Hebbian neural networks and the emergence of minds

Cory Stephenson

December 13, 2010

Abstract

The goal of this work is to provide an overview of artificial neural networks and some of their emergent properties. The neural network model is briefly motivated from a biological point of view, and then the typical network architecture is introduced. A Back-Propagation learning rule is briefly explored using a simple code as an example of supervised learning, and Hebbian learning is introduced as a simple example of unsupervised learning. The emergent pattern recognition and novelty filtering aspects of Hebbian networks is discussed in this context, and a rudimentary novelty filter is run over historical stock market data to show the ability of a Hebbian network to recognize anomalous patterns in data. The manner in which these networks are able to store information is explored, leading to the conclusion that spontaneous structure and memory formation in neural networks fed random data is related to broken time reversal symmetry. This overview ends by drawing parallels between the emergent phenomena found in Hebbian neural networks and human intelligence.

Introduction to neural networks

The nervous systems of the various creatures that have evolved on Earth are some of the most marvelously complex systems known to man. We find that people and animals can make use of their brains to learn new behaviors, remember past events, recognize patterns and even create. In spite of the ubiquitousness and importance, little is known about how a brain functions. We know that it has something to do with interactions between its components, but the sheer number of components and the intricacy of the interactions make it very difficult to see how it accomplishes these amazing feats. One commonly used approach is to examine the structure of the neurons in the brain and to create a simplified model based on this. It turns out that a typical neuron is connected to a large number of other neurons via branching fibers These connections are very complicated (see Figure 1) and they can even change

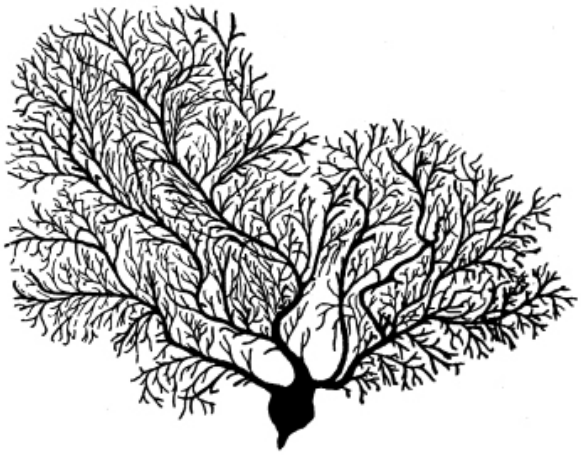

Figure 1: A Purkinje cell from the cerebellum of a 63 year old man [1]

during ordinary functioning of the brain. To study the behavior of connected groups of neurons, the concept of an artificial neural network was introduced. A large number of phenomena were found to emerge from these networks, such as learning, memory, pattern recognition, and even the ability to repair themselves. They have even found applications in the fields of computer science, financial analysis, and a host of others. To see how these simple models allow for such diverse and complex behaviors, it is necessary to understand how an artificial neural net is constructed.

How an artificial neural network functions

Artificial neural networks were inspired by the nervous systems of biological organisms in order to better study the function of naturally occurring brains and to computationally emulate their aptitude for solving certain problems that are ill suited to con-

ventional techniques. Mathematically, an artificial neural network (ANN) is viewed as a collection of nodes (termed 'neurons') which are connected to each other via unique interactions between each neuron. The strength of the interaction is referred to as the synaptic weight w_{ij} of the connection between neurons i and j. A typical (feed-forward) neural network consists of three layers of neurons: the input layer, the hidden' layer, and the output layer. It can be shown that this network architecture is sufficient to complete any task that can be done with a more complicated network with arbitrary precision[2].

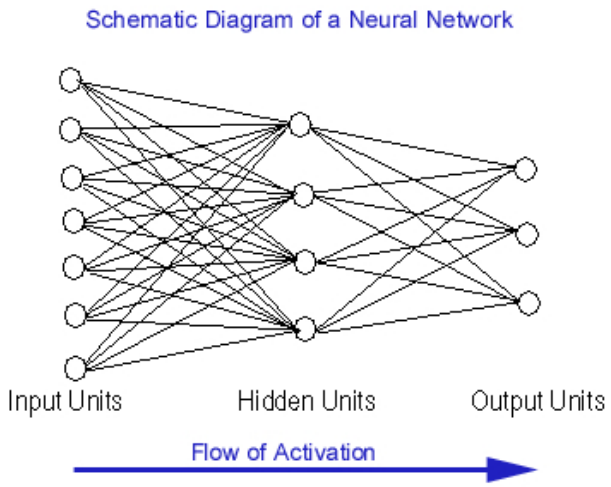

Figure 2: Conventional architecture of an artificial neural network[3]

The output of the neural net is calculated by using the input and the weight values between input layer and hidden layer to compute the 'activation' of the neurons in the hidden layer. The weights of the connections between the hidden layer and the output layer are then used with the activation of the hidden layer to produce the output. In order to compute the activation of a neuron, it is necessary to use a function called a Sigmoid function. A common choice of Sigmoid function inspired by the function of real neurons is

$$S[x] = \frac{1}{1 + e^{-x}}$$

Though other common choices include simple step functions, and piecewise linear functions. The Sigmoid function is usually restricted to take values in the range of $(0, 1)$ but this is not mandatory. The activation of the j^{th} neuron (N_j) in the hidden layer can then be computed as

$$N_j = S[w_{ij} I_i]$$

where I_i is the activation of the input neuron. Similarly, the activation of the k^{th} output neuron O_k can then be computed by

$$O_k = S[m_{jk}N_j]$$

where m_{jk} is the weight of the connection between the j^{th} hidden neuron and the k^{th} output neuron.

While this is all that is necessary to compute the output of a neural network given the input, it is not very interesting or particularly useful unless the weights are tuned to produce desired outputs. However, the complexity of a neural net rises dramatically as more neurons are added, meaning it is very difficult to determine what set of weights will produce a given outcome. The power of artificial neural nets only becomes apparent when they are made to tune themselves by learning and adapting to different inputs. A neural net can be trained to recognize different inputs and produce desired outputs if it is repeatedly presented with these inputs and the desired outcomes, and modifies its own weights to reduce the difference between the actual output and the desired output. This is called 'Supervised Learning;' and can be simply implemented using a variety of techniques. One common method of teaching a neural net (called back-propagation) presents the network with a set of inputs and a set of desired outputs D_k. The neural net is run over an input, and then computes the error between the actual output and the desired output via [4]

$$E_k^o = (D_k - O_k)O_k(1 - O_k)$$

This error is then propagated backwards through the net to compute the errors in the hidden layer by

$$E_j^n = N_j(1 - N_j)m_{jk}E_k^o$$

The weights are then updated according to the rule [4]

$$\begin{aligned} w_{ij}^{new} &= w_{ij}^{old} + \alpha I_i E_j^n \\ m_{jk}^{new} &= m_{jk}^{old} + \alpha N_j E_k^o \end{aligned}$$

where α is the "learning constant." It is typically chosen to be a small value so that the learning process does not immediately overwrite any previously learned information. The network is repeatedly exposed to the inputs and outputs to be learned, and over time adjusts itself to produce the desired result. It is important to note that there are many different weight configurations which can complete a given task. As an example of this feature, and this type of neural network in general, I have programmed a simple network that has three inputs, three hidden neurons and one output (source code can be found in Appendix A). The network was trained to produce a response of one for the inputs $(1, 0, 0)$ $(0, 1, 0)$ and a response of zero for $(1, 0, 1)$. The network was shown the correct input and output combinations 30,000 times, which is far

longer than necessary, to ensure that differences in the weight matrices would not be transient. This was done twice, producing weight matrices

$$w_{ij}^1 = \begin{pmatrix} 0.004 & 0.864 & -2.171 \\ -0.128 & -1.045 & 2.383 \\ 0.921 & 1.205 & -3.410 \end{pmatrix} \quad \text{and} \quad w_{ij}^2 = \begin{pmatrix} 0.005 & 0.401 & 1.974 \\ -0.443 & 1.104 & -2.099 \\ 1.064 & 0.630 & -3.850 \end{pmatrix}$$

And so we can see explicitly that while these two networks were trained to perform the same task (which they both do quite well), they have arrived at very different solutions. Although it is not required by the architecture, the weight values are often restricted to lie between 0 and 1, or between -1 and 1.

The benefit of employing a neural network with this sort of training algorithm is that the network can be presented with an input that is similar but different to one of the learned inputs, and it will be able to recognize what learned input it is similar to and return the desired output. For example, the neural net described above, an input of $(0.75, 0.2, 0.67)$ generates an output of 0.359, a relatively low response which indicates that the network has determined that this input is similar to $(1, 0, 1)$, a pattern which it has learned to provide a low response for. An input of $(0.75, 0.2, 0.25)$, however, generates an outcome of 0.742, a relatively high response indicating similarity to the learned pattern $(1, 0, 0)$. Even though the network was not taught these inputs, it has little trouble determining how close it is to the inputs it has learned. This sort of flexibility is very difficult to achieve with traditional algorithms.

Hebbian Learning

However, the back-propagation method and other supervised learning methods require the network to be spoonfed input and output combinations; it is incapable of learning on its own. Unsupervised learning algorithms do exist, and are capable of organizing themselves to categorize inputs in ways it has not been taught. A particularly simple and important manner in which a neural net can learn is the biologically inspired Hebb's learning rule. In Hebb's own words, the rule is

"When an axon of cell A is near enough to excite cell B and repeatedly or persistently takes part in firing it, some growth process or metabolic change takes place in one or both cells such that A's efficiency, as one of the cells firing B, is increased" [5]

In simpler language, if two connected neurons are activated simultaneously, the connection strengthens which makes them more likely to be activated together in the future. There is a growing body of evidence that suggests that this process plays a role in the learning mechanisms of living organisms, including humans [6].

This type of learning is easy to incorporate into the artificial neural network architecture described above. Instead of modifying the weights in accordance with the back-propagation method, the weights are simply updated as [7]

$$\begin{aligned} w_{ij}^{new} &= w_{ij}^{old} + \alpha I_i N_j \\ m_{jk}^{new} &= m_{jk}^{old} + \alpha N_j O_k \end{aligned}$$

And similarly for additional layers, if present. The new weights only depend on the input and the properties of the network, no other information is needed. It is noteworthy that this rule does not allow the synaptic weights to decrease. For this reason, the Hebbian learning rule is often implemented using the slightly modified form [7]

$$\begin{aligned} w_{ij}^{new} &= w_{ij}^{old} + \alpha(2I_i - 1)N_j \\ m_{jk}^{new} &= m_{jk}^{old} + \alpha(2N_j - 1)O_k \end{aligned}$$

Which causes the synaptic weights to decrease if the activation of the first neuron is less than one half.

In contrast to supervised learning methods, networks that employ Hebbian learning have the advantage of learning as they operate; they don't need separate training and operating cycles.

Emergence of pattern recognition in Hebbian networks

Now that the behavior of an artificial neural network has been quantified, we are ready to see how a network using the Hebbian learning rule can learn how to recognize a pattern, or equivalently, how to determine if a pattern has been violated.

If a neural network with arbitrary but nonzero weights (in computer simulations the weights are typically initialized to random values) is presented with input data, some neurons in the hidden layer will be activated, and some will not. In accordance with the Hebbian learning rule, this increases the synaptic weights linking activated input neurons to activated hidden neurons. If the network is then presented with a new input that is in some way similar to the previous input, some of the previously activated neurons will preferentially become activated again, due to the increase in weights from the previous cycle. This further strengthens the connection between the input neurons and the the hidden neurons. In this way, the connections that are activated the most frequently become by far the most likely to be activated again. The more similar the input data is to data that the network has seen before, the larger

the output becomes. Thus, if the network is repeatedly exposed to some pattern of input data, it will produce a large output when presented with additional data that fits the pattern it has learned.

This ability can be inverted by changing the learning rule to an anti-Hebbian rule. In practice, this is done by setting the learning constant to be negative, meaning neurons that are simultaneously activated become less likely to be simultaneously activated in the future. Instead of producing a large output if the input data corresponds to the pattern the network has learned, a large output is produced if the input does not fit the pattern, as activation configurations corresponding to the learned pattern have been suppressed, and other configurations amplified. These types of networks are often called "Novelty Filters," and have many practical uses [8]. A well known application of these occurs in the financial sector, where they are employed to detect anomalous trading patterns. Figure 3 shows an extremely simple anti-Hebbian novelty filter that I wrote and applied to the Dow Jones Industrial Average between 1985 and 1993. This network was trained using the DJIA data from 1928 onwards. Source code can be found in Appendix B.

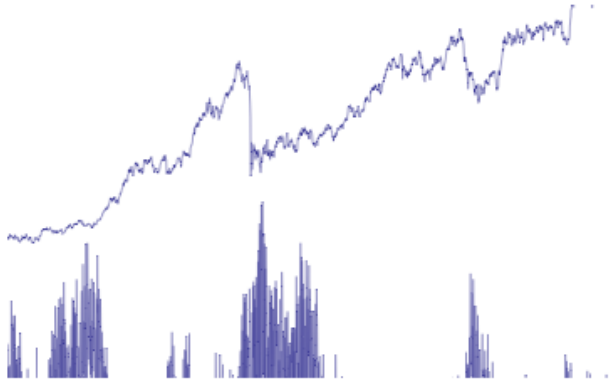

Figure 3: Novelty filter applied to DJIA between 1985 and 1992. Prominent spikes in output are shown corresponding to an unusual bull market in 1985, the Black Monday crash (October 19^{th} 1987) and the early 1990s recession

Such novelty filters are quite useful for detecting anomalous patterns, but as Figure 3 illustrates, they do not offer any information as to what the anomalous pattern is. We see that the novelty filter detects both abnormal rises and falls in the DJIA.

Emergence of memory

We have seen how a neural network can learn to recognize patterns. To do this, the network must have some way of storing past data for future use; it must somehow remember something of what it has seen before. Though it may seem obvious that

the information is stored in the various synaptic weights, it is far from obvious how this is actually done. If one were to look at the actual value of the synaptic weights, it would simply be an incomprehensible collection of numbers (for example, the weight matrices for the simple neural net shown above). Fortunately, several methods of retrieving stored memories from neural networks have been found. A particularly useful method found by Kek called the B-matrix approach allows a stored memory to be retrieved from the network using only a small fraction of the memory as a key to recall a stored memory[9]. This clearly demonstrates what was only implied earlier: Hebbian neural networks posses associative memories.

This associative memory immediately suggests an analogy between the memory of a neural network and the memory of the human brain, which is thought to be associative as well[10]. Another important similarity between biological memory and artificial neural network memory can be seen in the manner in which memories are stored. In artificial neural networks as well as in biological brains, memory is not stored locally, it must be reconstructed from data distributed throughout the system. This is advantageous; if the synaptic weights of a few connections are corrupted, entire memories are not lost but merely degraded somewhat. The same is true for a human brain. We are constantly losing and replacing neurological connections, but we are not constantly forgetting entire memories.

There is also a deep mathematical implication here. If a Hebbian neural network with randomized synaptic weights is exposed to random data, certain neurons will preferentially activate, and become more likely to activate in the future. Even if the input data is totally random, the network will begin to form a nonrandom structure in the same manner as if the data contained a pattern. In fact, any string of random data will by pure chance contain small scale patterns, and the pattern recognition capabilities of the network as described above will adapt to these illusory patterns as if they were meaningful. Thus, by gaining nonrandom structure due to random data, memory of previously seen data is encoded. It is in this way that the introduction of any data to a Hebbian neural network breaks time-reversal symmetry, the result of which is structure and memory formation[11].

The relation between memory storage and time-reversal symmetry breaking is fundamental. The formation of a memory distinguishes past from future in that we have knowledge of one but not the other. As humans, we remember the past but not the future. The same holds for an artificial neural network.

Conclusions

In our brief overview of neural networks, we have seen something quite remarkable. An extremely simple model of cognition was found to perform complicated and varied

tasks. It was shown that collections of simple neurons connected with one another are able to learn from outside data, to recognize patterns, and to spontaneously organize themselves and form memories even in the absence of meaningful input. We even see some semblance of creativity in the way a neural network fed random inputs tries to find patterns and make sense of its disorganized world. In keeping with the origins of this simple model, one cannot help but see parallels between these simple networks and the human mind. From this model of a highly connected network and a learning rule, many of the phenomena we associate with human intelligence emerge. It is therefore not too far fetched to wonder if the human mind itself is an emergent property of highly connected brain cells.

References

[1] L. Burns, http://www.meridianinstitute.com/eamt/files/burns2/bur2ch14.html.

[2] K. Hornik, *Neural Networks* **4** (2) 251 (1991).

[3] Artificial Intelligence — Artificial Neural Networks, http://www.psych.utoronto.ca/users/reingold/courses/ai/nn.html.

[4] Y. Chauvin and D. E. Rumelhart, *Backpropagation: Theory, Architectures, And Applications* (Lawrence Erlbaum Associates, Inc., Hillsdale New Jersey, 1995).

[5] D.O. Hebb. *The Organization Of Behavior.* (Wiley & Sons, New York, 1949).

[6] O. Paulsen and T.J. Sejnowski, *Current opinion in neurobiology* **10** (2) 172 (2000).

[7] C. Fyfe. *Hebbian Learning And Negative Feedback Networks* (Springer-Verlag London Limited, United States, 2005).

[8] The Talented Dr. Hebb, Part 1, Novelty Filtering, http://blog.peltarion.com/2006/05/11/the-talented-dr-hebb-part-1-novelty-filtering/.

[9] S.C. Kak, *Phys. Lett. A* **143** (6,7) 293 (1990).

[10] J. R. Anderson and G. H. Bower. *Human Associative Memory: Brief Edition* (Lawrence Erlbaum Associates, Inc., Hillsdale New Jersey, 1980).

[11] C. Gros, *New Journal of Physics* **9** 109 (2007).

Appendix A

Mathematica code

```
(*initialization of variables*)
dimi = 3; (*number of inputs*)
dimn = 3; (*number of hidden neurons*)
dimo = 1; (*number of outputs*)
tcy = 30000; (*numberoftimestoteacheachpattern,
doesnotneedtobethislarge*)

(*initializing neuron activations*)
i = Table[RandomInteger[{0, 1}], {j, 1, dimi}];
n = Table[0, {j, 1, dimn}];
o = Table[0, {j, 1, dimo}];

(*defining weight matrices*)
w = Table[Table[RandomReal[{-1, 1}], {nu, 1, dimi}], {mu, 1, dimn}];
m = Table[Table[RandomReal[{-1, 1}], {sig, 1, dimn}], {nu, 1, dimo}];

(*defining sigmoid function and thresholds*)
S[s_] = 1/(1 + Exp[-s]);
T = {Table[0, {j, 1, dimn}], Table[0, {j, 1, dimo}]};

(*learning rates*)
a = .01;
h = .001;
(*Patterns to learn*)
np = 3;
p1 = {0, 1, 0};
p2 = {1, 0, 1};
p3 = {1, 0, 0};
p = {p1, p2, p3};

(*desired outputs*)
od = {{1}, {0}, {1}};
(*training*)
Do[
Do[
i = Part[p, l];
n = S[w.i - Part[T, 1]];
o = S[m.n - Part[T, 2]];
```

```
Eo = (Part[od, l] - o) * o * (1 - o);
En = n * (1 - n) * (Eo.m);
dt = h * {En, Eo};
dm = a * KroneckerProduct[Eo, n];
dw = a * KroneckerProduct[En, i];

(*HebbianLearning;
dw = a * KroneckerProduct[n, (2 * i - 1)];
dm = a * KroneckerProduct[o, (2 * n - 1)];
*)

tt = T;
T = tt - dt;
wt = w;
mt = m;
w = wt + dw;
m = mt + dm;
, {l, 1, np}];
, {k, 1, tcy}]
(*running*)
i = {0, 1, 0};
n = S[w.i - Part[T, 1]];
o = S[m.n - Part[T, 2]]
```

Appendix B

Mathematica code

```
(*importing data*)
djia = Import["djiatable.csv"];
open = Table[Part[djia, 20643 - j, 2], {j, 1, 20641}];
close = Table[Part[djia, 20643 - j, 5], {j, 1, 20641}];
volume = Table[Part[djia, 20643 - j, 6], {j, 1, 20641}];
dopen = Table[(Part[open, j] - Part[open, j - 1])/Part[open, j], {j, 2, 20641}];
dclose = Table[(Part[close, j] - Part[close, j - 1])/Part[close, j], {j, 2, 20641}];
dvolume = Table[(Part[volume, j] - Part[volume, j - 1])/Part[volume, j], {j, 2, 20641}];
(*initialization*)
dimi = 3; (*number of input neurons*)
dimn = 20; (*number of hidden neurons*)
dimo = 1; (*number of output neurons*)
tcy = 20640; (*number of data points*)

(*initializing neuron activations*)
i = Table[RandomInteger[{0, 1}], {j, 1, dimi}];
n = Table[0, {j, 1, dimn}];
o = Table[0, {j, 1, dimo}];

(*initializing weight matrices*)
w = Table[Table[RandomReal[{-1, 1}], {nu, 1, dimi}], {mu, 1, dimn}];
m = Table[Table[RandomReal[{-1, 1}], {sig, 1, dimn}], {nu, 1, dimo}];

(*Definingsigmoidfunctionandthreshold(notused)*)
S[s_] = 1/(1 + Exp[s]);
T = {Table[0, {j, 1, dimn}], Table[0, {j, 1, dimo}]};

(*Learning Rates*)
a = -.003;
h = .001;
nov = {0};
ens = 10; (*numberofdifferentnetworkstoaverageover*}
(*running over data*)
novcum = Table[0, {j, 1, tcy + 1}];
Do[
novct = novcum;
nov = {0};
(*resetting network info*)
```

```
i = Table[RandomInteger[{0, 1}], {j, 1, dimi}];
n = Table[0, {j, 1, dimn}];
o = Table[0, {j, 1, dimo}];
w = Table[Table[RandomReal[{-1, 1}], {nu, 1, dimi}], {mu, 1, dimn}];
m = Table[Table[RandomReal[{-1, 1}], {sig, 1, dimn}], {nu, 1, dimo}];
Do[
i = {Part[dopen, k], Part[dclose, k], Part[dvolume, k]};
n = S[w.i - Part[T, 1]];
o = S[m.n - Part[T, 2]];
novt = nov;
nov = Join[novt, o];
(*Hebbian Learning Rule*)
dw = a * KroneckerProduct[n, (2 * i - 1)];
dm = a * KroneckerProduct[o, (2 * n - 1)];
wt = w;
mt = m;
w = wt + dw;
m = mt + dm;
, {k, 1, tcy}];
(*averaging over multiple network runs*)
avg = Sum[Part[nov, j], {j, 1, Part[Dimensions[nov], 1]}]/Part[Dimensions[nov], 1];
novadj = nov - avg;
novcum = novct + novadj;
, {j, 1, ens}]
novtot = novcum/ens;

(*plotting data*)
Show[ListLinePlot[novtot, PlotRange → {{14000, 16000}, {0, .2}}, Axes → False],
ListLinePlot[close/96000, PlotRange → {{0, 20640}, {0, 1}}]]
```