Derek Strasters

In Depth Documentations of a

Spring 2015

Neural Net Architecture With Demo Application

Author:
Derek Strasters

September 30, 2015

Contents

1 Introduction to this Neural Network Architecture **3**

1.1 Preface 3

1.2 An Abstract Definition 3

1.3 Backpropagation by Stochastic Gradient Descent 4

1.3.1 The Sigmoid Neuron 5

2 Header File — `Neuron.h` **8**

2.1 Instantiable — `class Neuron< >()` 8

2.1.1 Member Function — `FuncSig GetActivationFct()` 9

2.1.2 Member Function — `GetActivationFctPrime()` 9

3 Header File — `Utility.h` **9**

3.1 Instantiable — `typedef double (*FuncSig)(double z)` 10

3.2 Instantiable — `class DownCast< >()` 10

3.3 Instantiable — `struct CommAry< >()` 10

3.4 Instantiable — `CommAry's` Printing Tools and Math Tools 10

3.5 Instantiable — `class RandTools()` 11

4 Source File — `Utility.cpp` **11**

4.1 Member Functions of — `class RandTools()` 11

5 Header File — `DLog.h` **11**

5.1 Instantiable `struct dbglog()` 12

6 Header File — `Color.h` **12**

7 Header File — `Layer.h` **12**

7.1 Instantiable `class Layer< >()` 12

7.1.1 Member Function `void Z_Cascade()` 12

7.1.2 Member Function `void Activation()` 12
7.1.3 Member Function `void ErrorCascade()` 13
7.1.4 Member Function `void DescendGradiant()` 13

8 Header File — `Perceptron.h` and `Sigmoid.h` 13

9 Header File — `POSI.h` 13

10 Source File — `Alan.cpp` 13

1 Introduction to this Neural Network Architecture

1.1 Preface

The Neural Net Architecture being presented here is built with the intention of providing other programmers with a broad set of tools for the creation of multipurpose Neural Networks. To complete the documentation, there is within the project directory a source file named `Alan.cpp` (honouring one of the fathers of computer science, Alan Turing). Alan, compiles compiles into an executable interactive demonstration that is meant to offer somewhat of a visual observation on the inner workings of a neural network. The demonstration shows just one of the most simple examples of the many exciting ways in which a user might make use of an architecture such as this to implement a Neural Network. It is my hope to continue to improve the user-friendliness, efficiency, and open-endedness with such goals as implementing the OpenGL library to make use of the powerful video cards in most home computers, and full cross platform windowed user interface. There truly are innumerable possibilities for the uses of neural networks, and I think it would be just neat if we could make them as readily available to the world as perhaps an on-line calculator is now. This has been the main driving factor behind my attempt at a design that both - leaves room for users to easily make modifications in places that are likely to be experimented with and tuned, while also maintaining a reasonably small number of easy to use classes and functions to handle the heavy computing. All of which have been designed with the intent to make it difficult for the user to introduce bugs by accident.

1.2 An Abstract Definition

Before continuing, a brief explanations of what a neural net actually is and does, are in order. My own source of understanding Neural Networks has come largely from the work of Michael Nielsen on his *living* web book wherein he does a stand up job of introducing the structure of the neural network [2]. I recommend his work to any interested readers in want of more information about neural networks. An interesting side point about his book, is that at the time of my writing this, it is not even finished. Being a living book, he is publishing it incrementally as he goes.

A neural network or, neural net, is so named because of it's (albeit somewhat crude) semblance to the manner in which a brain's network of neurons function. It is not, however, an identical analog by any means. In the shortest way of stating a neural network's function: it is a system that is capable of "learning" how to behave. For example, in the demo implementation, a network is constructed and then "shown" some information in the form of a string of numbers. At the same time, the network is also "shown" what outcome we want it to produce, and after we do this many times, we stop the so called training and can then present the network a value it has never seen and hopefully it will have learned to respond in the way we have taught it. Note the use of the word "shown". This is part of a pattern of cute brain analogies that seems to be endemic to the field, as we shall see soon the array that does the "seeing" is typically named the retina, after our own high speed arrays of information throughput pipelines located at the backs of our eyeballs.

Now, the idea of a machine that learns to respond to number sequences with a list of other "learned" number sequences may sound at the surface like a really inefficient way of doing nothing more complicated than regurgitating values from some kind of look up table of recorded values. The thing that makes this type of system so much more special than trivial behaviour of a look up table is that you can then show the network some data it has never been exposed to before and if that data follows some pattern, even a pattern that we ourselves are unaware of, it is very likely going to output the correct response to that new data. It's a matter of interpretation, but this student finds such behavior to be moving the idea of a computer that could arguably said to "think" is arriving in a much ***less*** distant future.

1.3 Backpropagation by Stochastic Gradient Descent

I will go into greater detail on the specifics of my design later, but for now the idea behind a neural net is that a set of items, call them neurons, are linked together in a network in the sense that a neuron may read data from other neurons that it is linked to. These neurons have the capability of reading in multiple numerical values (typically values between 0 and 1) as impetus for generating a singular output value. For now I will just say that the neuron simply does ***something*** with those values to generate it's output. It is unsurprising that most of the neurons will be taking the outputs of other many other neurons as input. Knowing nothing more than what has been stated - *. . . a semblance to the manner in which a brain's network of neurons work,* - and that each neuron accepts the outputs from many other neurons, it then seems reasonable to expect that each neuron has a way in which to grant greater weight to some inputs and less weight to others. The way in which this concept is implemented in this type of Neural Network is for each neuron to keep a simple scalar for each of it's inputs. These scalars are aptly called the neuron's weights. To use the brain analogy, the neuron's **weights** strengthen or weaken the synapses between other neurons. This implies the need for some feedback mechanism capable of modifying the weights to produce the desired behavior of the network. Adding to the idea of weights, each neuron also keeps a singular value that acts as a scalar to it's output; this is called the **bias** and must also subject to change by our feedback mechanism. We will soon see that it is useful to define an equation for the sum of weighted inputs to which the bias is added, a value that is often represented as Z.

$$Z \equiv \left(\sum_{j=0}^{N} w_j a_j \right) + b \tag{1}$$

Where w is a weight, a_j is an input values, N the total number of inputs, and b the bias.

This looks like it could be that ***something*** I had referred to earlier that the neurons are doing with their input data, but in our case, it is not – although it is part of it. To elicit Boolean logic type behavior (not to say that this is the best behavior we could want), we should desired that we have an output that behaves somewhat like a switch, since that is what is going to dive any logic type computations that develop in the network but, as we will see, we also want a smooth function that describes our output. This comes back to our feedback mechanism for changing the weights and biases of the system, a method called ***Backpropagation***.

1.3.1 The Sigmoid Neuron

Before discussing Backpropagation we will need to look at the underlying concept of **Gradient Descent**. As one can imagine with any appreciable amount of neurons in a network we are going to have a very large amount of variables that will need to be "tuned" through a training algorithm to make our system behave as desired. For a network of the type discussed here containing 9 neurons there could be up to 36 such variables, and a network of 25 could have up to 150. Remember, what we are looking for is a best value for how well the system is producing expected results, a measure we shall now define as **cost**. To optimize cost over such a large set of variables one might first think to use a evolutionary or genetic algorithm, which wouldn't be a bad place to start, but a more direct route would certainly be desirable in terms of the time it takes to find an optimal solution. The more efficient strategy used in our system is best described as treating each variable as a dimension, in the same sense that we move about in our 3 dimensions. An algorithm then estimates the gradient of the cost over these dimensions with the values thereof pointing to a local or hopefully global minima.

To liken this to our 3 dimensional reality again, it would be as though one were standing on a mountain, altitude being a measure of cost in this case, and looking at the slope of the ground near your feet to determine how to get to the bottom of the mountain. I rather like this analogy because it elucidates a similar pitfall experienced by the neural network, if our mountain climber judges the slope by looking too near his feet, he may find himself eventually stuck in a ditch or crevasse before reaching bottom. If he judges the slope by looking too far into the distance, he may end up traveling from mountain to mountain aimlessly never realizing if he were for a moment in the lowest spot. Enough about mountain men though, we should finish defining the behaviour of the neurons of this system. The reason Gradient Descent is important to how our neurons work is that if we were to use a step function (See **On Perceptrons**) there would be no mountains, only cliffs. We need a function that is smooth such that we can calculate a gradient therefrom. Our neurons implement what is called an ***activation*** function to do this, and there are many different types, but they more or less have the same features – they are continuous, smooth, and they act Boolean-like at their extremities. What I mean by that is that it is desired that our neurons have an output that behaves somewhat like a switch, since that is what is going to help logic type computations that develop in the network. The activation function we apply what is called the **Sigmoid** function $\boldsymbol{\sigma}$. This function is applied directly to the Z value as shown in the following definition:

On Perceptrons

In the earliest neural networks, equation 1 was used to determine a binary neuron output by the simple rule:

$$a = \begin{cases} a = 0 & : Z < 0 \\ a = 1 & : Z \geq 1 \end{cases}$$

Neurons of this type are called Perceptrons. They were a revolutionary idea developed by Frank Rosenblatt [4] in 1957. There are however limitations in Perceptrons' ability to learn, a point which newer systems seek to address with the methods found within this text.

$$\sigma(z) \equiv \frac{1}{1 + e^{-z}} \tag{2}$$

Definition of the Sigmoid function.

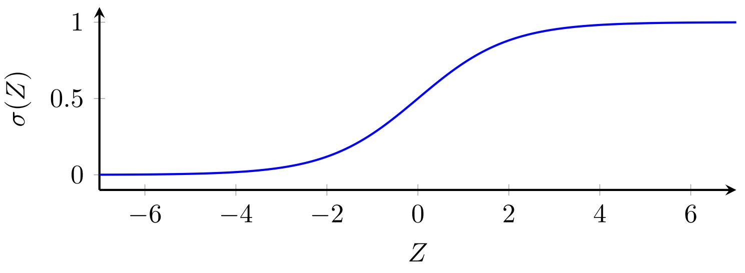

Figure 1: A plot of the Sigmoid function

START HERE There is still the problem of figuring out by what measure we want to determine how "good" are neurons are behaving and whether to strengthen or weaken their bonds. Well I'm going to jump ahead a little by just stating that I'm describing what is called the Cost Function of our network and it doesn't actually have a hard Type. What I mean by that is there are a number of ways of measuring how "good" the output of the system is when compared against training data. It is good for our purposes you use the somewhat natural intuition of using the mean squared error of the output against the expected output. It turns out to be *very* good for our puropses, because when we take the derivative (which we'll end up needing), we get a simple difference calculate that is very easy to process.

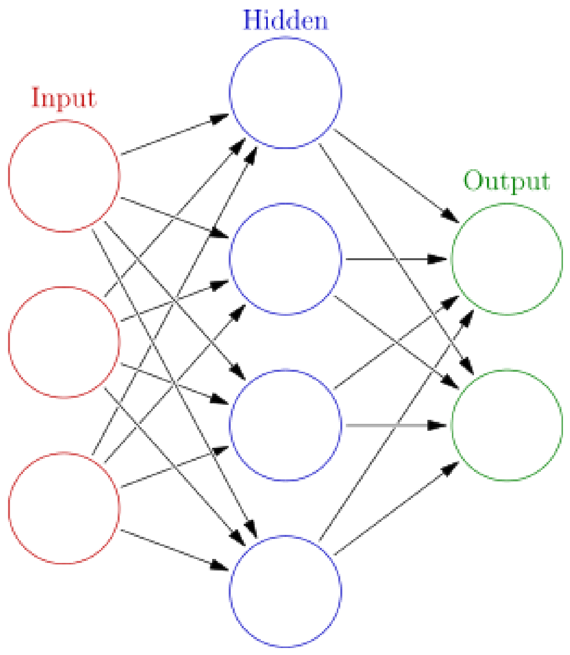

Figure 2: Artificial neural network with layer coloring [1]. [1]

[1]Glosser.ca. ***Artificial neural network with layer coloring***. Feb. 28, 2013. URL: `https://commons.wikimedia.org/wiki/File:Colored_neural_network.svg` (visited on 06/02/2015)

So within our network the neurons are arranged into layers like what is shown in Figure 2. You will have data available to the so called input layer, each neuron takes the value in from every object that is in front of it, if we take the front to be left. The values they take in could be either raw data or from other neurons of course depending on where they're situated. Now there are networks that will let neurons reach out other places than just from the next layer, but the software I'm presenting as is stands now doesn't make that easy to do so I will only be referring to the simple layer after layer variety. Now we're setup to talk about the cascade of events that is going to happen as our neurons do their action, which we now know to be finding Z_i then finding $\sigma(Z_i)$.

So we place some data in front of the the input layer and tell the neurons to go ahead and look at it and start calculate all the outputs or activations a, for the layer. The input layer reads the data and activated, the next layer reads the input layer, and so on... until reaching the output layer which we can then look at for the result. The first time through, as could be expected the data is more or less random, so now is when we've got our opportunity to give the neurons some feed back that will determine how they should change their internal values. Now comes the second cascade of events that our neurons will perform, a concept known as back propagation. Here the difference of the expected outcome and the actual outcome if calculated, and using that to calculate the error δ, the previous layer will take δ and also each of the weights that were used against it and calculate it's error, and the process repeats all the way back to the input layer.

To be more rigorous about what's happening, we shall be estimating the gradient of the **cost** function with respect to w and b in terms of the expected outcome y, with the cost function being the aforementioned mean squared error. If we think of each layer as a vector, which has a matrix of weight values, and another vector of bias values we can express the error of any given layer with:

$$\delta^l = ((w^{l+1})^T \delta^{l+1}) \odot \sigma'(z^l) \tag{3}$$

Where l is index of the current layer.

We now have all the partial derivatives, here are the gradients for w and b

$$\nabla_w = \delta^{l+1}(a^l)^T \tag{4}$$

$$\nabla_b = \delta^{l+1} \tag{5}$$

And then update your weights and biases negative of this multiplied by some constant.

$$w^l \rightarrow w^l - \frac{\eta}{m}\sum_x \delta^{x,l}(a^{x,l-1})^T \tag{6}$$

$$b^l \rightarrow b^l - \frac{\eta}{m}\sum_x \delta^{x,l} \tag{7}$$

Notice the x in these equations. This representation assums that a mini-batch of items are selected at random from the list of training data. These equations show that we are taking the average of a mini-batch to update the **bias** and **error** values of the system. This process is the origin of the name stochastic backpropagation. Ok, lets move on to some actuall code now.

Within the Neural Net project directory there are a number of c++ header and source files containing a number of classes, structures, and functions many of which are c++ templates. These items will be briefly discussed here.

There are also some sub-directories, the one named 'res' contains resources meant for use with the Neural Net architecture to train on but this remains unimplemented at this time. The directory named 'DEMO' contains the files needed to compile the working demo.

The reader may build and run the demo program by simply entering the 'DEMO' directory from a Linux terminal and entering the command `make` followed by executing the now compiled file with the command `./DEMO`. The reader may note that most of the header files do not have accompanying source files with the `.cpp` extent ion; this is because most of the classes and functions of this architecture are templates for performance reasons. The template class member functions and normal functions that are also templates are both declared and defined entirely in the header file, this is because templates are evaluated at compile time based on their template parameters, the compiler reserves memory for each unique set of template parameters. The only reason this is worth mentioning is because it is a strategy for increasing performance in a sort of trade off between longer compile time and more memory usage is exchange for faster execution. This also has the unfortunate effect that template parameters for things like an array size cannot be declared at run time unless a compiler activity can be executed somewhere outside of the architecture's files before any features of those classes are called upon (this is actually a planned feature for the future of this project, I plan to be tinkering with and improving thiIs code for some time).

I will now enumerate the contents of each file, but I will only be listing some of the more important functions within due to the large number of functions that are defined.

2 Header File — `Neuron.h`

Within the header file of **`Neuron.h`** is a single class representing the most basic component of the Neural Net, the **`Neuron< >()`** .

2.1 Instantiable — `class Neuron< >()`

This class contains all the field members that will be used extensively during operation of the Neural Net. A programmer will never need to interact directly with Neuron objects since any member access's or calls made to are handled by a friend class, the **`Layer< >()`** and other objects therefrom as we will see later. Another point of interest is that no actual **`Neuron< >()`** objects will ever be instantiated. This is because **`Neuron< >()`** implements an ***abstract class*** design patter, meaning that other classes will extend this class as their parent class. The derived, or child, classes will have all the member functions and fields as **`Neuron< >()`** while still being able to have their own unique members, however what makes this class ***abstract*** is the interesting ability to call upon certain member functions of it's children classes, regardless of what that child class is, so long as it has the functions with the correct name. This is done using a concept know as the ***Curiously Recurring Template Pattern*** or just CRTP [3] . Use of the CRTP opens up all sorts of interesting possibilities, but our main use is for making calls to derived, children classes.

2.1.1 Member Function — `FuncSig GetActivationFct()`

These functions are the special abstract functions that make calls to the objects of the derived classes. The CRTP is what makes this possible, specifically it is the template class **`DownCast< >()`** ; which is found in the acting as the **`Neuron< >()`** 's base class aka the parent class. The **`DownCast< >()`** class provides the member function **`Self()`** which follows a convoluted path of casting pointers around and passing them to other functions to be cast again, but at the end of it all you have a pointer returned (that isn't assigned to a variable, which making it temporary) that cast is such a way that you can call children object's functions. Yikes.

In plain terms, the whole purpose is to resolve the following situation: The **`Layer< >()`** does not have a set type of neuron, it just has a place for neurons of any variety we want. The type of neuron in a layer is defined when an instance of **`Layer< >()`** is made into an object. But the layer still needs to be able to be able to call on the specialized function that each different type of neuron has, well, this allows that to happen safely and reliably.

This was easily on of the most difficult problems to solve for the way I wanted the program to flow, luckily I was so stubborn in wanting it to perform just so that I received a painfully frustrating, but ultimately rewarding, lesson in the use of an advanced template technique. I arrived here after what was probably about a week spent pouring over somewhat similar issues found on the StackExchange website that a variety of users had posted questions and answers about. I certainly wouldn't pretend to be an expert on the subject, but just learned enough to get the result needed for this particular issue.

`GetActivationFct()` returns a **`FuncSig`** which is a typedef that simply holds a function, in particular activation function for the **`Neuron`** child object.

2.1.2 Member Function — `GetActivationFctPrime()`

This returns the derivative function of **`FuncSig GetActivationFct()`** . Further information is provided in 2.1.1

3 Header File — `Utility.h`

This header contains declarations for a variety functions that have utility or ***could*** have utility, as many are yet unused. They go in the utility files generally because they are used in many places, and that just makes for better organization if you're looking for something.

3.1 Instantiable — `typedef double (*FuncSig)(double z)`

This type Type is a wrapper class used to pass around any function pointer that has a double as a parameter and returns a double. Its the return type used for the functions of section 2.1.1.

3.2 Instantiable — `class DownCast< >()`

This class is designed to interact with another class that extends it as it's base. It's main purpose is to provide a member function which descendants may call upon to be returned an instance from which to make call from. The implementation of the utility of this class is well described in section 2.1.1.

3.3 Instantiable — `struct CommAry< >()`

This class simply holds a single array whose size is determined by the template parameters of the struct. It is used extensively in the Neural Net Architecture for passing data with out having to specify an array size.

3.4 Instantiable — `CommAry's` Printing Tools and Math Tools

These are whole sections that contain tools that offer a wide variety of options for printing values in a `CommAry< >()` array and for modifying values mathematically. For example, Figure 3 is referred to as the "unrolling" of a `for()` loop. This term was prevalent on forums as I researched ways to maximize computational performance. It sends 8 instructions at once for each iteration of the upper loop, which has the effect of removing calculations that can clog up the processor by causing L2 cache misses.

```

for (int i = 0; i < rolled_total; i += 8) {

    a.ary_[i + 0] += b.ary_[i + 0];
    a.ary_[i + 1] += b.ary_[i + 1];
    a.ary_[i + 2] += b.ary_[i + 2];
    a.ary_[i + 3] += b.ary_[i + 3];
    a.ary_[i + 4] += b.ary_[i + 4];
    a.ary_[i + 5] += b.ary_[i + 5];
    a.ary_[i + 6] += b.ary_[i + 6];
    a.ary_[i + 7] += b.ary_[i + 7];
}

for (int i = rolled_total; i < Ary_Size; i++) {
    a.ary_[i] += b.ary_[i];
}
```

Figure 3: This is referred to as the "unrolling" of a `for()` loop. In this example we see how two objects

3.5 Instantiable — `class RandTools()`

This class has tools for generating both flat and Gaussian distributed random numbers. There are member functions for returning standard distributions that take no parameters, and there are member functions that allow control of the center location and spread. It has an interesting feature where it creates a static instance of itself the first time it is called such that it is only seeding the underlying `rand()` function once. It has a call back **`RandExitHandler()`** that is meant to be passed to `std::atexit()` in the main function so that the program has a way to delete the memory allocated to the static instance of the generator when the program is terminating.

4 Source File — `Utility.cpp`

This is the only source files that has methods separated from the header file. This is because every almost all of our important classes are templated. RandTools, however, has no dependence on the size of an index like the arrays in the other classes, which is also the same reason why the class uses a Singleton Pattern.

The Singleton Pattern will create a single static instance the first time the class member functions are called. Any subsequent calls refer to this same instance. The only extra consideration is that it has to register a call-back so that the static instance is destroyed when the program exits.

4.1 Member Functions of — `class RandTools()`

The member functions of **`RandTools()`** are separated from the header file into a source file as is usual. The reason more functions are not separated out this way for the Neural Net is because it relies heavily on templated functions and classes.

`RandTools()` itself, however, has no dependence on the size of an index like the arrays in the other classes, which is also the same reason why the class uses a Singleton Pattern.

The Singleton Pattern will create a single static instance the first time the class member functions are called. Any subsequent calls refer to this same instance. The only extra consideration is that it has to register a call-back so that the static instance is destroyed when the program exits.

5 Header File — `DLog.h`

This header contains debugging message tools. It contains a macro for enabling and disabling error messages that can be called with DEBUG().

5.1 Instantiable `struct dbglog()`

This struct does not get instantiated, rather it contains many operator overrides which makes it a functor. Usage is simple, you just place the message you want to say in the argument. You can also use `<<` in the same way as `std::cout` to send in strings to the stream.

6 Header File — `Color.h`

Color simply holds pre-complier definitions to the ANSI escape codes for colored terminal output. There are no classes or members for this file.

7 Header File — `Layer.h`

This is the largest file so far. It contains the tools for creating a layer of neuron objects, and is also responsible for communication between layers. As such, it houses some of the heaviest functions in terms of processing. The instantiable items consist of a structure, `Synapse< >` , and a class `Layer< >` .

7.1 Instantiable `class Layer< >()`

Instances of this class hold a group of instances of neuron objects. It holds a `ComArray()` object for the outputs of each neuron in its layer. It also has a `ComArray()` for Z values calculated from the inputs of Z. There is another `ComArray()` that hold the values of error. It contains one last `ComArray()` that holds the sum of error.

It also has references to a previous `class Layer< >()` 's input, Z, error, and sum of error values. In such a way all the layers are connected to each other. They connect to each other using `Synapse()` .

7.1.1 Member Function `void Z_Cascade()`

This calculates for each neuron, each previous layers outputs scaled by a corresponding weight value, added to their bias value. This is called the Z value and it is held onto because it's used in other places.

7.1.2 Member Function `void Activation()`

This applies the activation function of the type of neuron this layer is for, to each of its neurons.

7.1.3 Member Function `void ErrorCascade()`

This calculates the error for each neuron from the error in the previous layer, and the derivative of the activation function.

7.1.4 Member Function `void DescendGradiant()`

This uses the sum of errors of each neuron to calculate that neurons weight values and bias.

8 Header File — `Perceptron.h` and `Sigmoid.h`

These are the unique polymorphisms of the neuron class, and they hold special activation function. There are many differnt kinds of neurons, and it is intended that the user is able to create their own unique neuron that has the activation function the user chooses.

9 Header File — `POSI.h`

This has nothing more than a single enum for identifying whether a layer is at the head, body, or tail of the neural network. No additional adornments are necessary.

10 Source File — `Alan.cpp`

This contains the `main()` function, and therefore it is the starting place of our demo. It first handles preliminary tasks, before calling `BackPropTest` . Within `BackPropTest` , the first thing you'll notice are constant integers that will describe the size of each layer. Next, you will see some very large arrays of test data which will be used for training our network. We instantiate 2 `CommArray()` , one for the input data to the network, and one is the expectation which ultimately is used to measure against the training data.

Four layers are created, and linked to each other with the `Synaptogenisis()` function. Then, a `CommAry()` is created for each layer's output and again for each layer's error. Each layer has every bias and weight value initiated to a random value.

At this point, a loop is started that waits for valid user input. When it receives a valid number for the number of mini batches that it is set to run, it picks a random set of values from the array as a mini batch and calculates the Z and activation value in order from the first layer to the last. The error is calculated in reverse order, which is also known as back propagation.

It uses the error values to change the weights and biases for every neuron using `DescendGradient()` . Lastly, it prints the error values and output values of each layer to a terminal window.

References

[1] Glosser.ca. ***Artificial neural network with layer coloring***. Feb. 28, 2013. URL: `https://commons.wikimedia.org/wiki/File:Colored_neural_network.svg` (visited on 06/02/2015).

[2] Michael. Nielsen. ***Neural Networks and Deep Learning.*** URL: `http://neuralnetworksanddeeplearning.com/` (visited on 06/02/2015).

[3] In Wikipedia. ***Curiously Recurring Template Pattern.*** URL: `http://en.wikipedia.org/wiki/Curiously_recurring_template_pattern` (visited on 06/02/2015).

[4] In Wikipedia. ***Frank Rosenblatt***. URL: `http://en.wikipedia.org/wiki/Frank_Rosenblatt` (visited on 06/02/2015).