Analyzing Electroencephalograms Using Cloud Computing Techniques

Kathleen Ericson, Shrideep Pallickara, and Charles W. Anderson
Department of Computer Science
Colorado State University
Fort Collins, US
{ericson, shrideep, anderson}@cs.colostate.edu

Abstract— **Brain Computer Interfaces (BCIs) allow users to interact with a computer via electroencephalogram (EEG) signals generated by their brain. The BCI application that we consider allows a user to initiate actions such as keyboard input or control the motion of their wheelchair. Our goal is to be able to train the neural network and classify the EEG signals from multiple users to infer their intended actions in a distributed environment. The processing is developed using the Map-Reduce framework. We use our cloud runtime, Granules, to classify these EEG streams. One of our objectives is to be able to process these EEG streams in real-time. The BCI software has been developed in R, which is an interpreted language designed for the fast computation of matrix multiplications, making it an effective language for the development of artificial neural networks. We contrast our approach of using Granules with a competing approach that uses an R package – Snowfall that simplifies execution of R computations in a distributed setting. We have performed experiments to evaluate the costs introduced by our scheme for training the neural networks and classifying the EEG signals. Our results demonstrate the suitability of using Granules to classify multiple EEG streams in a distributed environment.**

Keywords *Brain Computer Interfaces, Cloud computing, R, Granules, MapReduce, EEG, Artificial Neural Networks*

I. INTRODUCTION

Brain Computer Interfaces (BCIs) allow users to interact with a computer via electroencephalogram (EEG) signals. The BCI application that we consider in this paper allows users who have lost voluntary motion control to initiate actions ranging from controlling a wheelchair [1] to interacting with keyboard interfaces based on their thoughts. EEG signals gathered using electrodes placed on the user's scalp are analyzed using a neural network to reason about the actions initiated by the user. Our focus here is to investigate the possibility of performing such analysis using our cloud runtime, Granules.

Granules [2, 3] is a lightweight runtime for cloud computing and is designed to orchestrate a large number of computations on a cloud. The runtime is designed to support processing of data produced by sensors. Granules supports two of the most dominant models for cloud computing MapReduce [4] and dataflow graphs [5]. In Granules individual computations have a finite state machine associated with them. Computations change state depending on the availability of data on any of their input datasets or as a result of external triggers. When the processing is complete, computations become dormant awaiting data on any of their input datasets.

In Granules, computations specify a scheduling strategy, which in turn govern their lifetimes. Computations specify their scheduling strategy along three dimensions: counts, data driven and periodicity. The counts axis specifies limits on the number of times a computation task will executed. The data driven axis specifies that a computation task needs to be scheduled for execution whenever data is available on any one of its constituent datasets, which could be either streams or files. The periodicity axis specifies that computations should be scheduled for execution at predefined intervals. One can also specify a custom scheduling strategy that is a combination along these three dimensions; for example, limit a computation to be executed 500 times either when data is available or at regular intervals. A computation can change its scheduling strategy during execution, and Granules enforces the newly established scheduling strategy during the next round of execution.

Computations in Granules build state over successive rounds of execution. Though the typical CPU burst time for computations during a given execution is short (seconds to a few minutes), these computations can be long-running with computations toggling between active and dormant states.

There are two benefits to analyzing EEG signals using Granules. Since Granules can interleave streams, and the concomitant processing, from multiple users on a single machine there is a potential for reducing costs; in the current BCI implementations, there is a dedicated processing unit per user. Since Granules orchestrates computations on the set of available machines, even a mid-sized cluster can support a fairly large number of users. This leads us to a second benefit: EEG data from a large number of users can improve the training of the neural networks, which could then be used to improve the accuracy of the inference algorithms.

A. *Research Challenges*

There are three research challenges that we need to address.

1. *Can we process these EEG streams in real-time?* The scheduling overheads introduced by Granules should not preclude real-time processing.
2. *Can we interleave processing of EEG streams from multiple users on the same machine?* Setting aside one machine per user would be inefficient and restrictive.

3. *Can we scale up to a sufficiently large number of users on a set of available machines?* We should be able to accommodate a greater number of users as more machines become available.

B. Paper Contributions

Our contributions are broadly in the area of BCI and cloud computing. The BCI contributions stem from the fact, to the best of our knowledge, this is one of the first attempts to classify EEG streams from multiple users in realtime in a cluster setting. Cloud computing contributions stem from the fact that this is the first time that the MapReduce programming model has been used to classify such EEG streams.

C. Paper Organization

The remainder of this paper is organized as follows: In section II we describe our BCI application. In section III we describe how we train the neural network and classify EEG signals using Granules. We describe an alternative scheme using an R package, Snowfall for performing the distributed analysis. The experimental setup for our experiments is described in section IV, with the results presented in section VI. We present our conclusions and future directions in section VIII.

II. Our BCI application

The BCI application we consider allow users who have lost voluntary motor control to interact with a computer using electroencephalograms (EEG). Users are fitted with an EEG cap, which holds a number of electrodes close to the users scalp. These electrodes pick up electrical impulses from the brain. Since these are non-invasive, only surface charges are picked up. These can be noisy, so correctly interpreting a user's intention can be difficult.

In general, users are given several tasks which involve different areas of the brain. For example, a visual or spatial problem should involve high activity (or strong electrical pulses) from the occipital lobe in the back of the head. Imagined movement, on the other hand, involves activity in the opposite hemisphere of the brain (left leg movement can be seen in the right hemisphere of the brain), as well as some frontal lobe activity. Tasks are selected to keep user fatigue to a minimum, users will generally find some tasks simpler than others, as well as attempting to keep EEG activity spatially separate. Having tasks spread out in different areas of the brain helps classifiers discriminate between different tasks, increasing accuracy. The tasks used in our application are: imagined left leg movement – which should be primarily seen in the right hemisphere, imagined right hand movement – primarily in the left hemisphere, a mathematical task – the frontal lobe, and 3D image manipulation – the occipital lobe. These tasks were chosen primarily for their distinct EEG patterns.

In the simplest case, a BCI application would ask a user to perform two separate tasks: one for confirmation, one for negation. While this is a fully-functioning interface, it can be quite tedious for more complex interactions, such as navigating a crowded hallway with a wheelchair. In these cases, more tasks are required of the user. In our BCI application, we are using data generated for a typing application. The application allows the user to select the next letter to type by subdividing the alphabet into quarters, and then further subdividing each selection until the user selects a single letter. While this is the original application our datasets were generated for, it is possible to take our generic backend and apply it to any application, which has been designed to handle four user tasks as inputs.

For this work, we are using an artificial neural network with Logistic Regression for classification tasks. Artificial neural networks, usually referred to as neural networks, are inspired by the interconnection of neurons in the brain. A basic neural network involves an input layer, a hidden layer, and an output layer. A very high-level overview can be seen in Figure 1. While the number of units in the input layer needs to match the dimensionality of the inputs, and the units in the output layer needs to match the dimensionality of the expected output, there can be any number of hidden units in the hidden layer. A general rule of thumb is that a larger number of hidden units can yield better results, but takes longer to train.

The hidden units are what drive a neural network. A neural network learns by modifying the parameters in this layer as well as the output layer. While larger numbers of hidden units can lead to a better trained network, it can also lead to over-specialization. A neural network may become too specialized to the training data, and cannot generalize well enough to handle new inputs.

In our implementation, the hidden units start with a very small, randomized weight which is applied to each input. Through the training process, these weights are modified to produce output closer to what is expected.

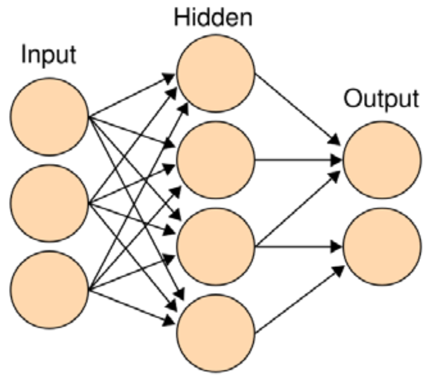


Figure 1. Simple Artificial Neural Network with one hidden layer, containing 4 hidden units

We are currently using an offline training method to train our neural networks. In offline training, we have a set of stored data consisting of sets inputs and their expected output. The neural network is trained by sending it input/output pairs, and finding the best set of weights for the hidden layer to minimize the error between actual and expected output.

The algorithms for training these neural networks and classifying EEG packets have been implemented in R. R is an interpreted language specialized for fast computation of matrix multiplications [6]. This makes it an ideal language for several types of computations, ranging from financial market analysis [7] to Bioinformatics applications [8], to Brain Computer Interfaces [1]. As an interpreted language, R allows for quick prototyping and testing. While interpreted languages are inherently not as fast as compiled code written in C or C++, R makes heavy use of C libraries in order to achieve fast matrix calculations – making it a better choice for computation-heavy applications, unlike other interpreted languages, such as Python.

III. Classifying EEG samples using Granules

The EEG processing algorithms have been developed in a language (R) that is different from the runtime's native language (java). Rather than re-implement these algorithms in Java, we decided to incorporate support for R computations in our runtime. Here, a challenge is to ensure that overheads introduced by bridging to R are acceptable. Since computations in Granules are activated when data is available on any of the input streams, an added consideration for the bridging scheme is to ensure that the activation overheads are acceptable.

To handle communication between Java and R, our Granules implementation uses the Java R Interface (JRI) package [9]. This interface allows an R session to be started and used through Java. With this, a Granules resource is able to start up an R session and train and classify through this session. This means that each resource can keep generated data (such as the trained neural networks) inside the R session, meaning less data needs to be passed across the network, as well as between Java and R.

The Granules-JRI bridge is lightweight, and does not maintain data structures that take up memory. Computation state is maintained within the Java and R computations, but not the bridge. Communications across this bridge are compact execution statements. This means that the minimal amount of data is contained in any message sent across the bridge. Instead of sending multiple commands through the Granules-JRI Bridge, we will start a complex command which will in turn perform the necessary series of operations.

The Granules runtime is started up on any machine that is expected to execute computations. The distribution of these computations is load balanced over the set of available machines. In our implementation, the processing is set up as a Map-Reduce [4] computation. Computations, both mappers and reducers, specify a scheduling strategy that activates them when data is available on any one of their input streams. Mappers produce their outputs as streams, and the reducer is configured to register an interest in outputs produced by individual mappers. The mappers are responsible for training and retaining a neural network. The first job of a mapper is to set up its bridge to R. Once the bridge has been set up, the mapper can train its neural network, and is then ready to handle incoming requests for classification by users. The mappers pass along any predictions to the reducer, which then makes a final prediction based on the results from all the mappers and sends the results back to the user. A high level overview of this process can be seen in Figure 2.

While there may be many mappers, there is only a single reducer. In Granules, computations can specify the data streams that they are interested in, and here each mapper has been subscribed to listen for EEG streams. Mappers are not attached to any single user, allowing a single Granules cloud to simultaneously handle streaming EEG data from multiple users. EEG data is streamed from a user, and read by each mapper. This data is not just a stream from a single electrode, but a combined stream of signals from all electrodes attached to the user. The combined EEG stream is analyzed by the neural network stored in the mapper. The neural network has been trained on data of the same dimensionality (the same number of electrode streams), and takes the full stream as input. Through the training process, the neural network has modified the weights, or importance of each separate stream, and developed a process for determining which type of signal the user is specifically trying to produce. The neural network uses this process to decide on the most likely classification based off of the current input, and will send this on as its prediction. The types of signals are predetermined, and cannot be modified once a neural network has been trained. Depending on the type of BCI application that the neural network is being trained to support, the network will be trained to recognize 2 or more specific signal patterns.

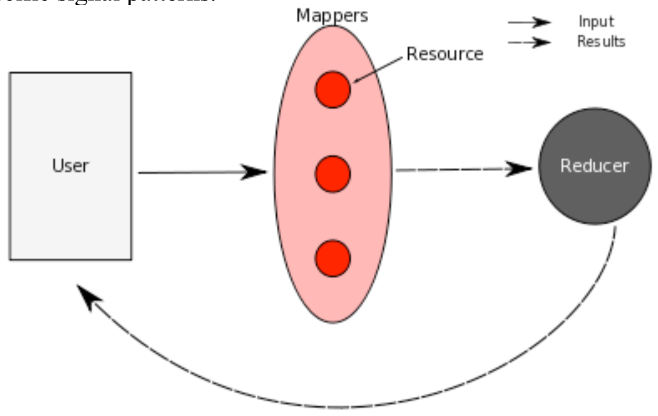


Figure 2. High level overview of Granules implementation

After a mapper has finished analyzing the stream, it will send a classification to the reducer. The reducer is responsible for gathering these classifications from all mappers, and determines the consensus classification by finding which classification was most often predicted by the mappers. Once the reducer has found a final classification, it then returns to the user the consensus classification. In the future, we wish to explore the benefits of training a neural network in the reducer. This would allow the reducer to learn which mappers gave better predictions, and learn to weight the classifications appropriately.

Our approach of multiple mappers and a single reducer allows us to cut short the training time for the neural networks, and still gain accuracy through a consensus. Any single neural network in this approach has not been trained exhaustively, so has a lower accuracy than a more thoroughly trained network. By allowing each mapper to independently train and keep a neural network, we can create a large number of these partially trained networks. Each network will learn a slightly different thing, meaning that together these networks can perform as well, if not better, than a single well-trained network.

Computations in Granules are capable of retaining state across successive executions, so the neural networks can be stored with each mapper and do not need to be retransmitted for testing new inputs. This cuts down on the amount of data that needs to be streamed across the cloud, and means that a larger neural network in the Granules environment would only add computation time on training, not on subsequent test runs.

IV. Using Snowfall

Snowfall [10] is an R package based on the Snow [11] package. Snow was one of the first R packages to allow programmers to distribute R code through a cloud without requiring a strong background in high performance computing. Snowfall builds on Snow and offers an easier interface, as well as efficient built-in load balancing. In our experiments, all calls with Snowfall are load-balanced, so if one node finishes computations before another slower machine, it will start running the other job as well, in order to finish in the most efficient manner possible.

Snowfall is designed so that programmers do not need to modify existing sequential code in order to distribute an application across a cloud. It operates along the same principle as the MapReduce paradigm – a computation and data are sent to different nodes, each node is responsible for performing the computation on the data sent to it, and all results are then returned to the source which distributed the computation.

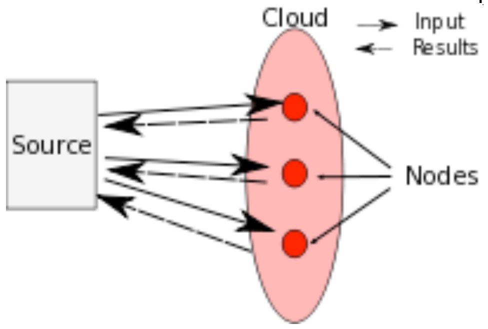


Figure 3. High level overview of Snowfall implementation

With Snowfall, a cluster is created from a list of potential resources (machines). Each resource listed is then checked for current load, and is added to the cluster if the load is light enough. After a cluster is established, it can be used for processing. Maintaining variables for later processing across cluster nodes is difficult, and can result in undefined results, particularly when a load-balanced approach is used. Because of this, subsequent testing of data with a trained neural network requires the trained neural network to be sent to cluster nodes along with the data to be processed.

A high level overview of the Snowfall implementation is shown in Figure 3. A source running R sets up a Snowfall cloud, with a specified set of nodes. The source then sends all data needed for the computation to each node, and each node then returns the results to the source.

As the purpose of this research is to evaluate streaming capabilities, as opposed to evaluating classification methods, we will not be checking classifications for accuracy. We instead focus on the amount of time off-line training takes as well as the time it takes to classify streaming data.

V. Experimental Setup

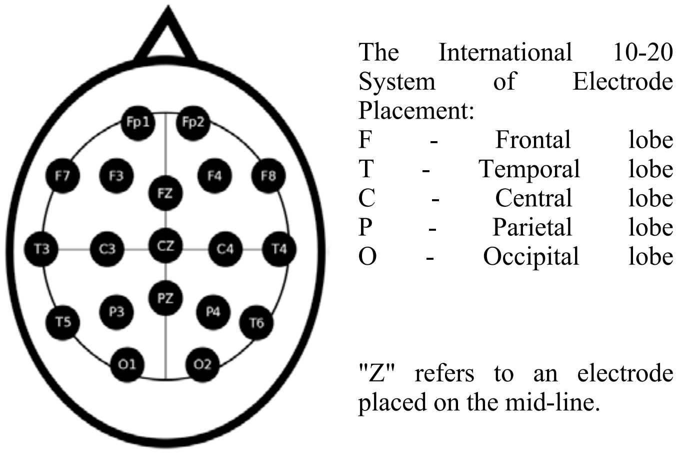
The machines (8-core Xeon, 16GB RAM) involved in the experiments were hosted on a 100 Mbps LAN. We used version 1.6.0_18 of the Java Virtual Machine, version 2.11.1 of R, and version 1.61 of Snowfall in our benchmarks.

In our experiments, both Snowfall and Granules contain the same R code to create, train, and use neural networks. Each, however, has a slightly different distributed backend. Additionally, each has a slightly different method of obtaining the pseudo-streamed data for classification, since Snowfall is not designed to handle streaming data. We first use an R script to read in EEG data from files, and pass this data to distributed environment for classification. This same script is used in both the Granules and Snowfall approach, though in Granules the data is then streamed to awaiting resources, while the Snowfall version simply pushes the data to the cluster.

A major difference between the Snowfall and Granules training approaches is where the initial training data is loaded. Snowfall reads in all the training data, and stores it as a local variable. This variable is then passed to all nodes in the network for training, using the function `sfExport()`. In the Granules version, each resource needs direct access to the training data, and is responsible for reading the data from file independently.

A. Experimental Data

The actual placement of electrodes followed the international 10-20 system of electrode placement, and is depicted in Figure 4. For these experiments, 19 channels were used: FP1, FP2, F3, Fz, F4, F7, F8, C3, Cz, C4, T3, T5, T4, T6, P3, Pz, P4, O1, and O2 from a Mindset EEG amplifier by Neuropulse-Systems (http://www.np-systems.com) with a sample rate of 256 Hz. The data was preprocessed before analysis – eye blinks and jaw clenches were filtered from the data, and the input has been further scrubbed by normalizing the data. This level of preprocessing is typical of datasets used for BCI applications.



The International 10-20 System of Electrode Placement:

F	-	Frontal	lobe
T	-	Temporal	lobe
C	-	Central	lobe
P	-	Parietal	lobe
O	-	Occipital	lobe

"Z" refers to an electrode placed on the mid-line.

Figure 4. The International 10-20 System of Electrode Placement

For classification purposes, four different tasks were attempted: imagined right hand movement, imagined left leg movement, counting backwards from 100 by 3, and imagining a spinning computer. These tasks involve increased activity in different areas of the brain, making them good candidate activities for analysis. Having four different tasks to classify allows a user to differentiate between four different options when using a BCI application. These options could include choosing the next letter to type, or the direction to move a wheelchair. All movements were imagined in order to keep with the premise of a primary user being unable to voluntarily control muscle movements. It is also possible that actual movements may introduce noise into any gathered data.

We trained the neural networks to classify various intervals of EEG data as one of these four tasks. The training sets are made up of 10 five-second sequences of each task. This data was gathered from a single user familiar with the tasks. Since the user who generated the data was familiar with the tasks, the data is clearer than an untrained user.

B. *Initializing the Networks*

Both the Granules and Snowfall based versions go through an initial training of neural networks. In the Snowfall implementation, a sequence of neural networks is returned - one for each resource trained. In the Granules version, each computation starts its own R session, and trained networks are held in session memory. For Granules, incoming data is first read in by a generator and then streamed to each resource for processing. This approach simulates real-time streaming which we may be able to expect from a person wearing an EEG cap. In the Snowfall version, Snowfall executes a LoadBalanced evaluation, where each resource receives a neural network and the set of data to be classified. Should one resource be running slower than others, the data originally sent to the slower machine can be resent to a faster machine to help cut down on overall execution time.

Each neural network is designed to use logistic regression for classification, contains 10 hidden units, and goes through a maximum of 400 iterations of scaled conjugate gradient descent [12] during the training process to optimize network weights, attempting to achieve a mean square root error (MSRE) of 0. The hidden units are randomly initialized with very small weights. This random weight initialization ensures that each network trained is slightly different from the others. This diversity is then used to help procure a joint classification based on the predictions of several small networks.

It is theoretically possible that the neural network will converge on a solution in less than 400 iterations, but due to the scale of the data being classified and the small number of hidden units it is highly unlikely that this will ever happen. During our tests, this situation never occurred, and the maximum number of iterations was always used.

The current implementation uses an offline method for training the artificial neural networks. All data used to train is on file, and the neural network is frozen once it finishes training – it does not continue to learn as data is classified. An area for future research is to explore the feasibility of on-line training, which would allow a neural network to continue learning as it is functioning. This approach would be able to fully leverage the benefits of multiple simultaneous users in real-time. After training has been completed, we use a reserved training set to test classification times.

To simulate streaming EEG signals in a way which both Granules and Snowfall can handle, we developed an R script to read a 5 second stream from file, split it if necessary, and send it out to be classified. With Granules, data is read in, and sent across the network to all resources. In the Snowfall version, data is read in from file – using the same R script as the Granules version – and is then dispatched by Snowfall to each node in the cluster along with a neural network to use for classification. In the Granules version, all classifications are then passed on to a node which then assesses the joint prediction and returns that result to the user. Snowfall returns a set of predictions, which are then passed into an R function on the local machine. This local function then returns the joint prediction.

C. *Data Processing*

For both training and test data sets, a lag of 3 frames is introduced into the data. This allows the artificial neural networks to see the previous 3 frames of the 19 channels of data, as well as the current incoming 19 channels. This approach gives the neural network more data to work with for every frame to be classified, and can improve accuracy. For these experiments, we kept the 3 frame lag constant but it is possible to vary this; however, once a network has been trained to use a particular lag, it will need to be retrained if a different lag is desired. The lag processing is handled by an R script which generates a pseudo-stream. We have included this processing in all of our timing, as we would be doing these computations on the fly in a live situation, as data was read in from the EEG cap.

VI. Performance Measurements

To fully grasp any lags inflicted by using Granules and JRI, we developed several tests. These tests are designed to bring to light any overheads incurred as the amount of nodes tested increases, as well as the amount of training data increases. For each test, we looked at both the original training cost, as well as the amount of time it took to get a full response from every node when various lengths of EEG data is streamed for testing. A Granules resource is configured with a set of worker threads. There are two components to the scheduling overheads introduced by Granules: activation and queuing delays. Activation delay is the time that elapses between the receipt of data over a stream, and the time that it takes to activate a dormant computation's finite state machine and have it ready for execution: this is typically in the order of 700-800 microseconds. Even though computations are activated, if the configured worker threads are busy the activated computations are queued till such time that a worker thread is available: queuing delays depend on the size of the queue when the computation was added and the average processing time per computation.

A. *Baseline Tests*

Before launching into the full suite of tests designed for assessing cloud applications, we first performed baseline tests. To get baseline values, we first ran a simple series of tests on a single machine, with a single training set. These tests use the base R code that is called in both our Snowfall and Granules implementations.

With these baseline tests, we should be able to isolate communication costs incurred in the move to a distributed setting, as well as find any idiosyncratic behavior in R. We looked at several measures, both directly reflected in the distributed tests – such as testing input streams – and indirect costs, such as loading training data. The baseline tests gather timing data for the following actions: loading a single training set, using the gathered data to train a neural network, and then classifying five seconds, one second, and 250ms of streamed data. Not only will this give us a baseline for each computation we will test later on in the distributed environments, but it also shows the basic cost of loading training files in R. This is a time inherent in all further tests, and also limits the capabilities of our current off-line training approach.

The results from these tests are shown in TABLE I. through TABLE III. For R to load and convert a 2MB EEG file, we have a constant overhead of roughly 6.5 seconds. While this is acceptable for our current training sets (we are using a maximum of 4, about 8MB on disk, which takes around half a minute to fully load), this could potentially be a bottleneck for future work, when we hope to leverage a cloud setting in order to amass training data from multiple users. These 2MB files contain 10 sets of 4 activities for 5 seconds apiece – about 200 seconds of EEG data. The baseline times to load a single dataset can be seen in TABLE I.

TABLE I. BASELINE TIME TO LOAD A SINGLE TRAINING DATASET (~ 2MB) IN MILLISECONDS

Mean (ms)	Min (ms)	Max (ms)	SD (ms)
6581.602	6439.742	6822.34	101.3716

We then looked at the amount of time it took to train a single neural network with one training set. Our results are shown below in TABLE II. . While this will be directly effected by the training set size, we should still be able to get an accurate read of this relationship in our subsequent runs. In this stand-alone case, it regularly takes over three minutes to train a single network. When this is done in a distributed setting, the training occurs in parallel, so we only see the amount of time for a single network to be trained even in an environment which has multiple networks.

TABLE II. BASELINE TIME TO TRAIN A NEURAL NETWORK FROM ONE TRAINING SET IN MILLISECONDS

Mean (ms)	Min (ms)	Max (ms)	SD (ms)
194463.7	192433.3	197094.9	1300.87

The last tests we ran as a baseline determined the base cost for classifying streaming data. The feasibility of using R as a real-time backend to BCI applications in a distributed environment hinges on these results. If, for example, it takes more than one second to classify one second of EEG data, it would be impossible to make the system function in real-time. For these tests, we looked at five seconds of data (the size from our training sets), one second, and 250 milliseconds of EEG data. The results of our experiments can be seen in TABLE III. From these results, we can see that an R backend to EEG classification is clearly possible – for every stream time tested, the processing time is less than 1% of the stream time.

TABLE III. BASELINE CLASSIFICATION TIMES FOR A SINGLE NEURAL NETWORK IN MILLISECONDS

Stream Time	Mean (ms)	Min (ms)	Max (ms)	SD (ms)
5s	23.0432	22.17889	23.56791	0.4734237
1s	5.28194	4.909039	11.16085	0.8568976
250ms	1.710529	1.673937	1.926184	0.03777157

B. Training Overhead

1) One Resource, One Training Set

For this test, a single resource was used, as well as a single training set (one set of 10 five-second sequences of each task). This is roughly 2MB on disk. This test is designed to get a baseline for the amount of time it takes to stream to a single resource using the smallest subset of our training data. As can be seen below in

TABLE IV. TRAINING A SINGLE NEURAL NETWORK IN A DISTRIBUTED SETTING WITH ONE TRAINING SET IN MILLISECONDS

Method	Mean (ms)	Min (ms)	Max (ms)	SD (ms)
Snowfall	409860.5	403364.3	419965.7	4216.875
Granules	313304.2	306402	329149	5141.031

2) Multiple Resources, One Training Set

This test is designed to bring to light any network latencies incurred by either approach. As the number of resources needing to be streamed to increases, we want to see if there is a noticeable difference in the time it takes to spread data across the network. For this experiment, we used three resources, and a single training set. Again, the training data is approximately 2MB on disk.

TABLE V. TRAINING 3 NEURAL NETWORKS IN A DISTRIBUTED SETTING WITH ONE TRAINING SET IN MILLISECONDS

Method	Mean (ms)	Min (ms)	Max (ms)	SD (ms)
Snowfall	462626.2	401475.9	483235.9	23512.64
Granules	675968.8	610550	772679	52823.21

3) One Resource, Multiple Training Sets

In this test we look at the cost of streaming data across the network, particularly in the case of initialization. For this test we used four sets of 10 five-second sequences of the four tasks. These datasets total to just about 8MB on disk.

TABLE VI. TRAINING A SINGLE NEURAL NETWORK IN A DISTRIBUTED SETTING WITH 4 TRAINING SETS IN MILLISECONDS

Method	Mean (ms)	Min (ms)	Max (ms)	SD (ms)
Snowfall	1001631	971224	1020680	17743.27
Granules	1933540	1782531	2057664	110686.7

4) Multiple Resources, Multiple Training Sets

This test explores the cost of sending multiple training sets to multiple resources, as well as any network lag from receiving results from each of the resources. We do not expect to see a noticeable difference between these results and when we looked at multiple resources and a single training set, but there is a possibility that the larger training set exacerbates any network lag from communicating with multiple resources. Again, the training sets are 8MB on disk.

TABLE VII. TRAINING 3 NEURAL NETWORKS IN A CLOUD WITH 4 TRAINING SETS IN MILLISECONDS

Method	Mean (ms)	Min (ms)	Max (ms)	SD (ms)
Snowfall	988410.4	964499	1023549	17513.37
Granules	1964255	1779853	2131574	136452.1

From these results, we see that Snowfall outperforms Granules in the initialization task of training neural networks.

C. Streaming Test data

This test focuses on the amount of time it takes to classify pseudo-streamed EEG data. A five second EEG stream is read from file, split into smaller time sets if needed, processed to introduce the 3 frame lag, and then sent across the network to the waiting resources/cluster in order to classify. All individual resources/nodes classify and then send their prediction on to a node responsible for producing a consensus classification. In the Granules example, we use a reducer for this, while the pure R example simply performs a post-processing step at the source which performs the same function. We are not interested in the quality of the classifications – our artificial neural networks are not very complex, and we are only using 3 separately trained networks – not enough to have a strong consensus. Instead, we are interested in the amount of time it takes to read the file, process the data, send it to be classified, and to tally together all the predictions to result in an expert classification.

In these tests, we look at data gathered from classifying the full 5 second streams, 1 second streams, and approximately 250ms streams. While the 5 second streams are larger than we would send in a typical BCI application, the 1 second and 250ms are closer to what we expect. By initially testing the 5 second streams, we can isolate any overhead generated by our pseudo-streaming application as it splits the 5 second streams into smaller streams, as well as find out if there is any preference for processing streams of the size the neural networks were initially trained with.

TABLE VIII. CLASSIFICATION TIMES IN MILLISECONDS

Method	Stream Time	Mean (ms)	Min (ms)	Max (ms)	SD (ms)
Snowfall	5s	8884.60	8797.745	9069.47	85.82
Granules	5s	141.69	136.42	266.63	12.75
Snowfall	1s	5825.71	5815.38	5857.98	10.09
Granules	1s	93.16	47.51	492.68	32.13
Snowfall	250ms	2831.32	2830.38	2849.83	2.03
Granules	250ms	87.25	48.57	92.67	4.49

These results demonstrate that Granules significantly outperforms Snowfall in the classification of EEG streams. More importantly, these results demonstrate that Granules can classify EEG streams in realtime.

D. *Granules Specific Tests*

In order to explore the limitations and capabilities of Granules, we ran several Granules specific tests. These tests are designed to simulate load from multiple users, and isolate any problems that may occur as a resource is overloaded. We ran a series of stress tests using a single resource on a single machine. In these cases, the resource was allowed access to 4, 6 and 8 cores. For each core available, we stressed the network by having multiple generators sending EEG streams of 250ms every 250ms. We attempted to ensure that no two generators were sending data in the exact same millisecond, as this would be highly unlikely in a live environment.

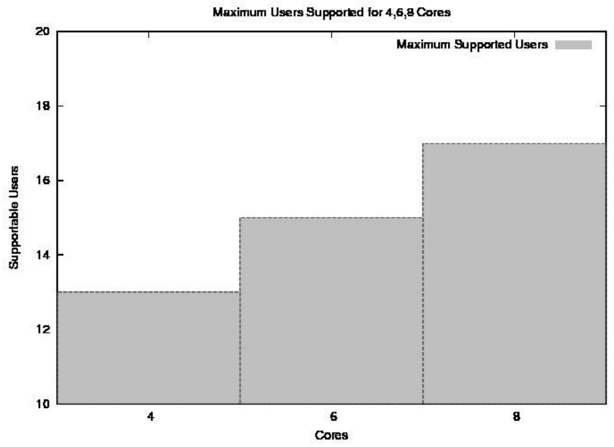


Figure 5. Maximum supported users for 4, 6, and 8 cores with Granules-JRI EEG classification

In each test, we sent about a minute of continuous streamed data from multiple generators. In these tests we increase the number of generators for a given number of cores until we hit the point where the resource was overloaded with requests. Once this happens, the resource cannot handle classification requests in an acceptable amount of time, classification results for 250ms of EEG data start taking longer than 250ms to return to the user. The results of these stress tests can be seen in Figure 5.

TABLE IX. shows the mean times for classification in our stress tests. All means are well below the maximum acceptable time (250ms), but through our tests we discovered that increasing the number of concurrently connected clients for any of the cores would result in classification requests taking too long to return.

TABLE IX. MEAN CLASSIFICATION TIMES IN STRESS TESTS ON A SINGLE MACHINE IN MILLISECONDS

4 Cores	6 Cores	8 Cores
66.1	55.29	51.9

We further stressed our setup by attempting to handle 75 and 150 concurrent users on 5 and 10 machines respectively. While our previous tests showed that we could support up to 17 users on a single 8 core machine, in these tests we decided to only have 15 users on a single 8 core machine. As each 250ms EEG signal results in about 20KB of data needing to be sent across the network, with 150 users on 10 machines, we are generating 12MB of data per second. With 75 users, we were generating half that, 6MB/s and overloading our network switch. For further stress tests, we needed to set up a temporary network of ten 8-core machines on a gigabyte switch.

The results of our tests are displayed in TABLE X. Across these tests, we achieved over 99.9% reliability. For 75 users, only 0.01% of the messages were over the 250ms threshold – 1 of every 10,000 messages was lost, or once every 41 minutes a user would lose a 250ms data packet. For 150 users, these statistics increased slightly: 0.04% of the messages were over the 250ms threshold: one of every 2,500 messages, or once every 10 minutes the response time was too slow. Due to this pattern, we believe that these losses may be a direct result of network overload, and we shall look into compression schemes in our future work.

TABLE X. STRESS RESULTS FOR 75 AND 150 CONCURRENT USERS ACROSS 10 MACHINES

	Mean (ms)	Min (ms)	Max (ms)	SD (ms)
75 Users	64.33	21.69	268.30	20.51
150 Users	69.81	22.01	352.82	22.49

From these results, we can clearly see that our approach is capable of scaling up to handle hundreds of users simultaneously. Our current bottleneck appears to be network bandwidth.

VII. RELATED WORK

As mentioned previously, we have relied heavily on the R package Snowfall [10] in this work. We are also relying on our cloud runtime, Granules [2, 3], for our distributed approach, along with JRI [9].

Apart from Snowfall, there are several other R libraries which focus on parallel computing. One of these is mapReduce[13], which is an R implementation of the MapReduce paradigm [4]. This library extends R's internal 'apply' function to handle mappings across multiple cores on a single machine. While users can develop their own apply function to act in parallel across multiple machines, this leads to an initial startup cost needed to convert sequential code. We would have additionally needed to rework the original R scripts to fit into the MapReduce paradigm, and would then have needed to use Snow or Snowfall to gain access to multiple machines. We believe that using Snowfall yields a closer comparison of cost and runtime with Granules.

Hadoop[14] is another Java-based cloud computing environment which supports the Map-Reduce paradigm. Hadoop has execute-once semantics, which precludes computations from executing more than once while retaining state. Hadoop also does not incorporate support for data streams as a data type; so, we would not be able to use Hadoop in a live situation, where multiple users can continuously use trained resources. The recent HadoopStreaming [15] utility allows users to use map and reduce code written in languages other than java, such as R and python. It allows Hadoop to operate upon data from the command line, but it still relies on run-once semantics, without support for state retention, dormant computations, or datastreams.

Other frameworks which support the Map-Reduce paradigm include Phoenix [16], QtConcurrent [17] and Skynet [18]. While these all support the paradigm, like Hadoop they only support run-once semantics. They have also been designed to operate on static files, and not dynamic data streams.

There are several machine learning approaches which have made use of a distributed environment. Aside from forming consensus networks, as we have in our work here, there are swarm approaches such as ant-colony[19]. These approaches make use of multiple agents, which are responsible for learning about their portion of data. While there is a history of exploring the use of multiple agents in a distributed environment [20], to the best of our knowledge this is the first attempt at classifying EEG signals from multiple users using cloud computing techniques.

VIII. Conclusions and Future Work

While there is a definite overhead seen when using Granules for training a cloud, the amount of time it takes to test various sizes of EEG data streams shows that Granules can actually classify significantly more efficiently. As the primary bottleneck for real-time EEG signal processing is in the actual classification, as opposed to the training of a network, we do not see the training overhead as an impediment to further research into this area. Snowfall is not designed to handle streaming data, so it is understandable that it does not handle streaming data as well as Granules, which was designed specifically for that task. We also found that using JRI to handle R calls from Java does not incur an inhibitive overhead to the computations. While we are currently using JRI to interface with R computations, there are two approaches that we will explore to optimize the interface to R computations. First, we will explore the use of sockets for communications. Another viable approach would be the use of chaining, where we interface with R via C++; in previous experiments, we have found that our Java-C++ bridge introduces a communication overhead of around 1 millisecond, so this approach could be promising.

In the future, we plan to move from our pseudo-streams of EEG signals we used here to simulate streaming data to real-time streaming data from an EEG cap.

There are also several plans in place to modify the basic operation of the neural networks. In the current implementation, there is no way to retrain a network if we want to change the lag level except to restart the clusters. Our future investigation will focus on allowing a user to change this value while the system is running.

Another change would be to implement an online training method – this would allow the networks to continue to learn and refine as more data came in. While this would add to the amount of computations generated by any interaction with a user, we believe that the potential increases in accuracy as well as removing the need to take a network offline and potentially spend hours to retraining would prove to balance out this cost.

References

[1] F. Galan, et al., "A brain-actuated wheelchair: Asynchronous and non-invasive Brain-computer interfaces for continuous control of robots," Clinical Neurophysiology, vol. 119, pp. 2159-2169, 2008.

[2] S. Pallickara, et al., "Granules: A Lightweight, Streaming Runtime for Cloud Computing With Support for Map-Reduce," in IEEE International Conference on Cluster Computing, New Orleans, LA., 2009.

[3] S. Pallickara, et al., "An Overview of the Granules Runtime for Cloud Computing," in IEEE International Conference on e-Science, Indianapolis, 2008.

[4] J. Dean and S. Ghemawat, "Mapreduce: Simplified data processing on large clusters," ACM Commun., vol. 51, pp. 107-113, Jan. 2008 2008.

[5] M. Isard, et al., "Dryad: distributed data-parallel programs from sequential building blocks," in 2nd ACM SIGOPS/EuroSys European Conference on Computer Systems, Lisbon, Poutugal, 2007.

[6] (2010). The R Project for Statistical Computing. Available: http://www.r-project.org

[7] (2010). Rmetrics. Available: https://www.rmetrics.org/

[8] Bioconductor: Open Source Software for Bioinformatics. Available: http://www.biocunductor.org/

[9] "JRI - Java/R Interface," 0.5-0 ed, 2009.

[10] J. Knaus, "snowfall: Easier cluster computing (based on snow)," 1.84 ed, 2010.

[11] L. Tierney, et al., "SNOW: Simple Network of Workstations," ed, 2009.

[12] M. F. Møller, "A scaled conjugate gradient algorithm for fast supervised learning," Neural Networks, vol. 6, pp. 525-533, 1993.

[13] C. Brown, "mapReduce: flexible mapReduce algorithm for parallel computation," 1.02 ed, 2009.

[14] T. White, Hadoop: The Definitive Guide, 1 ed.: O'Reilly Media, 2009.

[15] D. S. Rosenberg, "HadoopStreaming: Utilities for using R scripts in Hadoop streaming," 0.2 ed, 2010.

[16] C. Ranger, et al., "Evaluating MapReduce for Multi-core and Multiprocessor Systems," presented at the IEEE HPCA-13: 13th International Symposium on High-Performance Computer Architecture, Phoenix, Arizona, 2007.

[17] "Qt Concurrent," 0.1.2 ed, 2010, p. Simplified MapReduce in C++ with support for multicores.

[18] A. Pisoni, "Skynet: A Ruby MapReduce Framework," 0.9.3 ed, 2010.

[19] M. Dorigo and L. Gambardella, "Ant colony system: a cooperative learning approach to the traveling salesman problem," IEEE Transactions on Evolutionary Computation, vol. 1, pp. 53-66, Apr 1997 1997.

[20] G. Weiß, "A Multiagent Perspective of Parallel and Distributed Machine Learning," in 2nd International Conference on Autonomous Agents, 1998, pp. 226-230.