# A new Neural Network

By Jan Bogaerts

Introduction

This paper will outline a new approach to the art of neural network development compared to previous attempts in order to provide better control, flexibility and power.

The past

Neural network algorithms have been known and used since the 1950-60's and have been studied since. They were initially designed based on the observations done by biologists on the human brain. Since then, this basic design hasn't changed very much.

These networks are made up of neurons which are connected to each other. Each neuron has a weight that determines when input is translated into output and/or a function that determines how input is distributed over it's outputs. Input can come from other neurons, or from starting points, outputs can go to other neurons or output points. By changing which neurons are connected to each other and varying the way that the weight of each neuron is determined or which function is used, different results can be achieved.
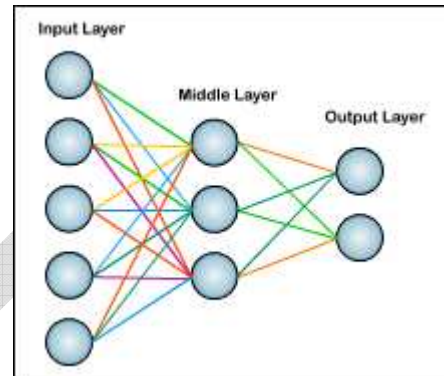
Off course, this type of network is very hard to design and control manually because of the enormous amount of neurons and connections between them for any 'usable' system that actually does something. This hurdle is usually taken by 'training' the networks instead of manually designing them. For a set of input values, the output is calculated and provided to the network, which will adjust the connections and weights so that it can calculate the result with the given input values and taking previous training values into account. Most learning algorithms are a variation on this theme. Recently, natural selection algorithms have also been used in combination with neural networks.

From biology to logic

Traditional neural networks, as already mentioned, have been notoriously difficult to create, maintain and control. Creating the correct initial links and providing the proper weight-function is, simply put, an un-mastered art. Furthermore, it is my opinion that the current algorithm will never be able to solve these issues. Instead, a new one should be designed, using a different approach, mostly borrowed from the world of software design.

When this community was still in its infancy, it faced similar problems although be it in a smaller grade. Writing modern, complex applications in pure assembly is simply put, too difficult. So higher level programming languages were created which are able to generate assembly, design techniques were developed and libraries have been created.

I believe that a similar approach should be taken to the neural network paradigm. Design tools able to generate neurons and links, libraries that offer predefined capabilities and techniques describing best practices should be made available to the future neuro designers. Before this is possible however, a shift is required from neural network algorithms, seen as applications, to a platform with processors and programming languages.

New elements

A shift doesn't mean a complete make-over. What was known is known and will therefore be the basis of the new. In other words, a neural network platform should run on neurons which are connected to each other. These neurons only have a possible weight value (it's use will become clear later on), incoming and outgoing links, nothing more. From there, we can build.

The processor

For instance, we will need a processor with instructions and a way to control the execution flow of this processor like traditional programming languages do. As you will see, this will eventually result in a change of focus from the '*neuron*' to the '*link*'.

In order to understand the role of a processor and it's interaction with the network, consider this:

- A single processor represents a single '*thought*' which can be split up into multiple possibilities,
- the links between the neurons determine the path that the processors traverse over the network and
- The processors create the structure of the network by executing the provided instructions which can create links and neurons.

But where exactly do those instructions come from, who provides them and how is the execution process controlled? This requires more elements like sensory interfaces, clusters, expressions and the instructions themselves.

Sensory interfaces

A neural network only becomes useful if it is able to interact with the outside world through input and output. The form of this interaction depends on the type of interface. Different interfaces could be:

- a Textual sin for keyboard input, or link to the web
- a mouse sin for tracking mouse positions
- a UI automation sin for controlling and monitoring windows, buttons, combo boxes and the likes,
- a reflection sin for providing a way to call traditional functions or for generating assembly code
- an audio interface which can translate audio samples into neurons
- a video sin which does the same for images
- or a sin for analog-digital-analog converters

Output is always triggered by a processor through an instruction. It's result depends on the sensory interface: some will generate a string on the screen, others might generate a sound. Input however is a different matter. As with biological entities, this has to be triggered from the outside. For computers, this means interrupts or hooks. All sensory interfaces also behave similar with regards to input: it triggers a thought. In other words, a processor run is started.

Links revisited

But what exactly is a processor run? To explain this, we first need to revisit the linking system between neurons. So far, they only have direction (this comes from the fact that a neuron has incoming and outgoing links, so the direction is known).

Things become far more interesting if we also give them a '*meaning*'. Now we can express something through a link. For instance, we can create a 'has' or 'contains' link from one neuron to another. And because our network only knows neurons and links, this meaning also becomes a neuron.
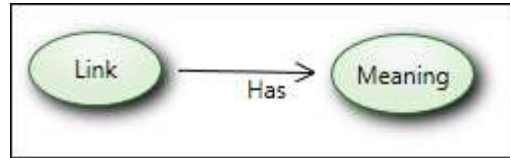
Now, suppose that a sensory interface, upon input, creates a neuron that has a number of links to other neurons, each link having a meaning. To initiate the thought, this neuron is sent to a processor which will process each link meaning.

As an example, take a text-sin, which, upon receiving a text string, splits it into it's characters. For each character, a new text neuron is created and finally, a last neuron is created to represent the entire string which links to all the character neurons, using the meaning 'Character'. This can then be translated by a processor, first to words, next to sentence parts and perhaps finally back as an output to another sensory interface.
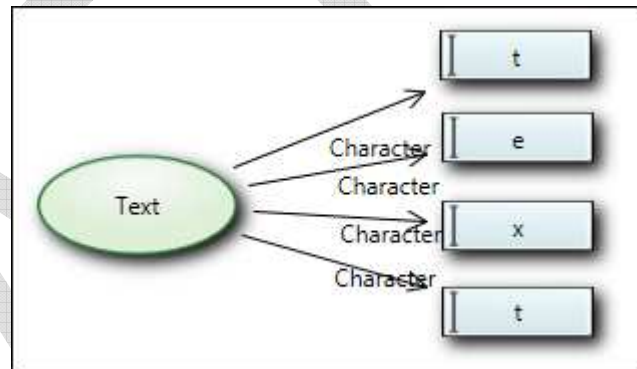
In this example, a new neuron is created for each character instead of reusing the same neurons for the same characters. This is done because some text strings might use the same letter multiple times. However, it is not possible to create 2 links between the same neurons with the same meaning because in that case, we would no longer be able to uniquely find a link, which is important for the search algorithms. As a side effect, this results in a neural network that is basically, always learning. Not learning needs to be performed explicitly by deleting newly generated neurons and destroying the created links.

Clusters, expressions and instructions

To perform the translation from letters to words and further, a processor requires some extra information, notably, the expressions (which determine the execution flow) and the instructions (which manipulate the network). In other words, it needs to know the application that it needs to run.

The expressions are, like the rest of the system, also neurons. The only difference with regular ones is that a processor recognizes them as being of a specific expression type (like an 'assignment', 'variable', 'code block' or 'Boolean expression') and therefore expects to find some predefined outgoing links on them which are the operands for the expression. If these are all found, the processor is able to 'perform' these expressions.

One of those, the '*statement*', is responsible for preparing the parameter values and calling an instruction, both of which are provided through links between the expression and the instruction -

list of parameter values). This means that a processor never really directly calls the instructions, it only calls the expressions and some of them are responsible for calling instructions. It also means that instructions must also be neurons, because we need to be able to link to them. These must be *hardwired* though since all the instructions must be known to a network and it is therefore not able to create new ones.

Finally comes the step of putting all these pieces together. As previously mentioned, expressions must be stored in a list (both as parameter values for the instructions and as a single code block for the processor). For this task, the '*cluster*' is created. This is essentially again just a neuron which is able to group other neurons together into a single set. Think of it as an array, list or collection. And, just like a link, clusters also have a meaning which is expressed by another neuron. So a cluster can be assigned to be a 'code' cluster or simply as a 'list'. Unlike links though, a cluster can contain the same neuron multiple times without any problem.
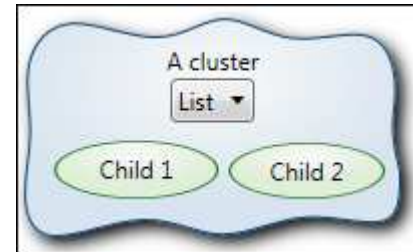
The list of parameter values, as previously mentioned, is assigned to the '*statement*' expression through a link, with the meaning '*arguments*'. This way, the expression can find it's own parameter values. But how does the processor find the list of expressions to execute for each link that it needs to solve? Well, that's where the '*meaning*' on the link comes into play. If this neuron has a link to a cluster where the link has a meaning of 'rules', it will execute all the expressions found in this list. If the processor finds a link with the meaning 'actions', the cluster to where it points to, is stored and finally executed when all other links have been processed.
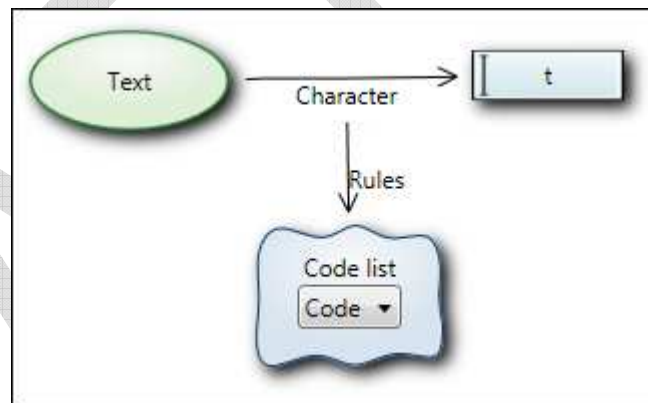
Also

As we are in the digital world, it can be useful to have some additional types of neurons, more akin to the micro processor world. Text neurons, for instance, are able to store a string value, int and double neurons can do the same with an integer and double respectively. These types of neurons provide a way to work with and store byte information in a neural network.

Another specialized neuron type is the '*Timer*'. This is actually another sensory interface, one which will execute a list of expressions every time a 'tick' is passed. Output sent to it, is interpreted as commands that control the timer function. This type of neuron plays an important role in the process of connecting different '*thoughts*' together.

About stacks, splits and weights

So far we have a very flexible system, but no *thoughts*. These are actually born out of the interaction between the processors and the code: a processor uses a *stack* to store the neurons it has to process and for as long as there is something on this stack, it will pop and process. Because there is an

instruction to put new neurons on the stack, this processing can be made to run indefinitely. This is also why the processor treats an 'actions' link differently (only execute the code when all other links have processed): it is the perfect time to clean up or add items to the processor stack since all links have been processed.

Furthermore, thought is characterized by uncertainty and multiple possibilities. A logical neural network should be able to handle this, just as a biological one can. For this purpose, processors have the ability to split themselves into multiple similar sub-processors which have the same variable values and execution position as the main processor at split time, the value of only 1 variable is different and all the neurons on the stack are cloned into new neurons with links pointing to the same values.

After the split, each processor continues to process from the same position, but since there is 1 difference, the variable values and stack content might and should begin to differ over timer for all sub processors. If the code doesn't put new neurons on the stack, a processor dies out, so do sub processors. If however, a sub processor is stopped through a special instruction, the neurons remaining on the stack will be copied to a cluster (assigned by the instruction). Once all sub processors have stopped, a callback is triggered which allows the system to call extra code.

This is usually when the *weight* of a neuron begins to play a role. The system has instructions to increase or decrease this value for a neuron. If a high probability is found, the weight of the neuron can be increased. The same goes for situations that are possible, but not likely: the weight can be decreased.

 If the code in each sub processor does this for its own neuron(s), the callback function can be used to pick the neuron with the biggest weight value out of the *result* cluster. This technique effectively simulates the idea that the result path is that which arrives first.

The combination of splits and weights allows for a very efficient and highly flexible search system. The idea is that whenever a search in a cluster list, links-list[1] or link-info list (an extra list of neurons that can be assigned to a link) returns multiple results instead of 1, a split is performed which results in multiple processors who will work on the *same* neural network, but with 1 difference and with unique result neurons. These sub processors will continue to transform and verify their result values until verification fails or a sub processor dies out. At this point, the callback is run and normal execution can continue. The actions to perform in this callback are not defined, the result list can be used as-is, or one or more neurons can be picked, possibly based on weight. Usually though, if there was only 1 result, this is used, but if there were multiple possibilities, the system performs more inquiries to solve the confusion.

Compound types

If you put all these new elements together, what you have really is just another database management system able to store, search and manipulate datasets. Of course, at this point, we don't know any datasets yet, we just have neurons and links. But we have already seen some examples of

---

[1] A links-list = a list containing only links. A neuron has 2: one for incoming links one for outgoing.

compound objects: the expressions which need to have a specific number of outgoing links with predefined meanings in order for the processor to be able to handle them. In other words, a compound object is a set of interlinked neurons that follow a predefined pattern: the compound type which declares the types of neurons and links that need to be present.

Except for the expressions, the network doesn't require any specific compound types; the user is free to create what he/she wants. There are some types however, which are very useful in the development of artificial intelligence.

Objects

The most basic of these types is the '*Object*'. WordNet[2] has synsets, FrameNet[3] has lemmas, and a neural network has objects. Basically an object represents a single instance of something, like 1 interpretation (meaning) of a word.

Structurally, it is nothing more than a cluster with the meaning 'Object' (a predefined neuron) and any type/nr of children. Normally, each child is of a unique type in the list, like a text neuron, and a regular one. This is not required though. The idea is that all the children are a different representation of the same object: one is a text value, (which could be shared among many objects, if the same word has multiple meanings), the other can be used as the meaning for a link, referencing code. In this scenario, some neurons function as 'entry points' to the object from other parts of the network (like text sins, which have a dictionary of registered text neurons).

Multiple relationships between these 'Object clusters' can be expressed through links. For instance, it is very easy to load all the thesaurus data provided by WordNet into a neural network, where each string can be searched and transformed into a text neuron. This text neuron is stored in each object cluster that represents a unique WordNet synset. This also means that you are able to find all the object clusters a text neuron belongs to, from that text neuron since all neurons know the clusters they belong to. During searches, a split is usually performed at this point to find only 1 correct value from the multiple results. Because of the links provided by the thesaurus (like synonyms, hyponyms, hypernyms,…) on all the objects, it is possible to jump to different objects that are better known/defined.

Frames

Objects alone are not enough to define meaning. Semantics needs a different form of compound type. The basic layout was mostly defined by the FrameNet project. This uses a frame to express a semantic relationship between a number of words/word groups. This frame contains all the possible parts of a sentence (frame elements) and the triggers (lexical units – single meaning of a word) for the frame.

As an example, let's take a simple frame to express the relationship between an 'attribute' word ('a', 'the') and a noun ('word', 'cat'); ex: the word, a cat, a house, the mouse,...). We have already defined the frame elements: an attribute and a noun, which are also represented by object clusters. Since the set of attributes is small compared to that of the nouns, and more importantly, it is relatively

---

[2] WordNet is a lexical database of words which can be used as a dictionary and as thesaurus. It contains over 100.000 words and relationships between them and is provided under a BSD style of license.
[3] FrameNet is a semantic database of frames with a large set of annotated sentences. It is housed at the International Computer Science Institute in Berkeley, California.

fixed as in it won't change much over time compared to the list of nouns, it is probably best to use all the attribute words as triggers for the frame. Finally, the possibly sequences are also important. In this example, we can only have 1: first the attribute, next the noun, but in some situations, multiple are allowed (think about the difference between a statement and a question, this can be simply a different order of words). This sequence information however is not defined in FrameNet, but is used by the Compound type.

This is a cluster with the meaning '*Frame*' (also a predefined neuron) that contains other clusters with the meaning 'Object' or 'Frame' which represent all the different frame elements. It should also have an outgoing link to a cluster containing again 'Object' or 'Frame' clusters, representing the triggers for the frame. Each sequence is a cluster with the meaning 'Sequence' containing one or more of the clusters representing the frame elements. All the sequences for a frame are grouped into another cluster to which the frame points to using a link with meaning 'Sequences'.

Note the recursive nature of frames: Frame elements and triggers can either be objects or other frames. This allows a designer to split up as many frames as possible into the smallest possible set: 2 items.

Flows

Another way of expressing semantic meaning is through flows. These can best be compared to EBNF[4] style of languages which are mostly used as compiler/parser generators because of how well they are suited to describe the '**what'** but skip the '**how'**.

Flows perform the same task; they describe which objects, frames or possibly other neurons to expect in a stream of neurons coming in from one or more sins.

Flows are very similar to Frames. Both compound types are able to describe the same semantic relationships. It is just a different approach, different syntax. Flows however are more verbose and allow describing a grammar more compact.

Compound type definitions

The following section contains a more formal description of the previously described compound types. Note that this is a preliminary design. For a definition of the compound type definition language that was used, please see Appendix 1.

```
Object = {Any}-Object.

Frame = {Object|Frame}-Frame
        <-FrameEvokers(Evokers)
        <-FrameSequences(Sequences).

Evokers = {Object|Frame}-FrameEvokers.

Sequences = {Object|Frame}-FrameSequences.

Flow = {Conditional | Sequence}-Flow.
```

---

[4] EBNF: Extended Backus–Naur Form, a meta language to describe other languages.

```
Conditional = {ConditionalBlock}-FlowConditional
              <-IsLooped(True | False)
              <-FlowItemRequiresSelection(True | False).

ConditionalBlock = {Conditional | Sequence}-FlowConditionalBlock.
                   <-FlowCondition(BoolExpression).

Sequence = {Object | Frame}-FlowSequence.
```

Putting it all together

Using the above described system, I believe it is possible to implement applications able to simulate intelligence. Depending on the initial network, the search algorithms and result checkers that are used, different levels of intelligence should be achievable. But how should these search algorithms and result checkers work together to form an application and what initial network structure should be used?

With respect to the structure, the frames, objects and the relationships between them naturally take up a large chunk. But just as important is the presence of some sort of '**Current**' neuron that serves as the entry point used by the verification routines to check various states that are currently valid or enabled. The internal structure of this 'current' neuron is up to the application. It could be as simple as keeping track of a list of objects in its vicinity, or a more complex structure, divided into hierarchies of importance or conversation structure. In any case, it is the responsibility of the application to keep it up to date by adding, changing and removing links and or changing the children for clusters.

A basic application flow structure could consist out of the following steps:

- Extract the individual words/items (in case of visual or other forms of data input) out of a stream of letters, phonemes or data points.
- Find the objects that go with the words/items
- Extract the frames out of the stream of objects. This is highly recursive and probably interacts with the previous step.
- Whenever a frame sequence is solved, any attached code is executed.
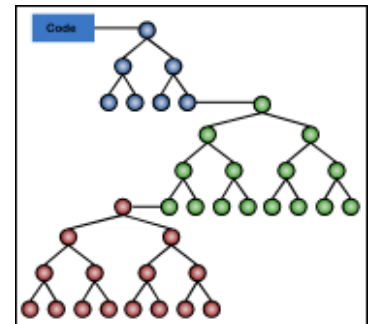
Solving each step probably requires multiple sub steps. For instance, extracting words out of letters or phonemes can be as simple as splitting at spaces, or at a longer silence, but could also make use of frames to express more complex structures. During these sub steps, a lot of splits will occur. For instance, a word is usually related to multiple objects (a word can have multiple meanings). A split should be created for each possibility so that the path can be checked if it is valid. Invalid splits should die out as soon as possible as not to tax the system to much.

To terminate invalid splits, multiple checks should be performed at the end of each step and or sub step. Checks can be: is the word known: is it stored in the dictionary, does it have relationships defined… Objects can be verified against other known data in the system, notably the 'Current' neuron. For instance, suppose the name 'Jan' is encountered. If the network has 2 objects attached

to the name 'Jan' (because it knows 2 individuals with this name), it could check if one of the 2 objects is somewhere attached to the 'Current'. If it is, this could result in a weight gain. If, on the other hand, both are attached to the 'current' neuron, a confusion needs to be solved first which can be done by activating a frame that asks this question and waits for the answer. This blocks the processor trying to solve the 'Jan' neuron, which blocks the entire split, effectively allowing multiple 'thoughts' to cross over time.

With this type of algorithm, the thesaurus data becomes very important. It is not possible to define frames for all situations and add all possible objects as triggers to the frame. Nor is it possible to create code for each 'action' word (verb). The accuracy of the thesaurus data and usage depth by the search algorithms greatly determine the 'intelligence' of the system, but also the required processing power.

Implementation

As a proof of concept and possible basis for future development, an initial implementation has been created in the form of a virtualized platform running on .net. A designer, N^2D, has also been created which runs on top of this platform.

Executing this type of application on top of traditional CPU's as used today, does pose some challenges though, notably in the usage of splits. This is because current CPU's are geared toward single (or very few) threaded applications, and not massively threaded ones like this type of network.

Future

Different paths can be taken to solve this threading problem, but I believe the best path is through hardware. This can be either through the further development of multi core processors up unto the point that they are able to provide enough cores to handle a plethora of splits. Another solution might be to truly elevate the network to a platform and design a new type of CPU better equipped to handle this kind of load.

At the Network design level, I believe the most promising results can be accomplished by implementing a natural selection algorithm. Imagine a system able to solve any type of question, the best it knows how, by trying out as many combinations as required, whith the entire human knowledge bank as it's seed.

Appendix 1: EBNF[5] for Compound type definition language

```
Definitions = {CompoundType}.
CompoundType = Name '=' TypeDeclare {LinkDeclare} '.'.
TypeDeclare=   ClusterDeclare
             | 'TextNeuron'
             | 'IntNeuron'
             | 'DoubleNeuron'
             | 'Neuron'
             | 'Any'
             .
```

---

[5] The Coco flavor of EBNF is used.

```
ClusterDeclare = '{' TypeDeclare {'|' TypeDeclare } '}'
                 '-' Meaning.
LinkDeclare= IncommingDeclare | OutGoingDeclare.
IncommingDeclare= '<-' Meaning '(' Name {'|' Name } ')'.
OutGoingDeclare='->' Meaning '(' Name {'|' Name } ')'.
Name= Identifier.
Meaning= Identifier.
Identifier = Letter {Letter | Number}.
Letter = ['a'..'z'] | ['A'..'Z'].
Number= ['0'..'9'].
```

A compound definition consists out of a name, the expected type of neuron and all the expected incoming and outgoing links. The name must be an identifier, so it must start with a letter, followed by 0 or more letters and/or numbers. You can use 'Any' to indicate that any type is allowed. Clusters are declared using brackets, with inside the brackets, all the allowed types for the children of the cluster, separated by a vertical line. After the brackets, the meaning of the cluster is declared, which must also be an identifier. Incoming and outgoing links are distinguished by the direction of the arrow. This is followed by the meaning and the name of another compound type that is expected on the other side of the link. If multiple types are allowed, a vertical line is used to separate them.