file.eof()) return false;

              while(getline(file, line, ','))
              {
                     //more crazy pointer magic to convert the bits to a float
                     file >> float_bits;
                     unsigned int * float_bits_addr = &float_bits;
                     float score = *(float *)float_bits_addr;
                     tmpweights.push_back(score);
              }

    }
       SetWeights(tmpweights);
       return true; //read completed without errors
}

  vector<float> GetWeights() {

      vector<float> result(numWeights);
      int k = 0;
      for (int i = 0; i < ihWeights.size(); i++)
        for (int j = 0; j < ihWeights[0].size(); ++j)
          result[k++] = ihWeights[i][j];
      for (int i = 0; i < ihBiases.size(); i++)
        result[k++] = ihBiases[i];
      for (int i = 0; i < hoWeights.size(); i++)
        for (int j = 0; j < hoWeights[0].size(); ++j)
          result[k++] = hoWeights[i][j];
      for (int i = 0; i < hoBiases.size(); i++)
        result[k++] = hoBiases[i];
      return result;
  }

// GetWeightsToFile
// This function takes gets the weights and out puts into a CSV file
```

```
// Pre Conditions:  NONE
// Post Conditions:  The data is stored in a file of the specified name

void GetWeightsToFile (string filename)
{
       std::ofstream csvfile;
       csvfile.open(filename.c_str());
       vector <float> weights = GetWeights();
       for (int i = 0; i < weights.size(); i++)
       {
                  //and now for crazy pointer magic to store the bits of a float
reather than risk losing precision
                      const float * ptr = &(weights[i]);
                      csvfile<<(*(unsigned int *)ptr) << " , ";
       }
       csvfile.close();
}

  vector<float> ComputeOutputs(vector<float> xValues) {


      for (int i = 0; i < numHidden; i++)
        ihSums[i] = 0.0;
      for (int i = 0; i < numOutput; i++)
        hoSums[i] = 0.0;

      inputs = xValues; // copy x-values to inputs

      for (int j = 0; j < numHidden; ++j)  // compute input-to-hidden weighted sums
        for (int i = 0; i < numInput; i++)
          ihSums[j] += inputs[i] * ihWeights[i][j];

      for (int i = 0; i < numHidden; i++)  // add biases to input-to-hidden sums
        ihSums[i] += ihBiases[i];

      for (int i = 0; i < numHidden; i++)   // determine input-to-hidden output
        ihOutputs[i] = SigmoidFunction(ihSums[i]);

      for (int j = 0; j < numOutput; ++j)   // compute hidden-to-output weighted sums
        for (int i = 0; i < numHidden; i++)
          hoSums[j] += ihOutputs[i] * hoWeights[i][j];

      for (int i = 0; i < numOutput; i++)  // add biases to input-to-hidden sums
        hoSums[i] += hoBiases[i];

      for (int i = 0; i < numOutput; i++)   // determine hidden-to-output result
        outputs[i] = HyperTanFunction(hoSums[i]);

      vector<float> result = outputs; // could define a GetOutputs method instead


      return result;
    } // ComputeOutputs


private:

  vector<vector<float>> MakeMatrix(int rows, int cols)
```

```
  {
  vector<vector<float>> result(rows,vector<float>(cols));
  return result;
  }

  float SigmoidFunction(float x)
  {
    if (x < -45.0) return 0.0;
    else if (x > 45.0) return 1.0;
    else return 1.0 / (1.0 + exp(-x));
  }

  float HyperTanFunction(float x)
  {
    if (x < -10.0) return -1.0;
    else if (x > 10.0) return 1.0;
    else return tanh(x);
  }
}; // End NN2 Class
#endif
```

C++ TO JS PORT OF NN2.H

```
/**
 * Created by tpayton1 on 11/7/2014.
 */
var NN = function(numinput, numhidden, numoutput){
    this.numInputs = numinput;
    this.numHidden = numhidden;
    this.numOutput = numoutput;
    this.numWeights = (numinput * numhidden) + (numhidden * numoutput) + numhidden +
numoutput;

    // Define Helper functions
    this.makeMatrix = function(rows,cols) {
        var a=[];
        for (var i=0;i<rows;i++){
            a.push(new Array(cols));
        }
        return a;
    };

    // Activation Functions http://jsperf.com/neural-net-activation-functions
    this.SigmoidFunction = function(x)
    {
        if (x < -45.0) return 0.0;
        else if (x > 45.0) return 1.0;
        else return 1.0 / (1.0 + Math.exp(-x));
    };

    this.HyperTanFunction = function(x)
    {
        if (x < -10.0) return -1.0;
        else if (x > 10.0) return 1.0;
        else return Math.tanh(x);
    };

    // Input hidden layer arrays
    // These needs to be updated if the hidden layer is adjusted during runtime
    this.inputs = new Array(this.numInputs);
    this.ihSums = new Array(this.numHidden); // best performance in V8 engine for
filling http://jsperf.com/preallocating-array/8
    this.ihBiases = new Array(this.numHidden);
    this.ihWeights = this.makeMatrix(this.numInputs,this.numHidden); // make a Flat
array for faster access;
    this.ihOutputs = new Array(this.numHidden);

    // Hidden Layer to Outputs
    // These also need to be updated if the hidden layer is adjusted during runtime
    this.hoSums = new Array(this.numOutput);
    this.hoBiases = new Array(this.numOutput);
    this.hoWeights = this.makeMatrix(this.numHidden, this.numOutput);
    this.outputs = new Array(this.numOutput);

    //Backpropagation arrays
    this.oGrads = new Array(this.numOutput); // output gradients
    this.hGrads = new Array(this.numHidden); // hidden gradients
```

```
    // Deltas from previous evaluation
    this.ihPrevWeightsDelta = this.makeMatrix(this.numInputs,this.numHidden);
    for (var i = 0; i < this.numInputs; i++)
        for(var j=0; j < this.numHidden; j++)
            this.ihPrevWeightsDelta[i][j] = 0;
    this.ihPrevBiasesDelta = new Array(this.numHidden);
    for(var j=0; j < this.numHidden; j++)
        this.ihPrevBiasesDelta[j] = 0;

    this.hoPrevWeightsDelta = this.makeMatrix(this.numHidden, this.numOutput);
    for (var i = 0; i < this.numHidden; i++)
        for(var j=0; j < this.numOutput; j++)
            this.hoPrevWeightsDelta[i][j] = 0;

    this.hoPrevBiasesDelta = new Array(this.numOutput);
    for(var j=0; j < this.numOutput; j++)
        this.hoPrevBiasesDelta[j] = 0;

    // Functions

    // BackPropagate
    // This updates the weights of the NN using the backpropagation technique
    // Takes an array of expected outputs, eta: amount to change, alpha:momentum based
on previous weight delta
    this.BackPropagate = function(tValues, eta, alpha) {
        // 1. Compute Output Gradients
        for (var i = 0; i < this.oGrads.length; i++){
            //Calculate Derivative of HyperTan Function (tanh)
            var derivative = (1-this.outputs[i]) * (1 + this.outputs[i]);
            this.oGrads[i] = derivative * (tValues[i] - this.outputs[i]);
        }

        // 2. Compute Hidden Gradients
        for (var i = 0; i < this.hGrads.length; i++){
            // Calculate Derivative of Sigmoid Function '1 / (1 + exp(-x))'
            var derivative = (1-this.ihOutputs[i]) * this.ihOutputs[i];
            var sum = 0.0;
            for (var j=0; j < this.numOutput; j++){
                sum+= this.oGrads[j] * this.hoWeights[i][j];
            }
            this.hGrads[i] = sum * derivative;
        }
        // 3. Update input to hidden weights (gradients must be computed right-to-left
but weights can be updated in any order
        for (var i=0; i < this.ihWeights.length; i++) {
            for (var j = 0; j < this.ihWeights[0].length; j++) {
                var delta = eta * this.hGrads[j] * this.inputs[i];
                this.ihWeights[i][j] += delta + (alpha *
this.ihPrevWeightsDelta[i][j]);
                this.ihPrevWeightsDelta[i][j]= delta;
            }
        }
        console.log("Old Network Hidden Weights: ",this.ihWeights);
        // 3b. Update input to hidden biases
        for (var i=0; i < this.ihBiases.length; i++){
            var delta = eta * this.hGrads[i];
            this.ihBiases[i] += delta + (alpha * this.ihPrevBiasesDelta[i]);
            this.ihPrevBiasesDelta[i] = delta;
```

```
        }
        console.log("Old Network Hidden Biases: ",this.ihBiases);
        // 4. Update hidden to output weights
        for (var i=0; i < this.hoWeights.length; i++){
            for (var j=0; j < this.hoWeights[0].length; j++){
                var delta = eta * this.oGrads[j] * this.ihOutputs[i];
                this.hoWeights[i][j] += delta + (alpha *
this.hoPrevWeightsDelta[i][j]);
                this.hoPrevWeightsDelta[i][j] = delta;
            }
        }
        console.log("Old Network Output Weights: ",this.hoWeights);
        // 4b. Update hidden to output biases
        for (var i =0; i < this.hoBiases.length; i++){
            var delta = eta * this.oGrads[i];
            this.hoBiases[i] += delta + (alpha * this.hoPrevBiasesDelta[i])
            this.hoPrevBiasesDelta[i] = delta;
        }
        console.log("Old Network Output Biases: ",this.hoBiases);
    };

    this.ComputeOutputs = function(xValues){
        for (var i = 0; i < this.ihSums.length; i++){
            this.ihSums[i] = 0;
        }
        for (var i = 0; i < this.hoSums.length; i++){
            this.hoSums[i] =0;
        }
        this.inputs = xValues;

        for (var j = 0; j < this.numHidden; ++j)  // compute input-to-hidden weighted
sums
        for (var i = 0; i < this.numInputs; i++)
        this.ihSums[j] += xValues[i] * this.ihWeights[i][j];

        for (var i = 0; i < this.numHidden; i++)  // add biases to input-to-hidden sums
        this.ihSums[i] += this.ihBiases[i];

        for (var i = 0; i < this.numHidden; i++)   // determine input-to-hidden output
        this.ihOutputs[i] = this.SigmoidFunction(this.ihSums[i]);

        for (var j = 0; j < this.numOutput; ++j)   // compute hidden-to-output weighted
sums
        for (var i = 0; i < this.numHidden; i++)
        this.hoSums[j] += this.ihOutputs[i] * this.hoWeights[i][j];

        for (var i = 0; i < this.numOutput; i++)  // add biases to hidden-to-output
sums
        this.hoSums[i] += this.hoBiases[i];

        for (var i = 0; i < this.numOutput; i++)   // determine hidden-to-output result
        this.outputs[i] = this.HyperTanFunction(this.hoSums[i]);
        return this.outputs;

    };


    this.SetWeights = function(weights) {
```

```
        // copy weights and biases in weights[] array to i-h weights, i-h biases, h-o
weights, h-o biases

        var k = 0; // points into weights param

        for (var i = 0; i < this.numInputs; i++)
        {for (var j = 0; j < this.numHidden; ++j){
                this.ihWeights[i][j] = weights[k++];}}


        for (var i = 0; i < this.numHidden; i++)
            this.ihBiases[i] = weights[k++];



        for (var i = 0; i < this.numHidden; i++)
            for (var j = 0; j < this.numOutput; ++j)
                this.hoWeights[i][j] = weights[k++];


        for (var i = 0; i < this.numOutput; i++)
            this.hoBiases[i] = weights[k++];


    };


};
```

APPENDIX B – BENCHMARKS

ARRAY CREATION AND FILLING

Live Example: http://jsperf.com/preallocating-array/15

Preparation code:

```
<script>
  var N = 10000;
</script>
```

Tests and Results:

Name	Code	Operations per Second

a=[]; a.push(0);	var a = [];
	for (var i = 0; i < N; ++i) {
	  a.push(0);
	}
		10,199

a=new Array(N);
a.length=0;
a.push(0);
	var a = new Array(N);
	a.length = 0;
	for (var i = 0; i < N; i++) {
	  a.push(0);
	}
		9,350

a=new Array(N);
a[i]=0;
	var a = new Array(N);
	for (var i = 0; i < N; i++) {
	  a[i] = 0;
	}
		40,016

a = []; a[i]=0;	var a = [];
	for (var i = 0; i < N; i++) {
	  a[i] = 0;
	}
		12,784

a=[]; a.length = N; a[i]=0;	var a = [];
	a.length = N;
	for (var i = 0; i < N; i++) {
	  a[i] = 0;
	}
		49,349

1D VS 2D ARRAY ACCESS

Live Example: http://jsperf.com/array-access-2d-vs-1d

Preparation code:

```
<script>
var rowSize = 100;
var colSize = 100;
var test1DArray = [];
var test2DArray = [];

for (var i = 0; i < rowSize; i++){
   test2DArray.push([]);
   for (var j =0; j < colSize; j++){
      var value = Math.random();
      test1DArray[colSize*i+j] = value;
      test2DArray[i][j] = value;
   }
}

</script>
```

Tests and Results:

Name	Code	Operations per Second

1D Array	`for (var i = 0; i < rowSize; i++) {`
	`  for (var j = 0; j < colSize; j++) {`
	`  test1DArray[i * colSize + j] = 0;`
	`  }`
	`}`
		70,860

2D Array	`for (var i = 0; i < rowSize; i++) {`
	`  for (var j = 0; j < colSize; j++) {`
	`  test2DArray[i][j] = 0;`
	`  }`
	`}`
		87,230

Sequential 1D	`for (var i = 0; i < rowSize * colSize; i++) {`
	`  test1DArray[i] = 1;`
	`}`
		94,303

TYPED ARRAY VS NORMAL ARRAY

Live Example: http://jsperf.com/array-comparison-typed-vs-regular/4

Preparation Code:

```
<script>
var N = 10000;
var float32Array = new Float32Array(N);
var float64Array = new Float64Array(N);
var jsArray = new Array(N);
</script>
<script>
  Benchmark.prototype.setup = function() {
    for (var i = 0; i < N; i++) {
      float32Array[i] = Math.random();
      float64Array[i] = Math.random();
      jsArray[i] = Math.random();
    }
  };
</script>
```

Tests and Results

Name	Code	Operations per Second
Native JS Array	`for (var i = 0; i < N; i++) { jsArray[i] = 0; }`	93,089
Float32Array	`for (var i = 0; i < N; i++) { float32Array[i] = 0; }`	62,841
Float64Array	`for (var i = 0; i < N; i++) { float64Array[i] = 0; }`	93,089

ARRAY COMPARISON FOR NETWORK EVALUATION

Live Example: http://jsperf.com/array-comparison-for-network-evaluations

Preparation Code:

```
<script>
var inputs = 40;
var hidden = 50;
var outputs = 10;

var inputValues = new Array(inputs);
for (var i = 0; i < inputs; i++){
    inputValues[i] = Math.random();
}

function Sigmoid(x) {
    return 1/(1+Math.exp(-x));
}

var numNeurons = hidden + outputs;
var numWeights = hidden*(inputs + outputs);

// Create 1D Float64Array Network
var Network1D = function(){
   this.Weights = new Float64Array(numWeights);
   for (var i=0; i < numWeights; i++){
       this.Weights[i] = Math.random();
   }
   this.Biases = new Float64Array(numNeurons);
   for (var i=0; i < numNeurons; i++){
        this.Biases[i] = Math.random();
   }

   this.Outputs = new Float64Array(numNeurons + inputs);
   this.Layers = [inputs,hidden,outputs];
   this.outputLayer = this.Layers.length - 1;
   this.ComputeOutputs = function(input){
               for (var i= 0,length = input.length;i<length;i++){
                   this.Outputs[i] = input[i];
        }
        // Compute the Outputs of each Layer
        var weights= 0, neurons = 0, outputs = this.Layers[0], output = 0;
        for (var layer = 1; layer <= this.outputLayer; layer++) {
            for (var neuron = 0, size = this.Layers[layer]; neuron < size; neuron++) {
                var sum = this.Biases[neurons];
                for (var z = 0, length = this.Layers[layer -1]; z < length; z++) {
                    sum += this.Weights[weights++] * this.Outputs[output+z];
                }
                this.Outputs[outputs++] = Sigmoid(sum);
            }
            output += length;
        }

        return Array.prototype.slice.call(this.Outputs,-this.Layers[this.outputLayer]);
   }
};
```

```
// Create 1D Float64Array Network with 3D Array Views
var Network1DViews = function(){
    this.Layers = [inputs,hidden,outputs];
    this.outputLayer = this.Layers.length - 1;


    this.randWeights = function (start, size) {
        var weights = this.WeightsBuffer.subarray(start,start + size);
        for (var i = 0; i < size; i++) {
            weights[i] = Math.random();
        }
        return weights;
    },

    this.randBias = function (offset,size) {
        var biases = this.BiasesBuffer.subarray(offset,offset+size);
        for (var i = 0; i < size; i++) {
            biases[i] = Math.random();;
        }
        return biases;
    };

    // Create Buffers
    this.WeightsBuffer = new Float64Array(numWeights);
    this.BiasesBuffer = new Float64Array(numNeurons);
    this.OutputsBuffer = new Float64Array(numNeurons + inputs);

    // Create Views
    this.Biases = [];
    this.Weights = [];
    this.Outputs = [];

    // Link Views to appropriate slices of the buffer;
    // Map first Layer of Outputs to OutputsBuffer
    this.Outputs[0] = this.OutputsBuffer.subarray(0,this.Layers[0]);
    var offset = 0;
    var weight = 0;
    // Map the rest of the Array views to their Buffers
    for (var layer = 1; layer <= this.outputLayer; layer++) {
        var size = this.Layers[layer];
        this.Outputs[layer] =
this.OutputsBuffer.subarray(offset+this.Layers[0],offset+this.Layers[0]+size);
        this.Biases[layer] = this.randBias(offset,size);
        this.Weights[layer] = new Array(size);
        for (var neuron = 0; neuron < size; neuron++) {
            var prevSize = this.Layers[layer - 1];
            this.Weights[layer][neuron] = this.randWeights(weight,prevSize);
            weight+=prevSize;
        }

        offset+=size;
    }

    this.ComputeOutputs = function(input){
        this.Outputs[0].set(input);
        // Compute the Outputs of each Layer
        for (var layer = 1; layer <= this.outputLayer; layer++) {
            for (var neuron = 0, size = this.Layers[layer]; neuron < size; neuron++) {
```

```
                var weights = this.Weights[layer][neuron];
                var sum = this.Biases[layer][neuron];
                for (var z = 0, length = weights.length; z < length; z++) {
                    sum += weights[z] * input[z];
                }
                this.Outputs[layer][neuron] = Sigmoid(sum);
            }
            var output = input = this.Outputs[layer];
        }

        return output;
    }
};

// Create 3D Network;
var Network3D = function(){
    this.Layers = [inputs,hidden,outputs];

    this.outputLayer = this.Layers.length - 1;
    this.randWeights = function (size) {
        var weights = new Array(size);
        for (var i = 0; i < size; i++) {
            weights[i] = Math.random();
        }
        return weights;
    };

    this.randBias = function (size) {
        var biases = new Array(size);
        for (var i = 0; i < size; i++) {
            biases[i] = Math.random();
        }
        return biases;
    };

    this.zeros = function (size) {
        var arr = new Array(size);
        for (var i = 0; i < size; i++) {
            arr[i] = 0;
        }
        return arr;
    };

    this.Biases = [];
    this.Weights = [];
    this.Outputs = [];

    for (var layer = 0; layer <= this.outputLayer; layer++) {
        var size = this.Layers[layer];
        this.Outputs[layer] = this.zeros(size);

        if (layer > 0) {
            this.Biases[layer] = this.randBias(size);
            this.Weights[layer] = new Array(size);

            for (var node = 0; node < size; node++) {
                var prevSize = this.Layers[layer - 1];
                this.Weights[layer][node] = this.randWeights(prevSize);
```

```
            }
        }
    }

    this.ComputeOutputs = function(input){
        this.Outputs[0] = input;
        // Compute the Outputs of each Layer
        for (var layer = 1; layer <= this.outputLayer; layer++) {
            for (var node = 0, nodes = this.Layers[layer]; node < nodes; node++) {
                var weights = this.Weights[layer][node];
                var sum = this.Biases[layer][node];
                for (var k = 0, length = weights.length; k < length; k++) {
                    sum += weights[k] * input[k];
                }
                this.Outputs[layer][node]=Sigmoid(sum);
            }
            var output = input = this.Outputs[layer];
        }

        return output;
    }

};

var NN1 = new Network1D();
var NN2 = new Network3D();
var NN3 = new Network1DViews();
</script>
```

Tests and Results:

Name	Code	Operations per Second
1D Float64Array	`NN1.ComputeOutputs(inputValues);`	119,276
3D JavaScript Array	`NN2.ComputeOutputs(inputValues);`	191,659
3D Array of ArrayViews on Single Float64ArrayBuffer	`NN3.ComputeOutputs(inputValues);`	190,552
Standalone Function (in version 2 of the test)	`NN2.Standalone(inputValues);`	16,955

C++ BENCHMARK

```
#include "NN2.h"
#include <iostream>
#include <vector>
#include <stdlib.h>     /* srand, rand */
#include <time.h>       /* time */
using namespace std;

#if defined(_WIN32) || defined(__WIN32__) || defined(WIN32)

namespace win32 {
#include <windows.h>
}

class timer
{
  win32::LARGE_INTEGER start_time_;
public:
  timer() { QueryPerformanceCounter(&start_time_); }
  void   restart() { QueryPerformanceCounter(&start_time_); }
  double elapsed() const
  {
    win32::LARGE_INTEGER end_time, frequency;
    QueryPerformanceCounter(&end_time);
    QueryPerformanceFrequency(&frequency);
    return double(end_time.QuadPart - start_time_.QuadPart)
      / frequency.QuadPart;
  }
};

#else

#include <ctime>

class timer
{
  clock_t _start_time;
public:
  timer() { _start_time = clock(); }
  void   restart() { _start_time = clock(); }
  double elapsed() const
  {
    return double(clock() - _start_time) / CLOCKS_PER_SEC;
  }
};

#endif

template< typename Func >
double measure_time(Func f)
{
  timer t;
  f();
  return t.elapsed();
}

NN2 testNetwork(500,200,10);
```

```
vector<vector<float>> inputs;
vector<float> output;

void RunComputeWeights(){
       output = testNetwork.ComputeOutputs(inputs[0]);
}


int main() {

    vector<float> weights;
    for (int i =0; i<testNetwork.getNumWeights(); i++)
    {
        weights.push_back(rand());
    }

    testNetwork.SetWeights(weights);

  cout << "Network Created.  Setting up Inputs" << endl;

    vector<float> tmpInputs(500);
    for (int i=0; i < 1000; i++)
    {
        for (int j=0; j<500;j++){
            tmpInputs[j]=rand();
        }
        inputs.push_back(tmpInputs);
    }
  cout << "Benchmarking ComputeWeights funciton" << endl;

  double time = 0.0;

  for (int i = 0; i < 1000; i++){
    time += measure_time(&RunComputeWeights);
  }
    cout << time / 1000 << endl;

  cout << output[0];
    return 0;
}
```

Results:

0.000240689 seconds optimized

0.0172737 seconds unoptimized (Debug Mode)

JAVASCRIPT VS C++ PERFORMANCE

Live Example: http://jsperf.com/intellect-js-comparison

Preperation code:

```
<script src="http://travispayton.com/Projects/Intellect/JS/neuralnetOld.js"></script>

<script src="http://travispayton.com/Projects/Intellect/JS/Intellect.js"></script>

<script>
var nn1 = new NN(500,200,10);
var nn2 = new Intellect({Layers:[500,200,10], SingleThreaded:true});

var inputs = new Array(500);
for (var i=0; i < 500; i++){
    inputs[i] = Math.random();
}

var weights = new Array(nn1.numWeights);
for (var i=0; i < weights.length; i++){
  weights[i] = Math.random();
}
nn1.SetWeights(weights);
</script>
```

Tests and Results:

Name	Code	Operations per Second		
		Chrome 41	Chrome 44	Firefox 37
NN2 JS Port	`nn1.ComputeOutputs(inputs);`	3,221	3,463	2,760
Intellect.js	`nn2.ComputeOutputs(inputs);`	6,265	7,453	9,687

ANN LIBRARY COMPARISION – INITIALIZATION

Live Example: http://jsperf.com/neural-net-libraries/3

Preparation Code:

```
<script src="http://travispayton.com/Projects/Intellect/JS/Intellect.js"></script>
<script src="http://travispayton.com/Projects/Intellect/JS/brain-0.6.3.js"></script>
<script src="http://travispayton.com/Projects/Intellect/JS/synaptic.js"></script>
<script
src="http://cs.stanford.edu/people/karpathy/convnetjs/build/convnet.js"></script>

<script>
 var inputs = new Array(500);

for (var i=0; i < 500; i++){
 inputs[i] = Math.random();
}
</script>
```

Tests and Results:

Name	Code	Operations per Second
Intellect.js	`var intellecttest = new Intellect({ Layers: [500, 200, 10], SingleThreaded: true });`	95
Synaptic.js	`var synaptictest = new synaptic.Architect.Perceptron(500, 200, 10);`	0.25
Brain.js	`var braintest = new brain.NeuralNetwork({ hiddenLayers: [200] }); braintest.initialize([500, 200, 10]);`	212
ConvNet.js	`var layers = []; layers.push({ type: 'input', out_sx: 1, out_sy: 1, out_depth: 500 }); layers.push({ type: 'fc', num_neurons: 200, activation: 'sigmoid' }); layers.push({ type: 'fc', num_neurons: 10, activation: 'sigmoid' }); var net = new convnetjs.Net();`	95

	net.makeLayers(layers);	

ANN LIBRARY COMPARISION – EVALUATION

Live Example: http://jsperf.com/neural-net-libraries/4

Preparation Code:

```
<script src="http://travispayton.com/Projects/Intellect/JS/Intellect.js"></script>
<script src="http://travispayton.com/Projects/Intellect/JS/brain-0.6.3.js"></script>
<script src="http://travispayton.com/Projects/Intellect/JS/synaptic.js"></script>
<script
src="http://cs.stanford.edu/people/karpathy/convnetjs/build/convnet.js"></script>

<script>
 var inputs = new Array(500);

for (var i=0; i < 500; i++){
 inputs[i] = Math.random();
}

var intellecttest = new Intellect({
  Layers: [500, 200, 10],
  SingleThreaded: true
});

var synaptictest = new synaptic.Architect.Perceptron(500, 200, 10);

var braintest = new brain.NeuralNetwork({
  hiddenLayers: [200]
});
braintest.initialize([500, 200, 10]);

var layers = [];
layers.push({
  type: 'input',
  out_sx: 1,
  out_sy: 1,
  out_depth: 500
});
layers.push({
  type: 'fc',
  num_neurons: 200,
  activation: 'sigmoid'
});
layers.push({
  type: 'fc',
  num_neurons: 10,
  activation: 'sigmoid'
});
var net = new convnetjs.Net();
net.makeLayers(layers);
var covnetInputs = new convnetjs.Vol(inputs);
</script>
```

Tests and Results

Name	Code	Operations per Second

Intellect.js	`intellecttest.ComputeOutputs(inputs);`	6,222
Synaptic.js	`synaptictest.activate(inputs)`	73.93
Brain.js	`braintest.run(inputs);`	6,089
ConvNet.js	`net.forward(covnetInputs);`	5,547

ANN LIBRARY COMPARISON – CREATION AND XOR TRAINING

Live Example: http://jsperf.com/neural-net-libraries/5

Preparation Code:

```
<script src="http://travispayton.com/Projects/Intellect/JS/Intellect.js"></script>
<script src="http://travispayton.com/Projects/Intellect/JS/brain-0.6.3.js"></script>
<script src="http://travispayton.com/Projects/Intellect/JS/synaptic.js"></script>
<script
src="http://cs.stanford.edu/people/karpathy/convnetjs/build/convnet.js"></script>

<script>
 var inputs = [{input:[0,0],output:[0]},
{input:[0,1],output:[1]},
{input:[1,0],output:[1]},
{input:[1,1],output:[0]}]
</script>
```

Tests and Results:

Name	Code	Operations per Second
Intellect.js	`var intellecttest = new Intellect({ SingleThreaded: true }); intellecttest.Train(inputs);`	286
Synaptic.js	`var synaptictest = new synaptic.Architect.Perceptron(2, 3, 1); synaptictest.trainer.XOR({ iterations: 100000, error: .0001, rate: 1 });`	84
Brain.js	`var braintest = new brain.NeuralNetwork(); braintest.train(inputs);`	123