II

Building artificial Neurons in three-dimensional space

# Introduction

We have discussed the possibility of three-dimensional mesh neural networks and the potential for artificial neurons. In doing so we have touched on some issues we may encounter in larger mesh networks and security issues whilst developing and designing such functionality.

During the course of the second iteration, we will be building neurons and deploying artificial intelligence within their core to called as a call-back function. The artificial intelligence will look at deploying and making use of MST, logging, reporting and providing return information for analysis.

We will deploy a 32 neuron setup with various neuron attributes and make use of a simple dictionary data set with specific output requirements. These neurons will simply be housed within a mathematical bounded three-dimensional space and have all of their respective connectivities in place either hard coded or autonomous if we deploy them at random space points within the volume.

In doing so we will also look at the dormant neurons within the volume and research a best practice to randomise with a little more even distribution in a volume whilst still maintaining ‘organic’ distribution. We will make use of standardised algorithms to assess their functionality and examine efficiency of both deployment at random compared to hard coded and in comparison, to existing results as if the artificial intelligence algorithm were simply a single entity.

With that in mind, we will perform those tasks to examine whether artificially, more neurons are better than one before proceeding to examine whether all neurons should have the same artificial intelligence or whether clusters of variable artificial intelligence can interact well.

They will be built in a shared folder on a guest virtual machine operating system which has had the virtual networking card stripped out after the updates and the necessary IDE installed. For now the development process will be localised on the machine and not run until completed.

The purpose of the second iteration is to refute the Spencer-Harper [2015] statement “It’s not necessary to model the biological complexity of the human brain at a molecular level, just its higher level rules”. Because sometimes, due to the complexity of the brain (processing several interpretations [senses] of the environment), it is necessary to compile the individual processing cortexes (volumes) to replicate and manage tasks in replication to the functionality in opposition to masses of algebraic equations and have them ‘talk’ to each other.

# Volume

When setting the volume metrics, we have two options; We can set mathematical bounds such as ten cubed giving us 1000 dots matrices to work with or we can use three dimensional ArrayLists which tend to get a bit cumbersome when trying to find out which matrix location each neuron has. In addition, each matrix location will be housing another multi-dimensional ArrayList and makes for very difficult decompression and confusion in the programmatic section.

What will be best for the volume is to set perhaps a simple mathematical representation yielding mathematical volumetrics and better precision in neural proximities. Furthermore, scalability will be a lot easier to manage when embedding to the hardware. What we end up with are mathematical boundaries and neural proximities pinpointing programmatic memory locations at their neural epicentre.

public static ArrayList<ArrayList<ArrayList<Object>>> three\_d\_list(int n) {

int i, j;

int x\_axis = n;

int y\_axis = n;

int z\_axis = n;

ArrayList<ArrayList<ArrayList<Object>>> c = new ArrayList<>(x\_axis);

for (i = 0; i <= x\_axis; i++) {

c.add(new ArrayList<ArrayList<Object>>(z\_axis));

for (j = 0; j < z\_axis; j++) {

c.get(i).add(new ArrayList<Object>(y\_axis));

return c;}

The issue we will encounter is programmatically indexing the mesh layers and ensuring the correct depth, though, it may be possible to simply set integer type precision for the volume and float data types for the neural proximities permitting better precision when connectivity is assessed.

When building the volume, we can make use of the base code from the first iteration and proof of concept. Each mesh is a layer index Y with a square surface area of Z multiplied by X and each can be stacked upon the other before the connectivity bonds are called.

static Map<Integer, ArrayList<Object>> *map* = new HashMap<>();

static ArrayList<ArrayList<ArrayList<Object>>> *c* = new ArrayList<>();

Doing so, permits the mesh to be generated, layer by layer should speed up the randomisation of neuron placement on the ZX plane, the connections will inevitably take longer as they are generated, compressed and stored per neuron and in a file. We could then use a Hash Mapping function from the Java API to map the ArrayList indices to an indexed location thus streamlining access to memory locations in the Mapped Index and have an iterable index of the volume.

*@SuppressWarnings*("unchecked")

public static Map<Integer, ArrayList<Object>> mapping(ArrayList<Object> neuron, int qty, int n) {

int j, k, l, i;

double v = Math.*pow*(n, 3);

ArrayList<ArrayList<ArrayList<Object>>> c = *three\_d\_list*(n);

for (j = 0; j < n; j++) {

for (k = 0; k < n; k++) {

for(l = 0; l < n; l++) {

c.get(j).get(k).add(l);

for (i = 0; i < v; i++) {

*map*.put(i, (ArrayList<Object>) c.get(j).get(k).get(l));

}}}}

*populate*(neuron, qty, v);

return *map*;}

## Volume – Batch and Variance

It would be a good idea to ensure the generation, placements and connections are stored in a file for analysis and regeneration. We have to make certain during build and test stages the data is stored because if an optimum is found, we can revert back to it for hard coding and we can run a batch optimisation from neurological mean average configuration to mean average result comparison and later check the neurological pattern against the expected result.

As such, we may be able to ascertain if there is a generic configuration which yields the optimal results. There are so many possible configurations of three-dimensional neural mesh, with such variance of artificial intelligences, we would have to run such a practice for the high-performance end of the scale too. More on those topics in further discussion, since we have digressed and should turn focus back to Batch and Variance more specifically populating the Indices, accessing them and activating them.

## Volume – Neurons and Training

With a focus on the type of neuron we are adding to the volume, for the example herein, we are simply making use of one’s and zero’s as the training data and deploy a situation to check if the situation fits in with he training data. The training data is an multi-dimensional ArrayList Object of sets of five one’s and zero’s per the original training template from Spencer-Harper [2015].

However, what we have tried to achieve in the modified version and port over to java is to make use of the object orientated nature of the language and even make use of objectifying data to alleviate the necessity to reprogram areas of the system to suite certain datatypes, though that may still be a nice requirement in future iterations.

The porting is quite difficult since we are often making use of ArrayLists and so modifications to standardised mathematical algorithms had to be made, two prime examples were the transpose function -

public static ArrayList<Object> transpose (ArrayList<Object> arrayList) throws IOException {

int i, j;

Logger\_Writer.*setTranspose\_before*(arrayList);

Logger\_Writer.*Logger\_Printer*(*PrinterState*.***TRANSB4***);

ArrayList<Object> transpose = new ArrayList<>();

int count = arrayList.size();

for (i = 0; i < count; i++) {

ArrayList<Object> temp = new ArrayList<>();

temp.add(arrayList.get(i));

for (j = 0; j < temp.size(); j++) {

ArrayList<Object> transpose\_row = new ArrayList<>();

transpose\_row.add(temp.get(i));

transpose.set(j, transpose\_row.get(0));

transpose\_row.clear();}}

Logger\_Writer.*setTranspose\_after*(transpose);

Logger\_Writer.*Logger\_Printer*(*PrinterState*.***TRANSFTR***);

return transpose;}

and the dot product function detailed below.

public static ArrayList<Object> dot (ArrayList<Object> in, ArrayList<Object> out) {

ArrayList<Object> outputs = new ArrayList<>();

for(int q = 0; q < in.size(); q++) {

float inputsI = (float) in.get(q);

for(int r = 0; r < out.size(); r++) {

float weights = (float) out.get(r);

outputs.add(r, inputsI \*= weights);}}

return outputs;

}}

The nature of the training module has been modified so far to accept only the training data to alleviate the necessity to provide other arguments to the function making it easier to activate the neuron later. We do that because it becomes increasingly easier to call the functions within the volume. After all, we are attempting to send the synaptic data as the input to other neurons effectively. What we really need is to modify the training function (neuron) to call the training data from within and accept only synaptic data as an argument and as an output once trained.

public static float training(ArrayList<Object> training\_set) throws IOException {

//**TODO**: Add timers

//**TODO**: Call to connection checks for I/O

Object a, b;

int sum = 0;

float a\_sum = 0;

int t, e, u, r, h, k, q;

float inputsY, sigdiv, inputsN, inputsA;

int t\_qty = Simple\_Neural\_Network.*getT\_qty*();

float synaptic = Simple\_Neural\_Network.*getSynaptics*();

ArrayList<Object> error = new ArrayList<>();

ArrayList<Object> mplex = new ArrayList<>();

ArrayList<Object> adjustment = new ArrayList<>();

ArrayList<Object> loaded = new ArrayList<>(Situation.*situation*());

ArrayList<Object> training = new ArrayList<>(training\_set);

ArrayList<Object> output = Think.*think*(training);

ArrayList<Object> sds = new ArrayList<>(Sigmoid\_derivative.*sigmoid\_derivative*(output));

for (t = 0; t <= t\_qty; t++) {

//here is where we get the think() to consider the float synaptic value as it changes

Logger\_Writer.*Logger\_Generic*("Beginning to assess the nucleus output in the think method: " + synaptic + "\n");

for (e = 0; e < output.size(); e++) {

a = loaded.get(e);

b = output.get(e);

if (a == b) {

sum += 1;}}

error.add(((output.size()-1 - sum) / 100) \* output.size()-1);

Logger\_Writer.*setErrors*(error);

Logger\_Writer.*Logger\_Printer*(*PrinterState*.***ERRORS***);}

sum = 0;

//Get sigmoid derivatives of the errors

for(q = 0; q < error.size(); q++) {

inputsY = (float) error.get(q);

for(r = 0; r < sds.size(); r++) {

sigdiv = (float) sds.get(r);

mplex.add(r, inputsY \*= sigdiv);}}

Logger\_Writer.*Logger\_Generic*("The sigmoid derivatives are: " + sds + "\n");

sds.clear();

error.clear();

//Get a dot product of the matrix transposition and the error multiples of the sigmoid derivatives

for (h = 0; h <= output.size(); t++) {adjustment.add(Dot.*dot*(Transpose.*transpose*(training), mplex));}

Logger\_Writer.*Logger\_Generic*("The multiples of the errors to sigmoid derivatives are: " + mplex + "\n");

mplex.clear();

//Take the adjustment values and summise them ready for mean average

for(k = 0; k < adjustment.size(); k++) {

inputsN = (float) adjustment.get(k);

for(u = 0; u < adjustment.size(); u++) {

inputsA = inputsN;

a\_sum += inputsA;}}

Logger\_Writer.*setAdjustment*(a\_sum);

Logger\_Writer.*Logger\_Printer*(*PrinterState*.***ADJUSTMENT***);

a\_sum /= adjustment.size();

Logger\_Writer.*setAdjustment*(a\_sum);

Logger\_Writer.*Logger\_Printer*(*PrinterState*.***ADJUSTMENT***);

//Add the mean average adjustment to the synaptic value

synaptic += a\_sum;

a\_sum = 0;

Logger\_Writer.*Logger\_Generic*("Final synaptic output from nucleus is: " + synaptic + "\n");

System.***err***.println("New Synaptics: " + synaptic + "\n");

//**TODO**: Making a synaptical load and output Axiom to other neurons

return synaptic;}

What we have is the training module being a part of the thinking aspect of the neuron which is deployed inside the volume at random when called. As you can see from our training module (nucleus) we can run call backs to other math functions and operations for better processing.

## Volume - Neuron

public static ArrayList<Object> neuron (ArrayList<Object> training\_set) throws IOException {

ArrayList<Object> neuron = new ArrayList<>();

neuron.add(Training.*training*(training\_set));

neuron.add(Touch.*t*());

neuron.add(Green.*g*());

neuron.add(Red.*r*());

return neuron;}}

## Volume - Mapping

Since we have the training module as part of the nucleus and call the training set directly from perhaps a plugin method. We could possibly change the nucleus all together simply by using a stack of differing artificial intelligences to be deployed in the nucleus. The deployment is simply pushed in by allocating the neuron to the mapping function with the correct arguments such as the cubic size of the volume and the neuron quantity to accompany the mapping functionality in cubic mapping class.

public static Map<Integer, ArrayList<Object>> mapped\_volume (int neuron\_qty, int vol) throws IOException {

Logger\_Writer.*Logger\_Generic*("Generating NeurologicalMap of Neurons...\n");

return Cubic\_Mapping\_4.*mapping*(Neuron.*neuron*(Training\_Set.*training\_set*()), neuron\_qty, vol); }

and mapped using the populate function below.

public static void populate(ArrayList<Object> neuron, int qty, double v) {

Random rand = new Random();

int i;

for (i = 0; i <= qty; i++) { *map*.put(rand.nextInt((int) v), neuron);

System.***err***.print("Original mapping after first generation: " + *map*.hashCode());

Logger\_Writer.*Logger\_Generic*("Original mapping after first generation: " + *map*.hashCode());}}

# Gathering

We are working on artificial neurological systems and thus, it would be nice if we could see what the individual neurons are doing and even what the randomised structure of the volume looks like when neurons have been added to the volume.

From that data collection we can assimilate optimums once data analysis has occurred, simply saying 100,000,000,000 neurons processing n = n\*(x + y) / (pow(n, 4)) \* n is a little dull because in essence neurons are adjusting their sigmoid functionality because of the nature of bioelectrical and biochemical stimulus affecting them. As such each cortex of the brain, performs a slightly different job and varies slightly in individuals and species (need some references).

To gather the data required to assess what constitutes to a nano sample of cells we need both logs piped out to the error console since that data is collected intrinsically and stored locally, in addition to file storage for multiple append calls to the file to gather sets of data for analysis. The collection process also enables timers to be written in later for benchmarking (though write outs to file, error logs and timers will slow the performance somewhat).

## Gathering – Data and Logs

When gathering data we are going to need to autonomously check and or create our file and to access it for writing data. In addition, what data do we collect? What is important? Well, we already discussed the mapped volume and its randomised configuration.

We would want the connections detailed to assess whether dormant neurons were created (too far away from any proximities) and therefore not sending or receiving data to or from other neurons. We would also want to know what data was being sent between the neurons, what the synaptic weights were for firing off to the surrounding connected neurons.

We may even want to simply gather reports of what is happening during the process of running the tests including benchmarking. These pieces of data are valuable during the testing stages and phases. Moreover, which tools would be the simplest way to gather such complex data?

The inbuilt system input output stream will be best suited for the error outputs since they are already built into and accessible by our IDE. However, the data file storage read and write will need to be automated somewhat.

## Gathering – Finite State Machine Triggers

In doing so and in keeping with the autonomous collection of varying data, how best to do that? Well, there is the finite state machine. The finite state machine can be used to change the state of a variable and trigger a response of some kind in a separate method or class for dealing with data change.

public enum *PrinterState* {

***VOID***,

***CONN***,

***TOUCH***,

***GREEN***,

***RED***,

***MAVOL***,

***SYNAP***,

***TRANSB4***,

***TRANSFTR***,

***SITUATION***,

***ADJUSTMENT***,

***TRAINSET***,

***ERRORS***,

***TIME***;

}

In our example, the change is a switch statement after a variable has been set to print the variable’ data to file. We set the new variable inside the printer class using encapsulation,

error.add(((output.size()-1 - sum) / 100) \* output.size()-1);

Logger\_Writer.*setErrors*(error);

From there we call to trigger the state change,

Logger\_Writer.Logger\_Printer(PrinterState.ERRORS);

And in the printer module of the Logger Writer class the event is triggered causing the variable to be printed to file,

case ***ERRORS***:

try {

*Logger\_Checker*();

stack.write("Errors before computation of synapse:\n" + Logger\_Writer.*getErrors*() + "\n");}

catch (IOException e) {

System.***err***.println("An error occurred." + e);

e.printStackTrace ();

}

break;

1. Spencer-Harper., M. 2015. How to build a simple neural network in 9 lines of Python code [online] Available from: <https://medium.com/technology-invention-and-more/how-to-build-a-simple-neural-network-in-9-lines-of-python-code-cc8f23647ca1> [29/11/2022].