# Artificial Neural Networks in Akka

Maciej Gorywoda

gorywodamaciej@gmail.com

November 13, 2016

# Table of Contents

# 1. Abstract

As CPUs reach their upper bounds in the terms of clock rate, parallel computing becomes the necessity in processing large data sets. Artificial neural networks (hereinafter called “networks”) can be seen as a means to parallelize computations where other methods fail. This article discusses one of possible approaches: a network designed to perform a specific task, where each artificial neuron is implemented as an independent entity, working in parallel with other neurons. The computer program which is a part of this project builds an asynchronous network and uses it as a data stream transformer. The input stream of symbols can be decoded into an input vector pushed into the network. The network is able to generate a stream of more abstract symbols, using as additional information both its internal state (i.e. data which it received before) and time gaps between consecutive chunks of data. In the following part the article discusses features exhibited by this type of a network and explores how to design it from smaller sets of neurons.

# 2. Introduction

In research on Artificial Neural Networks (ANN) we can roughly distinguish two popular approaches:

1. Mathematical modelling. This is the approach that gave us the Multilayer perceptron, the Hopfield network, and many others. It describes ANNs as mathematical models, where connections between neurons can be simplified to numbers in a matrix, and the input signals to numbers in a vector. In order to simulate the network's activity, instead of actually building the network, it is enough to perform mathematical operations. [1][2]
2. Spiking Neural Networks (SNN), which are much closer to actually simulate a working brain. Tightly connected to neurobiology, this approach takes what we know about how neurons are built and how they react to neurotransmitters and, followingly, model the activity of a nervous system consisting of thousands, or even millions of such simulated neurons, each described by many variables. This way we can get a valuable insight into how we think and – in consequence – who we are. [3]

The approach I use in this work lies in the middle. I draw inspiration from how organic neurons work, but I focus on a few of their most important traits and simplify the ways they cooperate – not so much that it would be possible to describe it with a single mathematical equation, but enough to track and anticipate their activity. Each artificial neuron receives signals from other neurons, process them taking into consideration only a few variables aggregated in its internal state and send the result to yet other neurons. This is similar to the mathematical modeling but, in resemblance to an organic brain, each of the neurons is a separate entity, “an actor”, and acts concurrently to others. As a matter of fact, such a neuron does not even has to know that it is a part of a network, just as our neurons do not know they are parts of our brains – they just react to the stimuli wherever they come from. Asynchronicity of artificial neural networks was discussed in theory in scientific works [4] [5], but only recently the development of frameworks simplifying the construction of concurrent and distributed applications, such as Akka, has made it easy to explore their practical applications and side-effects.

The goal of the project discussed in this article is to build a network which works in real time and use it in order to transform data chunks incoming in a form of a stream into a stream of data chunks of another type. Such a network can be relatively small and simple when compared to other types of artificial neural networks able to perform such tasks. It demonstrates some interesting features absent from those networks, such as usage of time gaps between consecutive data chunks as additional information. First, I will introduce and discuss the main concepts behind building such a network, and then I will describe how to build a small network which will recognize an S.O.S. signal in the Morse code from an input stream of 1s and 0s.

# 3. Concepts

## 3.1 Akka

Akka is a toolkit and runtime simplifying the construction of concurrent and distributed applications on the Java Virtual Machine. Akka supports multiple programming models for concurrency, but it emphasizes actor-based concurrency, which is message-based, asynchronous, and implements the actor model. Typically no mutable data are shared and no synchronization primitives are used. [6]

The most important features of Akka from this article's point of view s are:

1. Actors work concurrently to one another.
2. Actors communicate with one another with messages – small immutable data packets – and not in any other way.
3. Actors are reactive: their computations are triggered by incoming messages; there is no need for a supervisor entity which would iterate over them and send requests to perform computations. [7][8]
4. Sending a message takes time: every time a message is sent there is a considerable time gap between the moment it is sent and the moment it is received. Such time gaps exist even if the actor sends a message to itself. Considerable” means that it has to be taken into account when designing a system of actors (i.e. a network).
5. Communication is unordered: two messages sent at the same moment may reach their targets with a considerable time difference, and two messages sent one after another from one source to two different targets may be received in a different order. However, Akka ensures that two messages going from one source to one target will reach it in the same order they were sent. [9]

In my work, Akka actors are used in order to implement neurons, while messages are used to implement signals sent between neurons.

## 3.2 Network

The network consists of any number of neurons of the following three types:

1. **Input:** “dummy” neurons, working as entry points for signals coming in from the outside of the network; their only purpose is to pass the signals to other neurons.
2. **Standard**: neurons consisting of constants, set at the initialization of the neuron and variables which change during its work.
3. **Silencing:** simplified versions of standard neurons, used to send to other neurons signals of a special type in order to silence them. This feature is discussed below.

The network works as a data stream transformer: it takes chunks of data from a data stream, performs a series of transformations on them, and pushes the results out into another data stream. To the outside world, the network presents a very simple interface: a vector of numbers in the range of 0.0 to 1.0 can be pushed into the the input and at the output we receive signals from neurons which are tagged as output neurons. There is no distinction between middle and output layers of the network. Every standard neuron can be marked as an output one and then, when it fires, a symbol associated to it is pushed into the output stream. In order to work on more abstract data we need to put a wrapper around the network. The wrapper will take a symbol, decode it into a vector of numbers, and push it into the network. The vector is then distributed among input neurons, one neuron for one element of the vector. The input neurons pass them as signals to standard neurons which eventually send their own signals to yet other neurons. After some time these signals reach neurons tagged as output ones and, eventually, they will fire. Then the wrapper will push into the output stream a symbol associated with the given output neuron. This is similar to mathematical modeling, but in the case of an asynchronous network there are three important differences:

1. If the network still computes the output for the first vector when we push another input vector into it, the result of those computations may be affected. Also, the internal state of the network, which is a side effect of computations for the first vector, may affect the result of computations for the second one.
2. The time gap between consecutive input vectors is also affecting the computations.
3. The network may accept new input vectors while still working on previous ones.

In short, the network is not an (*input vector → output vector*) transformer, but (*input vector + internal state → output vector*), where the internal state of the network is the sum of internal states of all the neurons.

How are these time gaps useful? Why not hit the network with all data at once? An asynchronous neural network is naturally prepared to work within the time context and this makes it similar to how our brains work. Imagine starting to listen to a song and being immediately provided with all sounds at once. It would make no sense. What changes a bunch of sounds into a song are their length, their sequence, and the time gaps between them. In a very simplified example, which follows this theoretical part of the article, I show how short bursts of simple signals (basically just one-value vectors, where the value is the number '1') can be transformed into the SOS message. In this case, the lack of signal between three short bursts (“1...1...1...”) means that the input data stream carries the letter 's' in the Morse code- not just one long line which does not translate into anything useful.
But to be able to distinguish that, first we need a way to tell that the time gap between two consecutive signals is big enough to be considered important, instead of being just a random fluctuation in the input stream.

## 3.3 Local synchronization and iterations

Neurons in the network are not aware they are a part of a bigger system. They know their internal state and how to send signals to a set of other neurons. Usually it is a set much smaller than the whole set of neurons in the network. On top of that, their computations are triggered by signals coming to them, so even in this regard we can say they are independent – they do not have a supervisor telling them to perform computations.
On the other hand, in order for the network to use time gaps as information, the neurons must be at least locally synchronized. Local synchronization means that even though there is no one source of synchronization (I.e. the aforementioned supervisor triggering computations in the same moment of time) small number of interacting neurons are working in an established sequence. Consider the following example:

1. We have a network consisting of one input neuron IN and a set of N standard neurons connected to IN in parallel. Each standard neuron has the threshold set to 0, so any signal sent

from IN triggers it to send its own signal.

2. In a time gap between moments t1 and t2, the input neuron IN fires and sends signals to each of N standard neurons. Each of the signals may take a random, considerable amount of time to reach the neuron. We call this event S1.
3. Without waiting for the neurons to finish their computations, IN fires again and a new round of signals is sent to the neurons. We call this event S2. Because the communication is unordered and takes time, it is possible that some of the signals from S2 will reach their targets even before some of the signals from S1 will reach theirs.
4. Still, all neurons should send their output signals triggered by signals from S1, before any of them is triggered by signals from S2.

Without local synchronization it would be possible that in a block of interacting neurons one part of the block worked considerably faster or slower than another part. This could lead to errors in recognizing patterns and would make the network much more difficult to design and to track its work. Also, please note that if the N neurons from the example above are connected in sequence, instead of being parallel to one another, then with N big enough the total sum of delays in delivering messages would still be greater than the time between S1 and S2, no matter how large. This is why I called it local synchronization – neurons removed from each other do not have to be synchronized for the purposes discussed in this article.

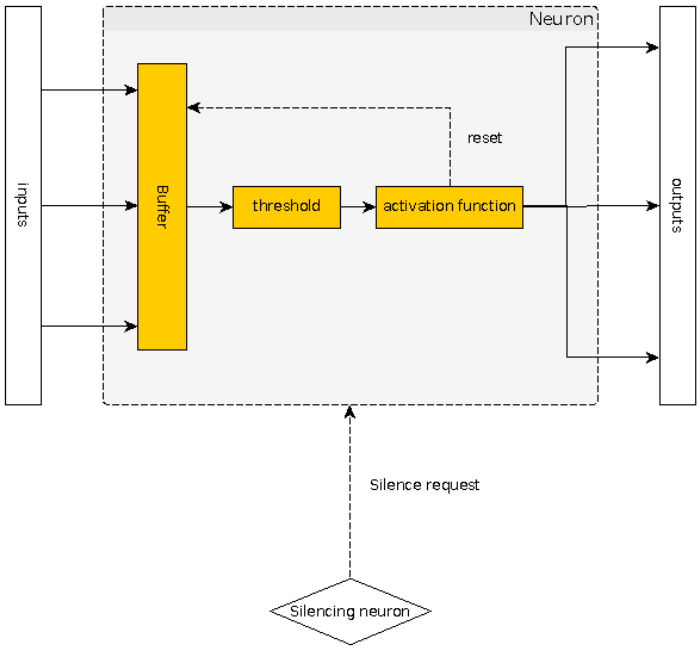
In order to achieve local synchronization, I introduced the concept of sleep time. Each time a neuron sends its output signal it goes to sleep for a given amount of time which is the same for all neurons in the network. During the sleep time the neuron adds signals to its buffer. However, even if the buffer goes over the threshold, it does not trigger the output signal. Only after the neuron wakes up, does it check the buffer against the threshold and either sends the output signal – which consequently makes it go to sleep again – or stays awake and wait for more signals to come.

With a block of locally synchronized neurons we can talk about iterations in their work. One iteration is a time span during which all neurons in a block either fire only once or do nothing. It is not possible for a locally synchronized neuron to fire more than once during one iteration. This concept will be useful in the following sections of the article when discussing how neurons interact with each other in order to perform specific tasks.

## 3.4 Neuron

A neuron is implemented as an Akka actor, meaning it is reactive, it works concurrent to other neurons, and it communicates with the outside world through signals. It can receive and process only one signal at the time.

In the Multilayer perceptron and Hopfield models we can say that a neuron reaches behind itself for input data [1]. The synapses connect the neuron to other neurons which present their outputs. The neuron sums the outputs multiplied by the synapses' weights, checks if the sum is bigger than some threshold and if yes, it applies an activation function to the sum and presents the result as its output to yet other neurons. This is not possible in an asynchronous environment where all the signals are not available at the same time. They are being sent to the neuron, each in a different moment. Also, the data flow is reversed: another neuron sends a signal to the one performing computations and it may or may not react to this event by computing the output and sending it to other neurons.




It means that in an asynchronous neuron we need a buffer which will hold signals sent by other neurons. After adding a new signal to the buffer, the neuron compares their sum with the threshold to check if it is big enough to trigger the activation function. In this case the activation function simply presents the full strength signal – 1.0 – as the output. We can say that the activation function and the threshold together form a binary step function, with the step defined by the threshold, but in this case, in the asynchronous environment, the binary step function works in time. The signals are gradually

accumulated and the neuron is activated when their sum reaches the threshold. It may even happen that the input is just one neuron sending a very small signal, but repeatedly, and – given enough time – it will still trigger the activation function.

## 3.5 Synapses

Another property of a neuron is a set of synapses. In this project a synapse is defined as a simple data structure consisting of a reference to another neuron and either a weight, which is a number in the range of 0.0 to 1.0, or a special constant called “silence”. When the neuron fires, the output signal is multiplied by the synapse's weight and the result is sent to the neuron at the other end of the synapse ( the opposite direction of the data flow as described in the paragraph above). If the synapse's weight is set to “silence”, the signal is transformed into a “silence request”.

## 3.6 Silencing

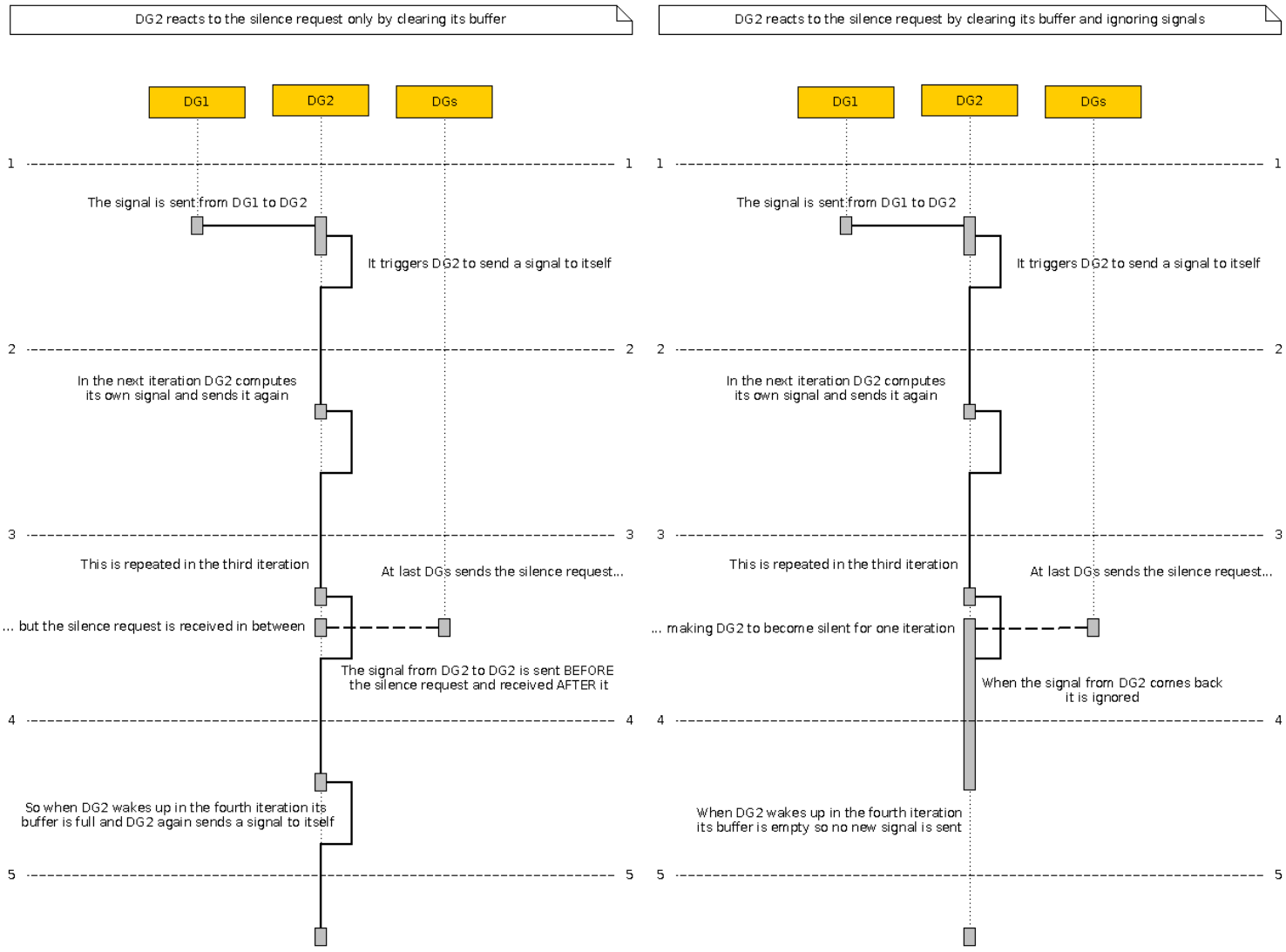
On top of these features an asynchronous neuron provides one more functionality: the ability to silence and to be silenced by another neuron. This functionality is crucial in a network where the neurons and blocks of neurons need to be assigned tasks of recognizing certain patterns in the input stream.
When one of the neurons signals that it recognized the pattern the signals of the other neurons need to be ignored. This is when we use one of the strengths of a neural network which lies in its ability to recognize patterns in noised input data . However, in certain situations this strength may become a weakness. Consider the following example:

1. The network is designed to recognize two patterns, P1 and P2. P1 is recognized through interaction of one block of neurons, P2 through interaction of another blocks. The blocks are not connected.
2. The input sequence S is a noised version of P1. It is similar enough to P1 that it should be recognized as such, but the modifications made by the noise make it also similar to P2.
3. S is sent to the network and is correctly recognized as P1.
4. However, the P2 neurons are not informed about the recognition. They recognize S as a noised version of P2.

5. In result, we are presented with both possible outcomes: P1 and P2.

To prevent this the P1 neurons should signal the P2 neurons that they should stop their computations because S has already be recognized as P1. This is done by creating a special synapse between neurons. A signal sent along this synapse is not added to the buffer, but it works as a message, telling the receiver to clear its buffer and, eventually, to “become silent” for a certain amount of iterations. This is very similar to a neuron sleeping for the rest of the iteration after it fired, but there is one significant difference: during sleep the neuron still accumulates signals in its buffer; when silenced the neuron ignores new signals. This feature is important if we don't want one neuron to clear its buffer but to stop a whole block of neurons.

In order to recognize a pattern in input data one neuron is almost never enough. This is why I introduce the concept of a “block of neurons”, which is explained in a subsequent chapter. Here it is enough to say that a block of neurons is composed both of the static part, that is neurons with certain thresholds, buffers and synapses' weights, and the dynamic part, that is signals sent between neurons belonging to one block. As it was mentioned already, in the asynchronous environment there is a moment in time when the signal has already been sent but is still not received. If the receiver is silenced in that very moment, but as a consequence it will just clear the buffer, a moment later it will receive that signal from before the silencing. The signal will be added to the buffer and if it is strong enough it may trigger the activation function, thus invalidating the silencing order. This is why sometimes it is useful to request the neuron to stay silent for one or more iterations. This way we can ensure that all signals sent from one silenced neuron to another silenced neuron will be ignored.

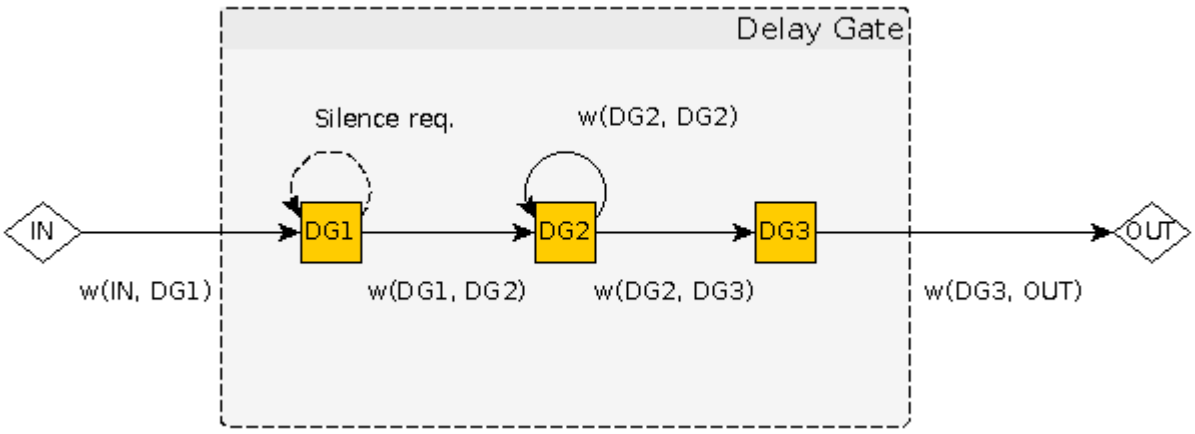



Silencing may be also useful for other purposes, like barring a block of neurons from receiving additional input while it is working. It is used for this purpose in the following example.

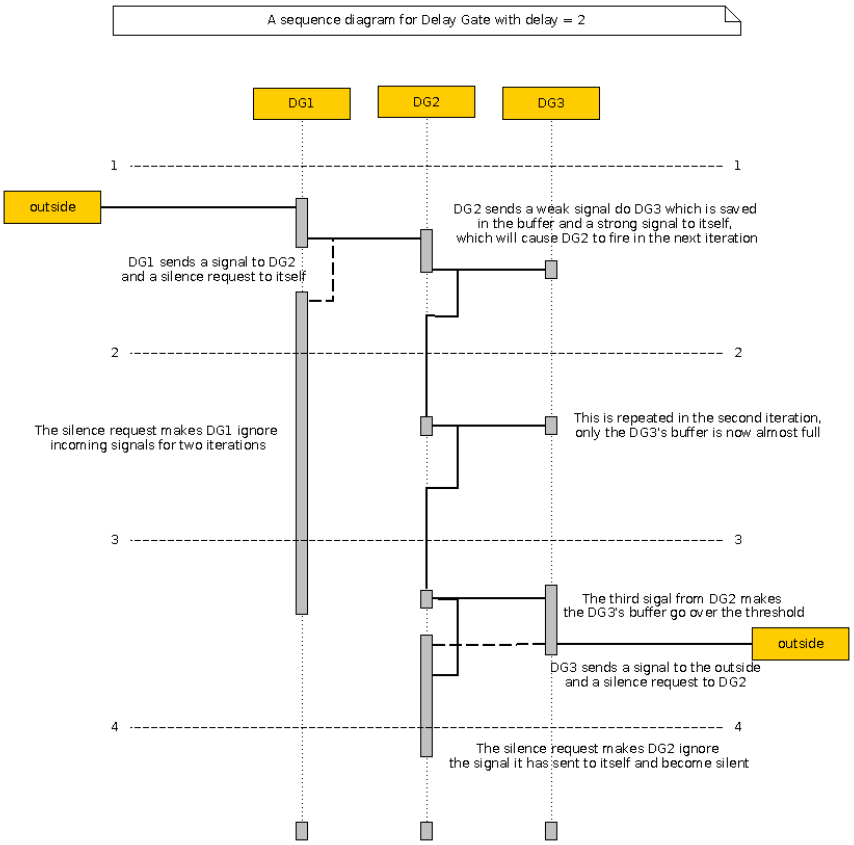
## 3.7 Blocks of neurons

For the purpose of this project, a block of neurons is defined as a set of connected neurons designed to perform a specific task. The simplest case of a block of neurons is a single neuron and its task is to fire when it receives a signal of a certain value, or a certain sum of signals. A more complex example of a

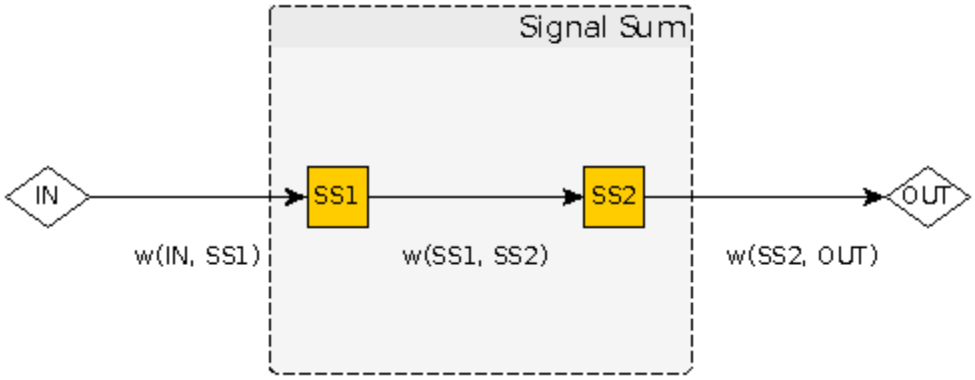
block is a Delay Gate (DG): a block which after receiving a signal ignores all subsequent signals for a given number of iterations, and then it fires its own signal and starts accepting new signals again. DG consists of three neurons connected as on the diagram below.

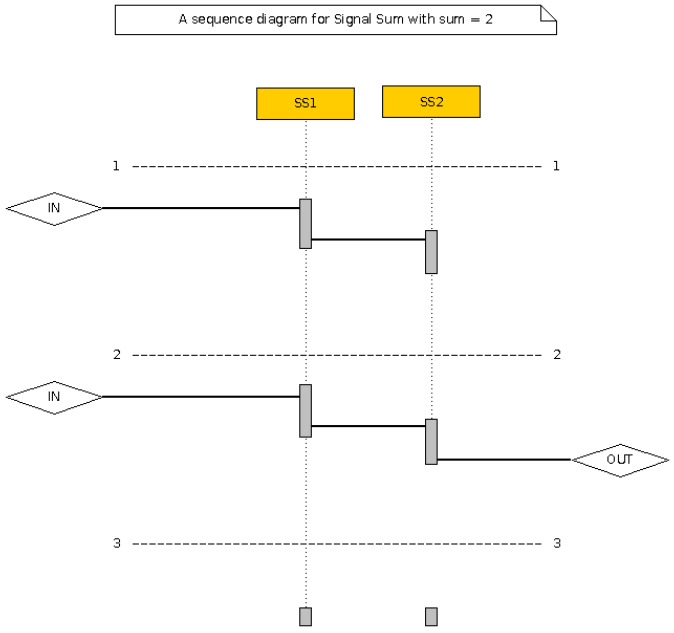



The neuron DG1 receives the input signal, immediately passes it to DG2 and silences itself for T iterations. DG2 passes the signal further to DG3, but the weight of the synapse (DG2, DG3) is low - it is exactly T+1 times lower than the DG3's threshold, meaning that DG2 has to send the signal T+1 times (one immediately after being triggered, then one during each subsequent iterations). In order to achieve that, DG2 uses a synapse to itself (DG2, DG2). A signal sent through (DG2, DG2) comes back to DG2 and in the next iteration triggers another signal from DG2, both to DG3 and again back to DG2. In the meantime, DG3 simply waits for its buffer to sum enough signals from DG2, which should happen exactly after N iterations. It then fires its own signal to the outside of the block. All the synapses in the block except from (DG2, DG3) have their weights equal to 1.0 which means that all the signals received from within the block except from the signal received by DG3 from DG2 have the value 1.0.




Another block used in the following example is the Signal Sum (SS), consisting of two neurons, SS1 and SS2. SS1 receives signals from the outside and if they are strong enough, or if the sum of subsequent signals is big enough, it sends its own signal to SS2. The weight of the synapse (SS1, SS2) is exactly N times lower than the SS2's threshold, so SS2 will fire only after receiving N such signals. The Signal Sum can be simplified to a single neuron. The difference between them is that a Signal Sum block encodes the requested sum of signals in its internal synapse. This way the neurons connecting to the block do not have to know about the requested sum. They can simply send their full signals and the Signal Sum block will handle all the computations.

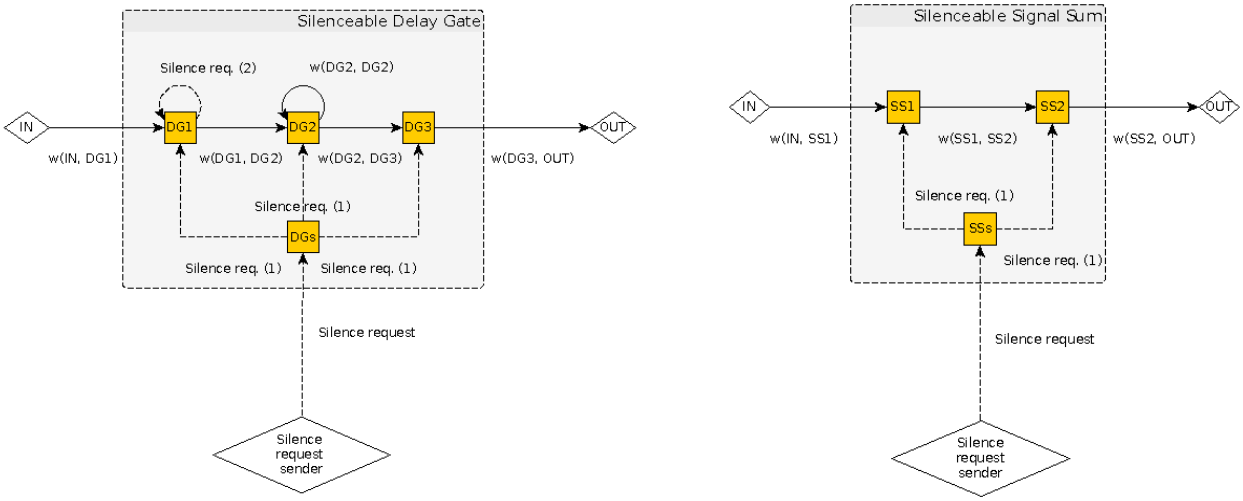



These two building blocks are enough to build an S.O.S. network from the main example, but one more property is needed. A silenceable block is a block with one of its neurons being a silencing neuron which, when triggered by any input, sends silence requests to all other neurons in the block. Thus it works as a single point of entry for silence requests from the other parts of the network. A neuron which wants to silence a silenceable block does not have to know about all the neurons in the block. It just has to know how to send a signal to the block's silencing neuron.

In the rest of the article I will assume that both DG and SS blocks are silenceable blocks, each with a silencing neuron connected to all other neurons in the block. All neurons except for DG1 are set to stay

silent for one iterations. This is especially important in the case of DG2 which can send a signal to itself. It may happen that the silence request comes just after it sends such a signal. If the silence request would cause the neuron only to clear its buffer, but not to ignore messages coming to it during the next iteration, it would simply receive the signal, add it to the buffer, and fire again, resuming its work.



# 4. The S.O.S. network example

The following example shows how to build a network which is able to recognize an S.O.S. message in an input stream. The data chunks in the input stream will actually consist of only one symbol – let's call it '1'. The symbol '0', used in the diagrams and the description, is only there to symbolize the lack of signal for the given iteration. A dot in the Morse code will be decoded as (1, 0, 0, 0) – a signal for one iteration followed by a three iterations long gap. A line will be decoded as (1, 1, 0, 0) – a signal for two iterations followed by a two iterations long gap. The importance of the time gap is discussed at the end of this section.

Internally, it means that one of the neurons in the network is tagged as the “dot” neuron and another as the “line” neuron. If the network receives only one '1', the “dot” neuron should fire. If the '1' symbol is immediately followed by another '1', “the” line neuron should fire. The network should clearly distinguish between the two: it should not answer that what it received is both a dot and a line, nor should it answer first that the received input is a dot and then, without receiving any new data, say it is a line, or the other way around.

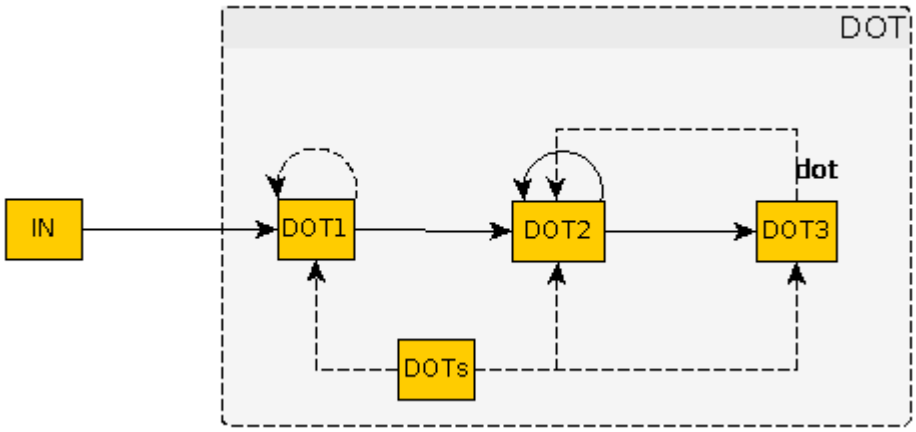
Being able to recognize dots and lines, the network should store this information in its internal state until it receives enough data to proceed further. Three dots, one after another should make the network produce the 'S' symbol, that is, a neuron tagged as the “S” neuron should fire. Three lines one after another should result in the 'O' symbol. If there is not enough input data, or dots and lines are mixed, the network should not produce anything.

## 4.1 The procedure

The following procedure will construct a network fulfilling these requirements.

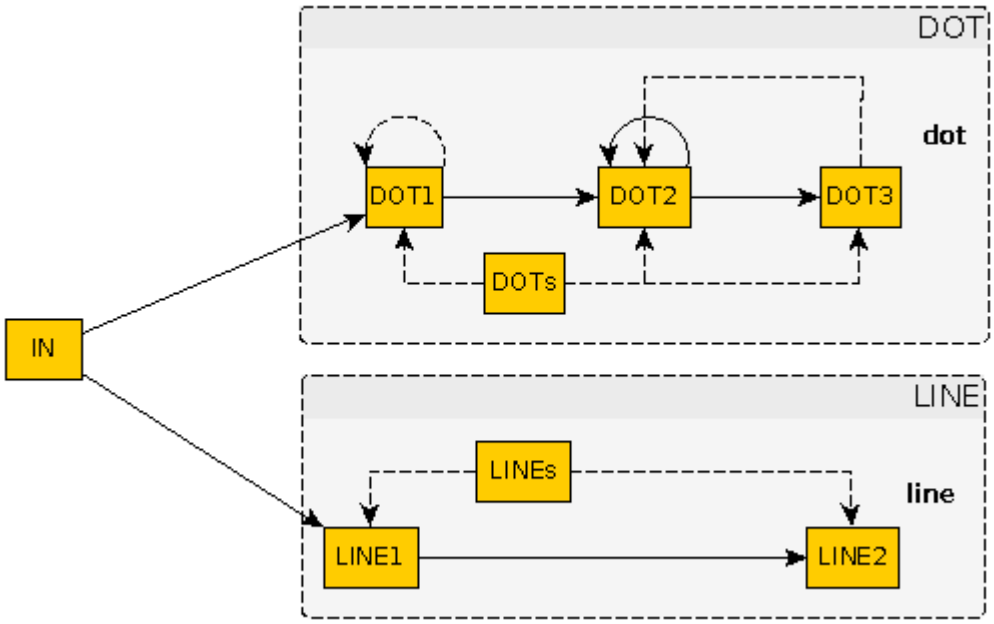
1. Create a dummy input neuron IN.
2. Connect In to a block of the Delay Gate type, called DOT, with a full weight (1.0) synapse. The DOT block has the delay set to 2, meaning that after receiving a signal it will wait for two

iterations and then produce the output signal. The output neuron DOT3, is tagged as the "dot" neuron and if it fires, it means a dot was recognized by the network. Please note that if the delay of the DOT block was shortened to 1, exactly the same number of iterations would be needed for another block to recognize a line from the signal sequence (1,1) as it would be for DOT to recognize a dot. The outcome would then depend on which signals move faster – something we want to avoid.



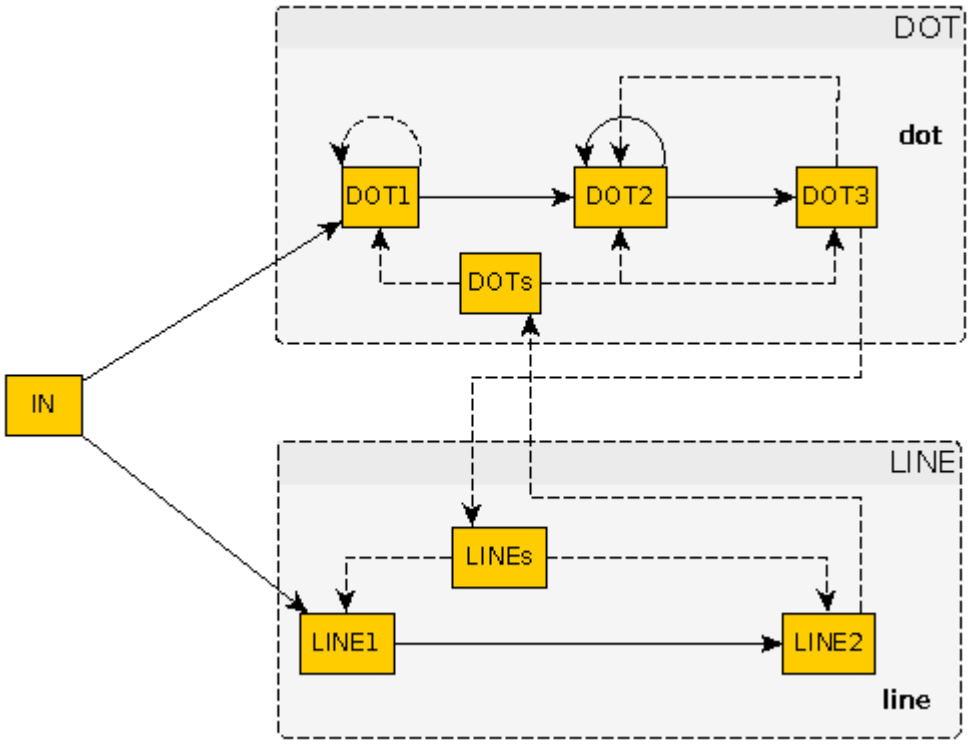

Every signal '1' sent from IN to DOT1 when DOT is at rest will be, after two iterations, recognized as a dot and DOT1 will ignore any signals sent to it from IN during these two iterations. Without additional information coming from other blocks, signal sequences of the type (1,0,0,0), (1,0,1,0), (1,1,0,0), (1,1,1,0), (1,1,1,1) will be recognized as a dot.

3. Connect IN to a block of the Signal Sum type, called LINE, with a full weight (1.0) synapse. The LINE block is set to fire its output neuron LINE2 after two signals '1' coming from IN to LINE1. The output neuron LINE2 is tagged as the "line" neuron".



The signals from IN to LINE1 do not have to be consecutive. Without additional information coming from other blocks, signal sequences of the type (1,1,0,0), (1,0,1,0), (1,0,0,1), as well as others with more than one '1', will be recognized as a line.

4. Connect DOT3 to LINEs and LINE2 to DOTs. If a dot is recognized, LINEs will receive a signal from DOT3 which will cause the LINE block to drop its computations. If a line is recognized, DOTs will receive a signal from LINE2 which will cause the DOT block to drop its computations.




Signal sequences (1,0,0,0) and (1,0,0,1) will be recognized as a dot – the latter due to the fact that the DOT block will fire during the third iteration, making LINE forget the first '1' signal. The sequences (1,1,0,0) and (1,0,1,0) will make the LINE block fire and make DOT drop its computations. The sequence (1,0,1,1) will be recognized as LINE as well if sent only once, but if we send many such sequences one after another (e.g. (1,0,1,1,1,0,1,1) ), the first sequence will affect the computations of the next one and will result in an additional DOT.

5. Connect DOT3 to a block of the Signal Sum type, called S, with a full weight (1.0) synapse.

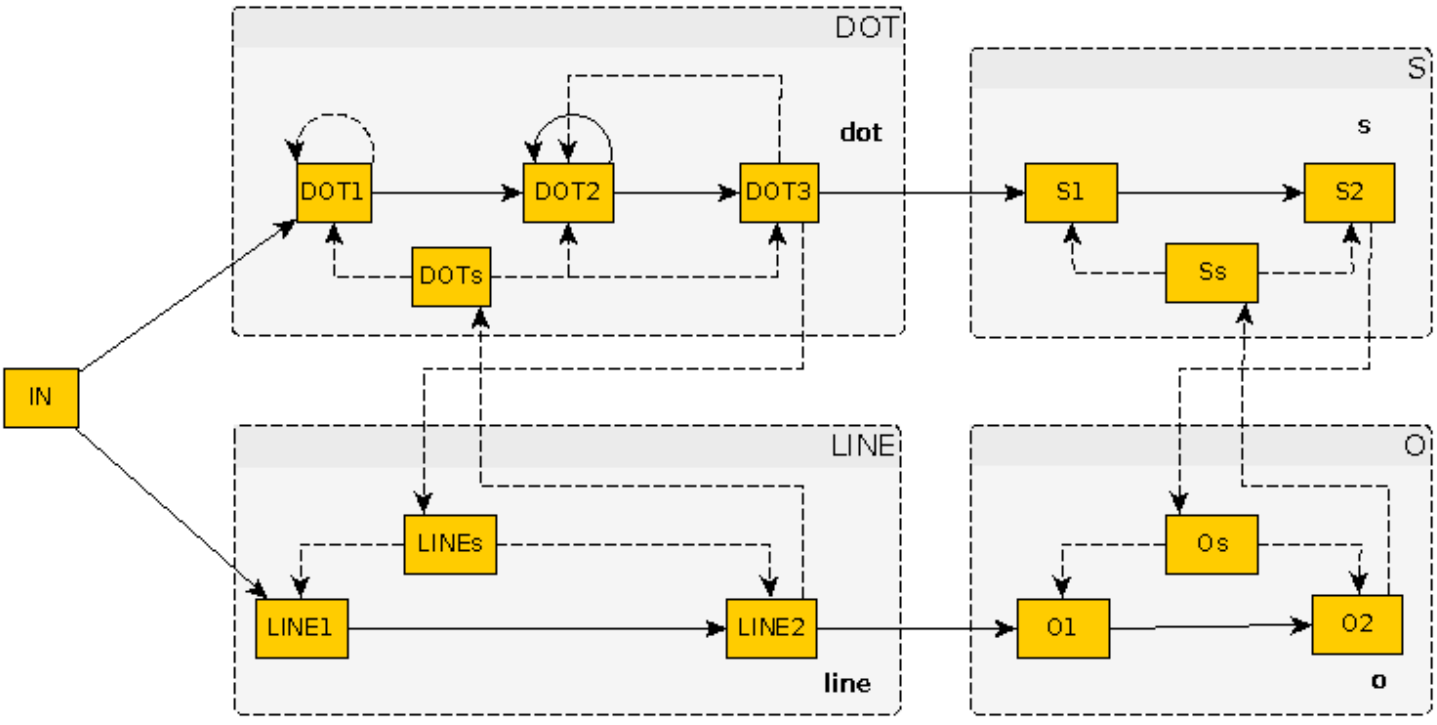
The S block is set to fire its output neuron S2 after three signals '1' coming from DOT3 to S1. The output neuron S2 is tagged as the “s” neuron.

How does it handle noised signals: There is no time limit for signals coming from DOT3. Without additional information coming from other blocks, any three “dots” – with whatever time gap between them – will be recognized as “s”.

6. Connect LINE2 to a block of the Signal Sum type, called O, with a full weight (1.0) synapse. The O block is set to fire its output neuron O2 after three signals '1' coming from LINE2 to O1. The output neuron O2 is tagged as the “o” neuron.

   How does it handle noised signals: There is no time limit for signals coming from LINE2. Without additional information coming from other blocks, any three “lines” – with whatever time gap between them – will be recognized as “o”.

7. Connect S2 with Os and O2 with Ss. If “s” is recognized, Os will receive a signal from S2 which will cause the O block to forget about “lines” which came to it. If “o” is recognized, Ss will receive a signal from O2 which will cause the S block to forget about “dots” which came to it.



How does it handle noised signals: It is an “all or nothing” competition. The noised signals coming from IN will be recognized as either dots or lines and whichever of the blocks of the next layer – S or O – first receives three dots or three lines respectively, is the winner.

The network's wrapper will react to the activation of “s” and “o” neurons by putting symbols 'S' and 'O' in the output stream. Therefore if the input stream will provide three short signals (3 times (1, 0, 0, 0)) then three long (3 times (1, 1, 0, 0)), and again three short ones, the output stream will be provided with symbols 'S', 'O', and again 'S'. Slightly noised signals will be recognized correctly, especially in the case of noised long signals.

| Input | Output | Input | Output |
|---|---|---|---|
| 1,0,0,0,1,0,0,0,1,0,0,0 | **… → s** | 1,1,0,0,1,1,0,0,1,1,0,0 | **--- → o** |
| 1,0,0,0,1,0,0,0,1,0,0,**1** | **… → s** | 1,1,0,0,1,1,0,0,1,**0**,**1**,0 | **--- → o** |
| 1,0,0,0,1,0,0,**1**,1,0,0,0 | **..-** | 1,1,0,0,1,**0**,**1**,0,1,1,0,0 | **--- → o** |
| 1,0,0,**1**,1,0,0,0,1,0,0,0 | **.-.** | 1,**0**,**1**,0,1,1,0,0,1,1,0,0 | **--- → o** |
| 1,0,0,0,1,0,0,0,**0**,**1**,0,0 | **… → s** | 1,1,0,0,1,1,0,0,1,**0**,**1**,**1** | **---. → o** |
| 1,0,0,0,**0**,**1**,0,0,1,0,0,0 | **..** | 1,1,0,0,1,**0**,**1**,**1**,1,1,0,0 | **---. → o** |
| **0**,**1**,0,0,1,0,0,0,1,0,0,0 | **.-** | 1,**0**,**1**,**1**,1,1,0,0,1,1,0,0 | **--.- → o** |
| 1,0,0,0,1,0,0,0,**0**,0,**1**,0 | **… → s** | 1,1,0,0,1,1,0,0,1,1,**1**,0 | **---. → o** |
| 1,0,0,0,**0**,0,**1**,0,1,0,0,0 | **.-** | 1,1,0,0,1,1,**1**,0,1,1,0,0 | **---. → o** |
| **0**,0,**1**,0,1,0,0,0,1,0,0,0 | **-.** | 1,1,**1**,0,1,1,0,0,1,1,0,0 | **--.- → o** |
| 1,0,0,0,1,0,0,0,**0**,0,0,**1** | **… → s** | 1,1,0,0,1,1,0,0,1,1,1,**1** | **---- → o** |
| 1,0,0,0,**0**,0,0,**1**,1,0,0,0 | **.-** | 1,1,0,0,1,1,**1**,**1**,1,1,0,0 | **---- → o** |
| **0**,0,0,**1**,1,0,0,0,1,0,0,0 | **-.** | 1,1,**1**,**1**,1,1,0,0,1,1,0,0 | **---- → o** |

## 4.2 Noise handling versus conciseness of input

In the mathematical models, the ability of the network to handle noise depends on the Hamming distance between input patterns [10]. If the distance between patterns P1 and P2 is too small, even a slight noise applied to the pattern P1 may result in the signal sequence which is closer to P2 and will be recognized as such by the network. In order to avoid that we need to alter the patterns, making the

distances between them bigger, so the same noise applied to the pattern P1 will result in R still being closer to P1 than to P2.

It is different in case of an asynchronous network: it is enough to add another '0' to the end of a signal sequence. It is equivalent to giving the network more time to process the information. In our example, this additional time is used to delay the DOT block. Shorter versions of DOT and LINE sequences – (1,0,0) and (1,1,0) – will be recognized correctly if sent in isolation. However, if sent one after another in a longer sequence, the computations of precedent signals will affect the computations of the following ones.

| Input | Output | Input | Output | Input | Output |
|---|---|---|---|---|---|
| 1,0,0 | . | 1,0,0,1,1,0 | .- | 1,1,0,1,0,0,1,1,0 | -.- |
| 1,1,0 | - | 1,1,0,1,0,0 | -. | 1,1,0,1,1,0,1,0,0 | --. |
| 1,0,0,1,0,0 | .. | 1,0,0,1,0,0,1,1,0 | .- | | |
| 1,1,0,1,1,0 | -- | 1,0,0,1,1,0,1,0,0 | .-. | | |
| 1,0,0,1,0,0,1,0,0 | ... | 1,1,0,1,0,0,1,0,0 | -. | | |
| 1,1,0,1,1,0,1,1,0 | --- | 1,0,0,1,1,0,1,1,0 | .-- | | |

All the examples discussed in this chapter can be tested with the ANNA program which can be found at https://github.com/makingthematrix/ann .

## 5. Learning

This article does not discuss any learning techniques for the asynchronous network. The way the network is built – with blocks of neurons, each prepared for a specific purpose instead of a general structure, makes it difficult, if not impossible to design an error correction algorithm. Instead, usage of genetic algorithms seems to be much more promising. Please note that the S.O.S. network discussed above is made of only four blocks, each having only one variable, one input, and one output neuron. In the case of the DOT block the variable is the required delay. In the case of the LINE block - the sum of required signals. This simplicity makes the S.O.S. network easy to describe as a sequence of terms similar to the pseudocode below:

```
Create:
Input(“IN”)
DelayGate(“DOT”, 2)
SignalSum(“LINE”, 2)
SignalSum(“S”, 3)
SignalSum(“O”, 3)

ConnectWith:
Signal(“IN”, “DOT”, 1.0)
Signal(“IN”, “LINE”, 1.0)
SilenceRequest(“DOT”, “LINE”)
SilenceRequest(“LINE”, “DOT”)
Signal(“DOT”, “S”, 1.0)
Signal(“LINE”, “O”, 1.0)
SilenceRequest(“S”, “O”)
SilenceRequest(“O”, “S”)
```

```
Tag:
Output(“S”, ”s”)
Output(“O”, ”o”)
```

Instead of using a genetic algorithm to directly modify neurons and synapses, we can step up to a more abstract level and use it to manipulate such sequences of terms which then can be used to create the network. A mutation on this level (a change to the block's type, to the block's variable, or to which blocks are connected) is much more likely to result in an interesting, even though usually invalid change to the output of the network, than an analogous change on the neuron level. The genetic algorithm could therefore start from a some kind of a template, i.e. a network with the correct input and output neurons, and add, modify, and delete terms which would subsequently be used to build and test the network.

# 6. Conclusions

The article touches only basics of the asynchronous approach to artificial neural networks. Nonetheless, it is enough to show certain interesting features, such as how the passage of time can be treated as information. Moreover, this ability comes naturally from the fundamental traits of a system composed of asynchronous, reactive actors communicating through messages, such as Akka. Akka actors work in a way similar to how a network of organic neural cells cooperate in order to transmit and parse information. Each of them is connected only to a limited number of other cells, uses simple signals to communicate and works without any kind of supervision or knowledge about the environment other than its neighbour cells. An artificial neural network implemented with Akka may then give us a glimpse on how our brains perceive time at the most basic level.

In total, in the S.O.S. example, I used only 14 neurons: 9 standard, 4 silencing and 1 input. Apart from the input neuron they were grouped in four blocks with each block belonging to one of only two types. The S.O.S. network is non-trivial, which suggests that, even though there is no general structure to the network, more complex networks can be constructed with relative ease, by designing other types blocks and making them cooperate, either by design or by using genetic algorithms. For example, if we want to create a network which recognizes all letters of the alphabet from the Morse code, we can use the S.O.S. network, and simply add to it blocks which recognize the rest of the letters in addition to “s” and “o”.

Asynchronicity and reactivity of the network means that this approach is easily scalable. Even though it is too early to talk in detail about practical application of this approach,my work proves that asynchronous networks can be used for cloud computing with relative ease. For example they can be applied for speech recognition, or to monitor electromagnetic spectrum in search for patterns which might suggest an intelligent agent trying to convey information, or the occurrence of a predicted event, such as a new pulsar flare. In all these cases the asynchronous network has an advantage over other solutions in that it could be extremely easily distributed over many CPUs, or even many servers, if the latency is small enough.

# 7. Bibliography

1: Kevin Gurney, An Introduction to Neural Networks,
2: Shashi Sathyanarayana, A Gentle Introduction to Backpropagation, 2014
3: Sander Marcel Bohte, Spiking Neural Networks, 2003
4: Xin Wang, Qingnan Li, Edward K. Blum, Asynchronous Dynamics of Continuous Time Neural Networks , 1993
5: Jacob Barhen, Vladimir Protopopescu, Preventing Computational Chaos in Asynchronous Neural Networks , 2002
6: Jonas Bonér, Introducing Akka - simpler scalability, fault-tolerance, concurrency & remoting through actors, 2010,
7: Ingo Maier, Tiark Rompf, Martin Odersky, Deprecating the Observer Pattern, 2010
8: Jonas Bonér, Dave Farley, Roland Kuhn, Martin Thompson, Reactive manifesto, 2014,
9: Lightbend, Inc, Akka Scala Documentation, 2016
10: Mohammad Norouzi, David J. Fleet, Ruslan Salakhutdinov, Hamming Distance Metric Learning,

## 8. Acknowledgments

I would like to thank Jakub Nowacki and Michał Strojnowski for beta-reading and meritorical advice, and Lidia Wojtal for language check and support.