RESEARCH ARTICLE

In Search of Intelligence: Evolving a Developmental Neuron Capable of Learning

Gul Muhammad Khan\textsuperscript{a} and Julian Francis Miller\textsuperscript{b}

\textsuperscript{a}Department of Electrical Engineering, University of Engineering and Technology, Peshawar, Pakistan; \textsuperscript{b}Department of Electronics, University of York, Heslington, York, YO10 5DD, UK

\textit{(Received 00 Month 200x; final version received 00 Month 200x)}

A neuro-inspired multi-chromosomal genotype for a single developmental neuron capable of learning and developing memory is proposed. This genotype is evolved so that the phenotype which changes and develops during an agent’s lifetime (while problem solving) gives the agent the capacity for learning by experience. Seven important processes of signal processing and neural structure development are identified from biology and encoded using Cartesian Genetic Programming (CGP). These chromosomes represent the electrical and developmental aspects of dendrites, axonal branches, synapses and the neuron soma. The neural morphology that occurs by running these chromosomes is highly dynamic. The dendritic/axonal branches and synaptic connections form and change in response to situations encountered in the learning task. The approach has been evaluated in the context of maze-solving and the board game of checkers (draughts) demonstrating interesting learning capabilities. The motivation underlying this research is to, ab initio, evolve genotypes that build phenotypes with an ability to learn.

\textbf{Keywords:} Cartesian Genetic Programming, Computational Development, Artificial Neural Networks, Maze, Checkers.

1. Introduction

Learning and memory development in brains arises as a consequence of changing morphology and electrochemical activities in biological brain. However, the genome itself does not encode learned information, learning is acquired in the lifetime of an individual organism (i.e. it is phenotypic, not genotypic) and stored in the form of changes in neural structure and chemicals concentration Miller and Khan (2011). Kandel demonstrated the importance of small collections of neurons in learning and adaptation from experimental studies on snail aplysia Kandel et al. (2000). In the work described in this paper, we have used Cartesian Genetic Programming (CGP) Miller and Thomson (2000), Miller (2011) to evolve the genotype of an artificial developmental neuron. The genotype of the neuron is multi-chromosomal. It consists of seven chromosomes responsible for specific developmental and signaling functions in various neural components (soma, axon, dendrite), similar to biological neurons Zubler and Douglas (2009). The evolved chromosomes encode programs that \textit{when executed} gives rise to a neuron with developmental structure capable of solving mazes and playing checkers.

A number of methods have been introduced in past to develop artificial neural networks (ANNs) using genetic programming. However, we have gone further in...
that we have devised a functional model of a neuron with biological morphology. We have attempted to do this by devising an abstraction of real neurons that captures many of the important features. Various studies have shown that “dendritic trees enhance computational power” Koch and Segev (2000). Neurons communicate through synapses, that are not merely the point of connection between neurons Kandel et al. (2000). They can change the characteristics of the signal over various time scales. We take the view that the time dependent and environmentally sensitive variation of morphology of the real neurons is very important for learning and richer models are required that incorporate these features. In our model, a neuron consists of a soma, dendrites, axons with branches and dynamic synapses and synaptic communication. Neurite branches can grow, shrink, self-prune, or produce new branches. This allows it to arrive at a network whose structure and complexity is related to properties of the learning problem. Our model is much more complex than previous models because we take into account many more aspects that are discussed in neuroscience. Our model lies somewhere between the highly simplified models used in the field of artificial neural networks and the faithful replication of detailed neuroscience as employed in the “blue brain” project Markram (2006).

Our aim is to find a set of computational functions that encode neural structures with an ability to learn through experience as a result of development. Such neural structure would be very different from conventional ANN models as they are self-training and constantly adjust themselves over time in response to external environmental signals. In addition, they could grow new networks of connections when the problem domain required it. The main motivation for such a new model is to arrive at a neural developmental model that can produce a neural network capable of solving multiple problems (i.e. not suffer from a phenomenon called catastrophic interference Ans et al. (2002); French (1994); McCloskey and Cohen (1989); Ratcliff (1990); Sharkey and Sharkey (1995)). We think that this can be achieved by arriving at a neural model in which new neural sub-structures automatically come into existence when the network is exposed to new learning scenarios. Essential computational functions of neurons and the communication mechanisms amongst the neurons are identified from the neuroscience literature and are included in the proposed model. A well established and efficient form of Genetic Programming, namely Cartesian Genetic Programming is used to evolve the unknown processes inside neurons responsible for their individual and group behaviour.

In the work we present in this paper we are studying the learning capability of a single neuron in solving maze problems and in playing the game of checkers. There are several motivations for concentrating on a single neuron. Firstly, it allows us to assess the capability of a complex single neuron for problem solving. Secondly, it is much more feasible to assess a computationally expensive single neuron model than an entire network of such neurons. Thirdly it allows us to demonstrate that a single complex neuron can have impressive learning capability.

We have evaluated the learning ability of the proposed system on two distinct well-known and challenging learning problems: Maze navigation and the game of Checkers. A maze is a complex tour puzzle with a number of passages and obstacles (impenetrable barriers). It has a starting point and an end point. The job of the agent is to find a route from starting point to the end point. The agent starts with a limited energy that increases and decreases as a result of interaction with the paths and the obstacles in the maze environment. We have demonstrated that the one-neuron agent is able to solve the maze a number of times. The agents start navigation in the maze with a single neuron having a random structure. However, while it is navigating the maze the weights and branching structure of the neuron change.
In the second study, we have tested the capability of our single-neuron model in the game environment of checkers. We carried out a variety of different experiments for evolving checkers playing agents. In contrast to previous approaches our method has not used a board evaluation function, game tree, explicit learning rules or human domain knowledge. The learning abilities are encoded at a genetic level rather than at the phenotype level of neural connections. The agent can only see the pattern of board in the form of a real valued matrix. Thus the agent has to learn the pattern of the board, understand the game strategies and act accordingly.

In previous work, we have evaluated the effectiveness of this approach on a classic AI problem called wumpus world Khan et al. (2011). We found that the agents improved with experience and exhibited a range of intelligent behaviours. In this paper, our focus is on the learning capability of a single developmental neuron in the complex tour puzzle of maze environment Khan and Miller (2010b) and learning to play a game from its board patterns Khan and Miller (2010a).

Section 2 will describe the background and motivation behind the research of this paper. Section 3 will provide the overview of the model. Section 4 will give a detailed description of the proposed neural model and the information processing in the network. Section 5 will describe the application of the neuron model to solving maze and playing checkers. Finally in section 6 we will provide some concluding remarks and suggestions for future work.

2. Background and Motivation

This section will provide some background studies of the relevant literature and motivation behind this research.

2.1. Biology of Neuron

Neurons are the main cells responsible for information processing in the brain. They are different from other cells in the body not only in term of functionality, but also in biophysical structure Kandel et al. (2000). They have different shapes and structures depending on their location in the brain, but the basic structure of neurons is always the same. They have three main parts.

- Dendrites (Inputs): Receive information from other neurons and transfer it to the cell body. They have the form of a tree structure, with branches close to the cell body.
- Axons (Outputs): Transfer the information to other neurons by the propagation of a spike or action potential. Axons usually branch away from the cell body and make synapses (connections) onto the dendrites and cell bodies of other neurons.
- Cell body (Processing area or function): This is the main processing part of neuron. It receives all the information from dendrite branches connected to it in the form of electrical disturbances and converts it into action potentials, which are then transferred through the axon to other neurons. It also controls the development of neurons and branches.

2.2. Neural modeling

A number of techniques are used for simulation of neural development either in the form of construction algorithms or biologically-inspired growth processes. One approach aims to reproduce the morphological properties of real neurons. The method
described in this paper does not consider the actual biological processes responsible for neural growth and production of realistic dendritic structures that are used in an electrophysiology simulator Stiefel and Sejnowski (2007). Lindenmayer Systems have been used to model plant branching structures Lindenmayer (1968) and later have been successfully applied to develop neural morphologies Ascoli et al. (2001). A number of other methods such as probabilistic branching models Kliemann (1987), Markov models Samsonovich and Ascoli (2005) and Monte Carlo processes da Fontoura Costa and Coelho (2005) are also proposed as construction algorithm for neural development. Although these methods produce interesting neuronal shapes, they do not provide any insight into the fundamental growth mechanisms for neuronal growth. Growth models on the other hand provide the biological mechanisms responsible for generation of neuronal morphology. A number of interesting agent-based simulations have been reported that model various aspects of biological development, such as cell proliferation Ryder et al. (1999), polarization Samuels et al. (1999), neurite extension Kiddie et al. (2005), growth cone steering Krottje and van Ooyen (2007) synapse formation Stepanyants et al. (2008) and axon guidance and map formation de Gennes (2007).

Although these methods introduce various interesting techniques to model the neuronal growth which is the early stage of development of brain, they have not considered the signal processing aspects and its effect on the growth during interaction with the environment via sensory mechanisms. Also the models are tested once development is over, implying that there is no development during the lifetime of the agent. We introduce the method of evolving the functions that are responsible for neuronal growth, signaling and synapse formation during the lifetime of the agent.

2.3. Developmental systems

In biology, multi-cellular organisms are built through developmental process from ‘relatively simple’ gene structures. The same idea is used in computational development to produce complex systems from simpler systems capable of learning and self-adapting Stanley and Miikkulainen (2003).

Quartz and Sejnowski proposed a ‘manifesto’ for the importance of dynamic neural growth mechanisms in cognitive development Quartz and Sejnowski (1997). Parisi and Nolfi suggested that if neural networks are to be viewed in a biological context, they should be accompanied by genotypes as part of a population and inherited from parents to offspring Parisi (1997), Parisi and Nolfi (2001).

In the first practical attempts at evolving neural developmental programs, Parisi and Nolfi used a growing encoding scheme to evolve the architecture and the connection strengths of neural networks Nolfi et al. (1994), Nolfi and Parisi (1995). They used their network to control a small mobile robot (for a similar method see Husbands et al. (1994)). The network consists of a collection of artificial neurons distributed in 2D space with growing and branching axons. The genetic code inside them specifies the instructions for axonal growth and branching in neurons. Connections between neurons are made when an axon of a neuron reaches other neuron.

Cangelosi proposed a neural development model, that starts with a single cell undergoing a process of cell division and migration until a complete network in 2D is produced Cangelosi et al. (1994). Subsequently, neurons grow their axons to make connections with each other thus producing a complete neural network. The rules for cell division and migration are stored in genotype. Others introduced similar models Dalaert and Beer (1994); Gruau (1994).
Gruau introduced a genotype-phenotype mapping technique that allows the repetition of the phenotypic structure by re-using the same genetic information. In this case the terminal cells (nodes) point toward other trees. This encoding method can produce complex phenotypic networks from a compact genotype. Gruau called this method “automatic definition of neural sub networks (ADNS)” Gruau (1994).

Rust and Adams used a genetic algorithm to evolve parameters of a developmental system to grow artificial neurons with biologically-realistic morphologies Rust et al. (2000), Rust and Adams (1999). They showed that reasonably realistic neural morphologies could be produced. However, they did not apply their work to any substantial computational problem.

Jakobi created artificial genomic regulatory networks, where genes code proteins and proteins activate (or suppress) Jakobi (1995). He used proteins to define neurons with excitatory or inhibitory dendrites. The individual cell divides and moves due to protein interactions with an artificial genome, causing a complete multi-cellular network to develop. After differentiation, each cell grows dendrites following chemical sensitive growth cones to form connections between cells. This develops a complete recurrent ANN, which was used to control a simulated Khepera robot for obstacle avoidance and corridor following. During every generation, genotypes develop phenotypic structures, that are tested, with the best genotypes selected for breeding.

Federici presented an indirect encoding scheme for development of a neuro-controller, and compared it with a direct scheme Federici (2005). The adaptive rules used were based on the correlation between post-synaptic electric activity and the local concentration of synaptic activity and refractory chemicals. Federici used two steps to produce neuro-controllers:

- A growth program in a genotype to develop the whole multi-cellular network in the form of a phenotype. The growth program inside each cell is based on local variables and implemented by a simple recursive neural network without hidden layer (similar to our use of CGP).
- In a second step it translates all the cells into spiking neurons.

Each cell of a mature phenotype is a neuron of a spiking neuro-controller. The type and metabolic concentrations of a cell are used to specify the internal dynamics and synaptic properties of its corresponding neuron. The position of the cell within the organism is used to produce the topological properties of neuron: its connections to inputs, outputs and other neurons.

The network was implemented on the Khepera robot and its performance was tested with direct and indirect encoding schemes. The indirect method reached higher fitness faster than the direct one, but had difficulty in refining the final fitness value.

Roggen et al. devised a hardware cellular model of developmental spiking ANNs Roggen et al. (2007). Each cell can hold one of two types of fixed input weight neurons, excitatory or inhibitory each with one of five fixed possible connection arrangements to neighbouring neurons. In addition each neuron has a fixed weight external connection. The neuron integrates the weighted input signals and when it exceeds a certain membrane threshold it fires. This is followed by a short refractory period. They implemented a leakage phenomenon which decrements membrane potentials over time.

A number of researchers have studied the potential of Lindenmayer systems Lindenmayer (1968) for developing artificial neural networks and generative design. Boers and Kuiper have adapted L-systems to develop the architecture of ANNs (numbers of neurons and their connections) Boers and Kuiper (1992). They evolved
the rules of an L-system to generate feed-forward neural networks. They found that this method produced more modular neural networks that performed better than networks with a predefined structure. Hornby and Pollack evolved L-systems to construct complex robot morphologies and neural controllers Hornby and Pollack (2001), Hornby et al. (2003).

Stanley proposed a new way of evolving artificial neural networks, known as Neuro-Evolution of Augmenting Topologies (NEAT). His approach used three novel features: tracking genes with historical markings to allow easy crossover between different topologies, protecting innovation via speciation, and starting from a minimal structure and “complexifying” as the generations pass. NEAT was shown to perform faster than many other neuro-evolutionary techniques. Unlike a conventional neural network whose topology is defined by the user, NEAT allows the network topology to evolve. The NEAT approach begins with a simple structure, with no hidden neurons. It consists of a simplistic feed-forward network of input and output neurons, representing the input and output signals. As evolution progresses, the topology of the network is augmented by adding a neuron along an existing connection, or by adding a new connection between previously unconnected neurons Stanley and Miikkulainen (2002), Stanley and Miikkulainen (2004). However, using evolution as the mechanism for producing more complex networks is potentially very slow, since evolutionary operators use random processes and also this approach has no biological plausibility since natural evolution does not operate on aspects of the brain directly. Stanley has recently introduced a promising extension to the NEAT approach called HyperNEAT Stanley et al. (2009) that uses an evolved generative encoding called a Compositional Pattern Producing Network (CPPN) Stanley (2007). The CPPN takes coordinates of pairs of neurons and outputs a number which is interpreted as the weight of that connection. With this approach the ANNs can be evolved with complex patterns where collections of neurons have similar behaviour depending on their spatial location. It also means that one evolved function (the CPPN) can determine the strengths of connections of many neurons.

Miguel et al. used Continuous Time Recurrent Neural Network (CTRNN) derived from dynamical neuron models known as leaky integrators. CTRNNs are an intermediate approach between sigmoidal and spiking neuron. They utilized important characteristics such as spiking neuron-like input integration over time and variable internal state. The latter can also dynamically change state in the absence of external inputs. CTRNNs were evolved using NEAT and applied on the pole balancing problem producing better results than the standard NEAT algorithm Miguel et al. (2008).

Downing chose a higher abstraction level in his neural developmental system, this was primarily to avoid the complexities of axonal and dendritic growth. Though he retained key aspects of cell signaling, competition and cooperation of neural topologies in nature Downing (2007). He tested his approach on a simple movement control problem known as starfish. The task for the k-limbed animate was to move away from its starting point in a given time.

The genotype-phenotype mapping in developmental approaches is highly indirect and non-linear. Genes act like instructions and development is the process of executing those instructions and dealing with the highly parallel interactions between them and the structure they create Kumar and Bentley (2003). As in biology, the genome has no direct interaction with the environment, it is the developing phenotype which it responds to and through that process, is shaped by interaction with the external environment.

Artificial neural networks are intended to mimic, in some sense, the computa-
tional models of nervous systems. However many ANN models ignore the fact that the neurons in the nervous system are part of the phenotype which is derived from the genotype through a process called development. The information specified in the genotype determines the rules that develop the nervous system based on environmental interaction during developmental phase.

One of the major difficulties in abstracting neuroscience for computational models is that one can lose the essential ingredients required to make a powerful learning system. We have found the evidence from neuroscience, on the importance of time-dependent morphological processes in learning, so compelling, that we have included it in a model of an artificial developmental neuron Miller and Khan (2011). The detailed motivations are described in more details in the later sections.

2.4. Motivation behind CGP Developmental Neural Network

ANN models proposed to date, although inspired by biological nervous system, adopt only a very few aspects of biological brain. Here, we have extended this view and identified a number of other important features that could be added to individual neuron structure. In the brain, these features prove to be extremely important for learning and memory. Synaptic weights are responsible only for extremely short term memory Kleim et al. (1998), long term memory results in modified structure of the neuron Terje (2003).

Originally there were good reasons for ANNs Gurney (1997) to ignore many aspects of biological neural systems. Simple models were required that could be executed on relatively slow computers. However, the computational power of modern computers has made more complex neuro-inspired approaches much more feasible. At the same time, our understanding of neuroscience has increased considerably. Important neglected aspects include the role of neuron morphology, developmental processes, neuron structure and mechanisms of communication between neurons. ANNs consider the brain as a connectionist system in which each neuron is a node containing signal processing functions. However, real neurons do complex processing through neurite branches before the signal even reaches the soma and there are an enormous variety of neurite morphologies in different types of nervous systems. Biological neurons are located in real space and transfer signals to their neighbours through electrochemical synapses Kandel et al. (2000). These synaptic connections are not fixed but change over the course of time. Sub-processes of neurons are highly time-dependent and many structures are in a constant state of being rebuilt and changed Smythies (2002). In addition, in the brain, memory is not a static process and the location and mechanisms responsible for remembered information is in constant (though, largely gradual) change. The act of remembering is a process of reconstructing and changing the original structure that was associated with the original event Rose (2003). The physical topology of the neural structures in the brain is constantly changing and is an integral part of its learning capability Kandel et al. (2000) (pages 67-70).

One of the difficulties in attempting to create a dynamic computational model inspired by neuroscience is that, the internal dynamics of biological neurons are extremely complicated and many of these processes may be unnecessary in a machine learning technique. However, we take the view that the biology of neurons (i.e. their gross morphology and connectivity) is sufficiently well understood Alberts et al. (2002), Shepherd (1990) to allow us to identify essential sub-systems that we must attempt to evolve in order to achieve a computational equivalent. Conventional models of neural networks do not consider the genetics of neurons and the development of a network during learning. How might we obtain programs
that represent the computational processes of neurons? We argue that Genetic Programming (GP) offers at least in principle, the capability of representing neural programs and the transfer of genetic changes from generation to generation. We have used a particular form of GP called Cartesian Genetic Programming Miller and Thomson (2000) to construct our computational model. We call our model the CGP developmental network (CGPDN).

We represent a neuron by seven neural programs that are encoded as CGP chromosomes Khan et al. (2007). Each chromosome encodes a particular aspect of real neurons. We have provided each neuron with a structural morphology such that it consists of a soma, dendrites Panchev et al. (2002), axons with branches and dynamic synapses Graham (2002) capable of synaptic communication. The neuron exists in a two dimensional toroidal grid to give branches a sense of spatial proximity. Branches are allowed to grow and shrink, and communication between axon branches and dendritic branches is allowed. Also dendrites, and axon branches can grow, die and change while solving a computational problem. The synaptic morphology can change at runtime and this affects the information processing. While this model is undeniably quite complex and involves many variables and parameters it more faithfully represents the dynamic capabilities of the brain while at the same time is still an immense simplification of it.

Artificial neurons in traditional ANNs use a simple non-linear function such as sigmoid and hyperbolic tangent. Such a simple neuron allows networks of such neurons to be mathematically tractable. However, like all biological cells, actual neurons are immensely complex, thus it remains unclear how the complexity of an artificial neuron relates to the computational properties if a network made of such neurons. The CGPDN neuron is inspired by the morphological structure of a biological neuron, and has a number of compartmental functions at dendrite branches, soma and axon branches. These internal processing compartments of the CGPDN neuron not only process the signal at various levels, but also help in growth of neuron structure that continue to change in response to the external stimuli. The single biological neuron has a lot of complexities yet to be explored, this work is focused on extracting the learning abilities of a single neuron and identifies the necessary functions responsible for signal processing and growth rules of the biological neurons. Once identified these functions can be replicated in larger network with more neurons for complex learning scenarios.

3. The CGP Developmental Network (CGPDN)

This section describes in detail the structure of the CGPDN, along with the rules and evolutionary strategy used to run the system. The CGPDN has two main aspects:

a) A neuron with a number of dendrites, with each dendrite having a number of branches and an axon having a number of axon branches.

b) A genotype representing the genetic code of the neuron, having seven chromosomes.

The first aspect (a) defines mainly the neural components and their properties and the second (b) is concerned with the internal behaviour of the neuron. These chromosomes represent the functionality of various parts of the neuron. During evolution the second aspect (genotype) is evolved toward the best functionality, whereas the first aspect (the neural components and their properties) only changes during the lifetime of the neuron, i.e while it is performing the learning task.

The CGPDN neuron is placed at a random location in a two dimensional grid (the CGPDN grid) and is only aware of dendritic and axonal branches in close proximity
Figure 1. On the top left a grid is shown containing a single neuron. The rest of the figure is an exploded view of the neuron. The neuron consists of seven evolved computational functions. Three are electrical and process a simulated potential in the dendrite (D), soma (S) and axo-synapse branch (AS). Three more are developmental in nature and are responsible for the life-cycle of neural components (shown in grey). They decide whether dendrite branches (DBL), soma (SL) and axo-synaptic branches (ASL) should die, change or replicate. The remaining evolved computational function (WP) adjusts synaptic and dendritic weights and is used to decide the transfer of potential from a firing neuron to a neighbouring neuron.

A neuron is initially allocated a random number of dendrites, branches, one axon and a random number of axon branches. It receives information through dendrite branches and transfers information through axon branches to neighbouring motor nerves (which are virtual dendrite branches, i.e. outputs of the system). The dynamics of the neuron changes during this process, branches may grow or shrink and move from one CGPDN grid point to another. They can produce new branches and can disappear. Axon branches transfer information only to dendrite branches in their proximity. This process is performed by passing the signals from all the neighbouring branches through a CGP program, acting as an electro-chemical synapse, which updates the values of signal strength (potential) only in neighbouring branches. An integer variable that mimics electrical potential is used for internal processing of neuron and communication with sensory and motor nerves. External inputs and outputs are also converted into potentials before being applied to the network.

Developmental programs determine the morphology of the neural network. The number of dendrites on a neuron is fixed, however the number of dendrite branches on each dendrite is variable. It is determined by whether the developmental dendrite branch programs (DBL) in the past decided to grow new branches. The neuron is invested with a single axon. However, the number of axo-synapses attached to each axon is determined by whether axosynaptic branch program (ASL) in the past decided to grow new axo-synapses. The number of programs that are run in the developing neural network may vary, however the size of the genotype stay
fixed. This is one of the advantages of the developmental approach we have taken. A relatively simple collection of evolved programs could in principle define an entire network of arbitrary complexity. However, of course, after the network is developed it may have neuron with complex dendrite, and axosynaptic branch structures. This will means that it becomes slower to run as all these programs are executed sequentially on the processor we run our experiments on.

In the next four subsections we will describe the basic parameters of CGPDN (Resistance, Health, Weight and State-factor), the Cartesian genetic program (used as genotype), the evolutionary strategy and the way inputs and outputs are applied to the network.

3.1. **Health, Resistance, Weight and State-factor**

Four variables are used to represent the fundamental properties of the CGP developmental neuron (health, resistance, weight) or as an aid to its computational efficiency (state-factor). Each dendrite and axo-synaptic branch has three variables: health, resistance and weight. The values of these variables are adjusted by the CGP programs (see below). The health variable is used to govern replication and/or death of dendrites and axon branches. The resistance variable controls growth and/or shrinkage of dendrites and axon branches (see section life cycle for details). Synaptic plasticity is incorporated in the CGPDN by introducing three types of weights: 1) dendrite branch, 2) soma and 3) axon branch Debanne et al. (2003); Frey and Morris (1997); Gaiarsa et al. (2002). These weights can be adjusted by genetic processes during development of the network. Changes in the dendrite branch weight are analogous to the amplifications of a signal along the dendrite branch (see London and Husser (2005)), whereas changes in the axon branch (or axo-synaptic) weight are analogous to changes at the pre-synaptic level and post synaptic level (at synapse). Inclusion of a soma weight is justified by the observation that a fixed stimulus generates different responses in different neurons Traub (1977). Each soma has only two variables: health and weight. The use of these variables is summarized in figure 7. Health, weight and resistance are represented as integers.

A state-factor is used as a parameter to reduce computational burden by keeping some of the branches inactive for a number of cycles. When the state-factor is zero, branches are considered to be active and their corresponding program is run. The value of the state-factor is affected by CGP programs, as it is dependent on the outputs of the CGP electrical processing chromosomes (see later).

3.2. **Cartesian Genetic Program (Chromosome)**

CGP is a well established and effective form of Genetic Programming. It represents programs by directed (usually acyclic) graphs Miller (2011); Miller and Thomson (2000); Walker and Miller (2008). The genotype is a fixed length list of integers, which encode the function of nodes and the connections of a directed graph. Nodes can take their inputs from either the output of any previous node or from a program input (terminal). The phenotype is obtained by following the connected nodes from the program outputs to the inputs.

In CGP an evolutionary strategy of the form $1 + \lambda$, with $\lambda$ set to 4 is often used Miller et al. (2000). The parent, or elite, is preserved unaltered, whilst the offspring are generated by mutation of the parent. If two or more chromosomes achieve the highest fitness then newest (genetically) is always chosen. We have used this algorithm in the work we report here.
Each genotype in CGPDN is a set of seven CGP chromosomes. Walker et al. investigated the utility of evolving genotypes consisting of a number of chromosomes. Walker et al. (2006), Walker and Miller (2008), they found that multi-chromosomal evolution could evolve solutions to a variety of digital circuit problems much faster than a single chromosome approach.

The CGP function nodes used here are variants of binary if-statements known as 2 to 1 multiplexers Miller et al. (2000) with three inputs each, as shown in figure 2:

Here $a$, $b$ and $c$ are the inputs to the node (as shown in Figure 2). These functions are Boolean operations representing logical AND (represented by ‘.’) and logical OR (represented by ‘+’). The multiplexers require four genes each to describe which type of multiplexer (underlined in Fig. 3) and its connections. All multiplexers operate in a bitwise fashion on 32-bit data. Multiplexers can be considered as atomic in nature as they can be used to represent any logic function Chen and Hurst (1982); Miller et al. (2000).
Figure 3. Structure of CGP chromosome. Showing a genotype for a 4 input, 3 output function and its decoded phenotype. Inputs and outputs can be either simple integers or an array of integers. Nodes and genes in grey are unused and small open circles on inputs indicate inversion. The function type in genotype is indicated by underline (underneath the integer showing function of multiplexer). All the inputs and outputs of multiplexers are labeled. Labels on the inputs of the multiplexer shows where are they connected (i.e. they are addresses). Input to CGP program is applied through the input lines as shown in figure. The number of inputs (four in this case) and outputs (three in this case) to the CGP program is defined by the user, which is different from the number of inputs per node (three in this case i.e. a, b and c.)

Figure 3 shows the genotype, the corresponding phenotype obtained and the inputs and outputs to the CGP. Output is taken from the nodes (6, 8, 4) as specified in the genotype. Inputs are applied to CGP chromosomes in two ways:

- Scalar
- Vector

In the former, the inputs and outputs are integers while in the latter, inputs required by the chromosome are arranged in the form of an array, which is then divided into ten CGP input vectors. If the total number of inputs can’t be divided into ten equal parts, then they are padded with zeros. This allows us to process an arbitrary number of inputs by the CGP chromosome simply by iterating through the elements of the vectors. In general CGP can’t take a variable number of inputs. We devised this method to allow it to take a variable number of inputs at run time. As the inputs are arranged in the form of vectors, and each vector can have arbitrary number of elements. This method adds some noise which is more pronounced when the number of inputs is less than ten.

3.3. Evolutionary Strategy

The evolutionary strategy utilized is of the form $1 + \lambda$, with $\lambda$ set to 4 Yu and Miller (2001), i.e. one parent with 4 offspring (population size 5). The parent or elite, is preserved unaltered, whilst the offspring are generated by mutation of the parent. The best chromosome is always promoted to the next generation, however, if two
or more chromosomes achieve the same highest fitness then newest (genetically) is always chosen Miller et al. (2000). This step is extremely important and utilizes neutral search Vassilev and Miller (2000); Yu and Miller (2001).

The steps in the evolutionary cycle are as follows:

- Create a random population of five genotypes (Each genotype consists of seven chromosomes)
- Create a CGPDN with random number of dendrites and branch structures.
- An evolutionary generation consists of:
  - For each genotype \( C \) in the population:
    - Produce a copy of the random CGPDN
    - Run the genotype on the CGPDN.
    - Calculate fitness of the resulting CGPDN, \( F(C) \)
  - From population select best \( F(C) \); if two or more are equally best then pick newest of them Miller et al. (2000)
  - Create new population by mutation, keeping the promoted genotype unchanged
  - Continue until either a maximum number of generations or a solution is found

Thus, although in the work we report here uses a single neuron, in principle a mature network could be produced by executing the program encoded in the genotype starting from the same initial random network of neurons with dendrites and dendrite and axon branches.

The promoted genotype is mutated to produce four new genotypes (the offspring) in the following way:

- Calculate the number of genes to be mutated i.e.
  \[ N_{bit} = N_g \times \mu / 100; \text{ and} \]
  \[ N_g = (N_i + 1) \times N_n \times N_{ch} \]
  Where,
  - \( N_g = \) Number of Genes
  - \( \mu = \) Mutation Rate
  - \( N_i = \) Number of Inputs per Node (3 in this case)
  - \( N_n = \) Number of Nodes per Chromosome
  - \( N_{ch} = \) Number of Chromosomes (7 in this case)
  - \( N_{bit} = \) Number of Bits to be mutated
- Select pseudo-randomly one gene at a time and mutate it pseudo randomly.
  Mutation of a gene means:
  - if it is a connection
    - replace with another valid connection.
  - if it is a function
    - replace with another valid function.

3.4. Inputs and Outputs

The inputs are applied to the CGPDN through axon branches from other fixed neurons in the environment by using the axo-synaptic electrical processing chromosomes. These branches, which represent sensory nerves, are distributed in the network in a similar way to the axon branches of CGP Developmental Neuron (as shown in figure 1). They take the input from the environment and transfer it directly to input axo-synapse. When inputs are applied to the system, the program encoded in the axo-synaptic electrical branch chromosome is executed, and the resulting signal is transferred to its neighbouring active dendrite branches.

Similarly, we have output (motor) nerves that receive the signal from the network
Figure 4. Electrical processing in neuron at different stages, from left to right branch potentials are processed by dendrites (D), then averaged. These average dendrite potentials are averaged at the soma, which adjusts the potential using the program encoded in the soma electrical chromosome (S), giving a final soma potential. This is fed in to a comparator which decides whether to fire an action potential. This is transferred using the program encoded in the axo-synapse electrical chromosome (AS) (transferring it using axo-synapses) through the dendrite branches representing the output of the system. These output dendrite branches are distributed in the CGPDN grid as shown in Figure 1. The signal strength on these branches are updated by the axo-synaptic chromosomes of neurons in the same way as other dendrite branches. These outputs are read without further processing after every five cycles. The number of inputs and outputs can change at run time (during development), a new branch for input or output can be introduced into the network or an existing branch can be removed. This allows CGPDN to handle arbitrary number of inputs and outputs at runtime.

In the next section we will describe the complete neuron model along with its sub-processes.

4. CGP Model of Neuron

In this model, neural functionality is divided into three major categories.

- Electrical Processing
- Life Cycle
- Weight Processing

These categories are explained in detail one by one in the subsections below.

4.1. Electrical Processing

The electrical processing part is responsible for signal processing inside neuron and communication with motor and sensory nerves (input and output branches). It consists of the following three chromosomes (as shown in Figure 1):

- Electrical Processing in dendrite
- Electrical Processing in soma, and
- Electrical Processing in axo-synaptic branch
The way they process electrical signal and transfer to output branches is shown in Figure 4.

4.1.1. Electrical Processing in Dendrite

This is a vector processing chromosome. This chromosome handles the interaction between potentials of different dendrite branches belonging to the same dendrite. Figure 1 shows the inputs and outputs. The input consists of the potentials of all the active branches connected to the dendrite plus the soma potential. The CGP program produces the new values of the dendrite branch potentials as output. Figure 1 shows the inputs and outputs to the Dendrite Electrical CGP (DECGP). Subsequently, the potential of each branch is processed by adding weighted values of Resistance, Health and Weight using the following equation.

\[ P = \dot{P} + \alpha_D H + \beta_D W - \gamma_D R \]  

(1)

Where \( P \) and \( \dot{P} \) are the potential and updated potential respectively. \( H, W \) and \( R \) are the adjustment parameters having values between 0 and 1. In this case they are 0.02, 0.05 and 0.05 respectively. The above equation shows that as the health and weight of the branch increases so does its potential and as the resistance increases, the potential goes down (simulating usual resistive behaviour). We allow increases in health to cause an increase in potential because it is reasonable to assume that healthy branches facilitate the flow of potential. Weights are responsible for amplification of potential. Thus high values of weights should cause an increase in potential.

The State-factor of branches is adjusted based on the updated value of branch potential. The branch is made more active (by reducing its state-factor) if there is an increase in potential after running the dendrite program (D). This is done to encourage more sensitive branches to participate in processing by keeping them active. We set up a range of thresholds for assigning the state-factor. If any of the branches are active (has its state-factor equal to zero), its life cycle CGP program (explained later) is run. The same process is repeated for all the dendrites and their corresponding branches. After processing all the dendrites, the average value of potentials of all the dendrites is taken, which in turn is the average value of the potentials at the active dendrite branches attached to them. This potential and the soma potential are applied as inputs to the CGP soma electrical processing chromosome as described below.

4.1.2. Electrical Processing in Soma

This is a scalar processing chromosome. This chromosome is responsible for determining the final value of soma potential after receiving signals from all the dendrites as shown in Figure 1. The chromosome produces an updated value of the soma potential (\( \dot{P} \)) as output, which is further processed with a weighted summation of Health (\( H \)) and Weight (\( W \)) of the soma using the following equation.

\[ P = \dot{P} + \alpha_S H + \beta_S W \]  

(2)

Where \( \alpha_S \) and \( \beta_S \) have been assigned the values 0.02 and 0.05.

The processed potential of the soma is then compared with the threshold potential of the soma and a decision is made whether to fire an action potential or not. If
the soma fires, it is kept inactive (refractory period) for a few cycles by increasing its *state-factor*.

After this, the soma life cycle chromosome is run, and the firing potential is sent to the output branches by running the axo-synapse electrical processing chromosome (as described in the next subsection). The threshold potential of soma is also adjusted to a new value (maximum) if the soma fires.

### 4.1.3. Electrical Processing in Axo-Synaptic Branch

This is a vector processing chromosome. The potential from the soma is transferred to the output branches through axon branches. Both the axon branch and the synapse are considered as a single entity with combined properties. Axo-synapses transfer the signal only to the neighbouring output branches as shown in figure 5. Branches sharing the same grid square form a neighbourhood. Figure 1 shows the inputs and outputs to the chromosome responsible for the electrical processing in each axo-synaptic branch.

The chromosome produces the updated values of the neighbouring dendrite branch potentials and the axo-synaptic potential as output. The axo-synaptic potential is then processed as a weighted summation of *Health*, *Weight* and *Resistance* of the axon branch using the following equation.

\[
P = \hat{P} + \alpha_{AS}H + \beta_{AS}W - \gamma_{AS}R
\]  

(3)

Where \( P \) and \( \hat{P} \) are the potential and updated potential respectively. H, W and R are the health, weight and resistance of axon branch respectively. \( \alpha_{AS} \), \( \beta_{AS} \) and \( \gamma_{AS} \) are the adjustment parameters having values between 0 and 1, in this case they are 0.02, 0.05 and 0.05 respectively.

The axo-synaptic branch weight processing program (see figure 6) is run after the above process and the processed axo-synaptic potential is assigned to the dendrite branch having the highest updated *Weight*. The *state-factor* of branches is adjusted based on the updated value of branch potential. The branch is made more active if the potential increases after the execution of the program encoded in the axo-synaptic electrical chromosome (AS). We set up a range of thresholds for assigning the *state-factor*. If any of the branches are active (has its *state-factor* equal to zero), its life cycle CGP program is run, otherwise it continues processing the other axon branches. The axo-synaptic branch CGP is run in all the active axon branches one by one.
4.2. **Weight Processing**

This is a vector processing chromosome. Weight processing is responsible for updating the *weights* of branches. It consists of one chromosome. The *weights* of axon and dendrite branches affect their capability to modulate and transfer the information (potential) efficiently. Weights affect almost all the neural processes either by virtue of being an input to a chromosomal program or as a factor in post processing of signals.

Figure 6 shows the inputs and the outputs to the weight processing chromosome. The CGP program encoded in this chromosome takes as input the *weight* of the axo-synapse and the *neighbouring* (same CGPDN grid square) dendrite branches and produces their updated values as output. The synaptic potential produced at the axo-synapse is transferred to the dendrite branch having the highest weight after weight processing.

4.3. **Life Cycle of Neuron**

This part is responsible for increase or decrease in the number of neurons and in the number, growth and properties of neurite branches. It consists of three chromosomes:

- Life Cycle of dendrite branch
- Life Cycle of soma (disallowed if only one neuron used)
- Life Cycle of axo-synapse branch

4.3.1. **Life Cycle of Dendrite Branch**

This is a scalar processing chromosome. Figure 7 shows the inputs and outputs of the chromosome. This process updates *resistance* and *health* of the branch. Variation in *resistance* of a dendrite branches is used to decide whether it will grow, shrink or stay at its current location. If the variation in *resistance* during the process is above certain threshold (*R_DB*), the branch is allowed to migrate to a different neighbouring location at random. The neighbouring location can be one of the eight possible squares around a rectangular grid. The branch can move to either of the eight neighbouring squares at random. It can move only one square away at a time. Changes in resistance can be negative (shrinkage) or positive (growth). In both cases, the absolute change in resistance is used to decide if the branch should
move from its current grid square to another grid square. Growth and shrinkage does not occur within one grid square, since a square is the smallest unit.

The updated value of dendrite branch health decides whether it produces offspring, dies or remains as it was with an updated health value. If the updated health is above certain threshold \((H_{\text{db}_{\text{max}}})\), it is allowed to produce offspring and if below certain threshold \((H_{\text{db}_{\text{min}}})\), it is removed from the dendrite. Producing offspring results in a new branch at the same CGPDN grid point connected to the same dendrite.

The values of \((R_{\text{DB}})\), \((H_{\text{db}_{\text{max}}})\) and \((H_{\text{db}_{\text{min}}})\) are specified by the user.

4.3.2. Life Cycle of Soma

This is a scalar processing chromosome. Figure 7 shows inputs and outputs of the soma life cycle chromosome. This chromosome is intended to evaluate the life cycle of a neuron. This chromosome produces updated values of health and weight of the soma as output. These updated values affect the firing decision of a neuron.

4.3.3. Axo-synaptic Branch Life Cycle

This is a scalar processing chromosome. The role of this chromosome is similar to dendrite branch life cycle chromosome. Figure 7 shows the inputs and outputs of axo-synaptic branch life cycle chromosome. It takes health and resistance of the axon branch as input and produces the corresponding updated values as output. The updated values of resistance are used to decide whether the axon branch should grow, shrink or stay at its current location. Like the dendrite branches, if the variation in axon resistance is above a certain threshold \((R_{\text{AS}})\), it is allowed to migrate to a different neighbouring location at random. The health of the axon branch decides whether the branch will die, produce offspring or merely continue with an updated value of health. If the updated health is above certain threshold \((H_{\text{as}_{\text{max}}})\), it is allowed to produce offspring and if below certain threshold \((H_{\text{as}_{\text{min}}})\), it is removed from the axon. Producing offspring results in a new branch at the same CGPDN grid point connected to the same axon.

Detailed information processing mechanism in the entire network is given in the appendix.
5. Application of CGPDN

We have evaluated the learning capabilities of the CGPDN on two well-known problems: maze navigation and checkers. The following two subsections will provide the experimental details, the experimental results and analysis.

5.1. The Maze: A complex tour Puzzle

A maze is a term used for complex and confusing series of pathways. It is an important task domain for autonomous robot navigation and route optimization Blynel and Floreano (2003); Tani (1996). The idea is to teach an agent to navigate through an unknown environment and find the optimal route without having prior knowledge. A simplified version of this enigma can be simulated by using a random two-dimensional maze. The pathways and obstacles in a maze are fixed.

5.2. Experimental Setup

In the experiments, an agent is provided with a single CGP Developmental Neuron as its signal processing “network”. The job of the agent is to find routes from a starting point toward an end point of a maze as many times as it can in a single life cycle. We have used a 2D maze representation for this experiment as shown in figure 8. The 2D Maze representation is explored in a number of scenarios Ilin et al. (2007); Pang and Werbos (1996). We have represented the maze as a rectangular array of squares with obstacles and pathways (as shown in the figure 8). A square containing an obstacle cannot be occupied. Movement is possible up or down on squares on the outside columns. Movement is either left or right on rows, unless there is a pathway, in which case downward motion is possible. This is inspired by the clustering approach used to improve learning capabilities of an agent Mannor et al. (2004). We used different sizes of mazes to test the ability of the agent. The location of the obstacles, pathways and exit are chosen randomly for different experimental scenarios.
5.2.1. Energy of Agent

The agent is assigned a quantity called energy, which has an initial value of 50 units. If an agent attempts to penetrate an obstacle its energy level is reduced by 5 units. If it encounters a pathway and moves to a row closer to the exit, its energy level is increased by ten units. If it moves a row further away from the maze exit, its energy is reduced by ten units. This is done to enhance the learning capability of agent by giving it a reward signal. If the agent reaches the exit, its energy level is increased by 50 units and it is placed back at the starting point and allowed to solve the maze again. Finally, if the agent arrives home, without having reached the exit, the agent is terminated. For each single move, the agent’s energy level is reduced by 1 unit, so if the agent just oscillates in the environment and does not move around and acquire energy through solving tasks, it will run out of energy and die.

5.2.2. Fitness Calculation

The fitness value, which is used in the evolutionary scheme, is accumulated while the agent’s energy is greater than zero as follows:

• For each move, increase fitness by one. This is done, to encourage the agents ’brain’ to remain active and not die.
• Each time the agent reaches the exit, its fitness is increased by 100 units.

5.2.3. Inputs to neuron

The maximum allowed neural potential is $M = 2^{32} - 1$. The agent’s axo-synapses can have the values of potential, $I$, depending on the circumstances in the following way. Note that the agent can sense only one signal on a maze square, even if there are more than one.

• $I = 0$ default.
• $I = M/60$ finds a pathway to a row closer to exit.
• $I = M/120$ tries to land on obstacle.
• $I = M/200$ on exit square.
• $I = M/100$ adjoining square north of an obstacle.
• $I = M/110$ adjoining square east of an obstacle.
• $I = M/130$ adjoining square south of an obstacle.
• $I = M/140$ adjoining square west of an obstacle.
• $I = M/180$ approaches exit from north direction
• $I = M/190$ approaches exit from east direction
• $I = M/210$ approaches exit from south direction
• $I = M/220$ approaches exit from west direction
• $I = M/255$ home square (starting point)

The agent’s axo-synapse can have signals only from its occupied square, it cannot sense signal from the neighbouring squares, this makes the task more challenging. This is done to avoid evolution finding a classifier network rather than a network that builds a memory based on its experience.

5.2.4. Agent movement and termination

When the experiment starts, the agent takes its input from the starting point (on the top left corner as shown in figure 8). This input is applied to the developmental network (CGP Neuron) of the agent using input axo-synapses. The network is then run for five cycles (one step). During this process it updates the potentials of the output dendrite branches. After the step is complete the updated potentials of all output dendrite branches are noted and averaged. The value of this average
potential decides the direction of movement for the agent. If there is more than one
direction the potential is divided into as many ranges as the possible movements.
For instance if two possible directions of movement exist, then it will take one
direction if the potential is less than \( \frac{M}{2} \) and the other if greater. The same
process is then repeated for the next maze square. The agent is terminated if
either its energy level becomes zero or when it returns home.

5.2.5. **CGP Neuron Setup**

The various parameters of CGP neuron are chosen as follows:

- The neuron branches are confined to 3x3 CGPDN neural grid.
- Inputs and outputs to the network are located at five different random squares.
- The maximum number of dendrites is 5. The maximum branch state-factor is 7.
- The maximum soma state-factor is 3.
- The mutation rate is 5%.
- The maximum number of nodes per chromosome is 100.
- Maximum number of dendrite and axon branches are hundred and twenty, re-
spectively.

These parameters have not been optimized and have largely been chosen as they
work reasonably well and do not incur a prohibitive computational cost.

5.2.6. **Difficulty of the problem**

It is important to appreciate the difficulty of the task. The agents starts with a
single neuron with random connections. Evolution must find a series of programs
that build a computational neural structure that is stable (not losing all branches
etc.). Secondly, it must find a way of processing infrequent environmental signals
(pathway, obstacles, exit, home etc) and understand their meaning (beneficial and
deleterious). Thirdly, it must navigate in this environment using some form of mem-
ory. It must also confer goal-driven behaviour on the agent. The agent performance
is determined by its capability to solve the maze as many times as it can during a
single life cycle.

The maze environment we produced is much more complex than the traditional
mazes, as the agent in this environment can only sense the signal from the maze
square it is occupying, not from neighbouring squares. So in order to solve the
maze the agent must develop a memory of each step it makes and the direction of
movement, and use this memory to find a route toward the exit. As the structure
and weights of branches change while solving the maze, the learned information
is stored both in weights and the structure of the neuron. The capability to learn
and the mechanism whereby learned information is transformed into memory in
the form of updates in weights and structure is encoded in genotype.

5.2.7. **Results and Analysis**

Figure 9 shows a number of mazes in first column. Fitness improvement during
evolution is shown in the second column. The third column in figure 9 shows the
energy variation of the best maze solving agent. The small continuous drop in
energy is due to an agent losing its energy after every step. Large decreases occur
through encounters with an obstacle or going away from the exit by following
the pathway in opposite direction. Small increases shows the result of following
the pathway and moving toward the exit and large increases happen when the
agent finds the exit. The fourth and the last column shows the variation in neuron
branching structure over the agent lifetime, while it is solving the maze.

The agent is able to solve the maze four to five times during a single life cycle
in all the cases as shown in the second column of figure 9. During this process the
structure of the neuron also changes in terms of the number of dendrite and axon branches. The fourth column of the figure 9 shows that although agents start with a minimal structure they soon achieve a structure that is most advantageous.

In traditional methods that train an agent to solve the maze and find a path, the network characteristics are fixed once it is trained to solve the maze. So if they are allowed to start the maze again they would always follow the same path. The CGP neuron continues to change its architecture and parameter values as it also continues to explore different paths on future runs. This makes it possible for it to obtain (or forget!) a global optimum route. The network is not trained to stabilize on a fixed structure, that it does so, seems to be because it has found a suitable structure for the desired task. The best architecture does not necessarily have the most neurite branches. This is evident from the varied characteristics in the last column of figure 9.

It is interesting to note that as the maze becomes larger the structure of the neuron grows in response. This is evident from the last column of the figure 9. For an 8x8 maze (first and second maze) the agent structure grows and stabilizes on a fairly small structure whereas for a 10x10 maze (3rd, 4th and 5th mazes) the number of dendrite and axon branches grows into a fairly large structure (the maximum allowed value of 100 in this case). Further investigation reveals that as the route toward the exit becomes more and more complex, the network structure become richer in terms of branches. This is evident from the second 10x10 maze (4th row) where the number of blocking paths are 10 (with each obstacle providing four walls in all the four directions, 40 walls), and number of pathways are 20 (ten on the sides (first and last column) with possibility to move in both upward and downward directions and ten that are only open toward the exit in downward
direction). In this case the agent was able to solve the maze three times, as is evident from the rises in the energy level diagram. However, it dies on the fourth run when it tried to escape through the starting point. In the next case, when we have reduced the number of obstacles to six (24 walls) while keeping the number of pathways the same as shown in the in fourth row of figure 9. This time the agent was able to solve the maze four times and its axon branch structure improves during the run but the dendrite structure stabilizes on a low value. The final maze is a variant of 10x10 maze in third row with similar characteristics. In 8x8 mazes when the environment is simple, the agent was able to solve the maze a number of times even though it stabilized on a fairly small branch structure. This strongly suggests that the complexity of the CGP Developmental Neuron structure increases with increases in the complexity of the task environment.

5.3. The Game of Checkers

In second case, we have evaluated CGPDN for its learning abilities in the game environment of checkers. Throughout the history of AI research, building computer programs that play games has been considered a worthwhile objective. Shannon developed the idea of using a game tree of a certain depth and advocated using a board evaluation function Shannon (1950) that allocates a numerical score according to how good a board position is for a player. The method for determining the best moves from these is called minimax Dimand and Dimand (1996). Samuel used this in his seminal paper on computer checkers Samuel (1959) in which he refined a board evaluation function. After two computer players have played a game, the loser is replaced with a deterministic variant of the winner by altering the weights on the features that were used, or by replacing features that had very low weight with other features. The current world champion at checkers is a computer program called Chinook Schaeffer (1996), which uses deep minimax search, a huge database of end game positions and a handcrafted board evaluation function based on human expertise.

More recently, board evaluations functions for various games have been obtained using ANNs and often evolutionary techniques have been used to adjust the weights: Othello Moriarty and Miikulainen (1995), Go Richards et al. (1998), Chess Kendall and Whitwell (2001), and Checkers Chellapilla and Fogel (2001).

Although the history of research in computers playing games is full of highly effective methods (e.g. minimax, board evaluation function), it is highly arguable that human beings use such methods. Typically, they consider relatively few potential board positions and evaluate the favourability of these boards in a highly intuitive and heuristic manner. They usually learn during a game, indeed this is how, generally, humans learn to be good at any game. So the question arises: How is this possible? In our work we are interested in how an ability to learn can arise and be encoded in a genotype that when executed gives rise to a neural structure that can play a game well. Each agent (player) has a genotype that grows a computational neural structure during the course of the game. Our method employs very few, if any, of the traditional notions that are used in the field of Artificial Neural Networks. Instead, most aspects of neural functions are obtained ab initio through evolution of the genotype.

Learning to play checkers is difficult especially when you cannot see the board. In order to do this, first of all, the agent starts with a single neuron, with random number of dendrites and branches and builds a computational network that is capable of solving the task while maintaining a stable network (i.e. not losing all the branches). Secondly, it must find a way of processing the signals from the board
and discriminate among them. Thirdly, it must understand the spatial layout of the board (positions of its players). Fourth it must develop a memory or knowledge about the meaning of the signals from the board, and fifth, it should develop a memory of previous moves and whether they were beneficial or deleterious. Also it should understand the benefits of making a king or jumping over and finally, and most importantly, it must do all these things while playing the game. Over the generations, the agents learn from each other about favourable moves, this learning is transferred through the *genes* from generation to generation.

### 5.3.1. Rules of Checkers

Checkers is played with different board dimensions internationally, however, traditional English checkers is always played on an 8 x 8 board with alternate light and dark squares. Checkers is played by two people, on opposite sides of a playing board. Each player has twelve pieces, with one player having dark pieces and the other light pieces. At the start these pieces are placed on the alternating squares of the same colour. The player with the dark pieces makes the first move unless stated otherwise. Pieces are allowed to move forward diagonally one square at a time unless a jump is possible. A jump occurs when a piece can jump diagonally over an opposing piece and land in an unoccupied square on the other side. When a piece jumps over the opponent’s piece, the opponent’s piece is removed. Multiple jumps are also possible in one go, removing several opponent’s pieces if another jump is possible.

When a piece reaches the last row i.e first row of the opponent, it becomes a king. A normal piece can only move in forward direction, but once it become a king it can move both in forward and backward directions. The jump has priority over all other moves, however if more than one jump is possible, the player can choose which jump is preferred. During a jump if a piece becomes a king, then it can jump like a king from there onward. There are different variants of games played around the world. In all variants, the player who has no pieces left or cannot move anymore loses the game unless otherwise stated.

### 5.3.2. Inputs and outputs of the System

Input is in the form of board values, which is an array of 32 elements, each representing a playable board square. Each of the 32 inputs represents one of the five different values depending on what is on the square of the board (represented by $I$). The maximum potential is $M = 2^{32} - 1$. The values taken by $I$ are as follows:

- if king, $I = M$
- if piece, $I = (3/4)M$
- if opposing piece, $I = (1/2)M$
- if opposing king, $I = (1/4)M$
- if empty, $I = 0$

The board inputs are applied in pairs to all the sixteen locations in the 4x4 CGPDN grid (i.e. two input axo-synapse branches in every grid square) as the number of playable board positions are thirty-two (as shown in figure 10). Figure 10 shows the interfacing of checkers board with CGPDN, input axo-synapse branches are allocated for each playable board position. These inputs run programs encoded in the axo-synapse electrical chromosome to provide input to CGPDN (i.e. the axo-synapse CGP updates the potential of neighbouring dendrite branches).

Input potentials of the two board positions and the potentials of neighbouring dendrite branches are the inputs to the axo-synapse electrical processing chromosome. This chromosome produces the updated values of the potentials of the dendrite branches in that particular CGPDN grid square. In each CGPDN grid
square there are two branches for two board positions. The axo-synapse chromosome is run for each square one by one, starting from square one and finishing at sixteenth.

Output is in two forms, one of the outputs is used to select the piece to move and second is used to decide where that piece should move. Each piece on the board has an output dendrite branch in the CGPDN grid. All pieces are assigned a unique ID, representing the CGPDN grid square where its dendrite branch is located. So the twelve pieces of each player are located at the first twelve CGPDN grid squares. The location of output dendrite branch does not change when a piece is moved as the ID of a piece represents the branch location not the piece location. Each of these branches has a potential which is updated during CGPDN processing. The values of potentials determine the possibility of a piece to move, the piece that has the highest potential will be the one that is moved, however if any pieces are in a position to jump, then the piece with the highest potential of those will move. In addition, there are also five output dendrite branches distributed at random locations in the CGPDN grid. The average value of these branch potentials determine the direction of movement for the piece. Whenever a piece is removed from the board its corresponding dendrite branch is removed from the CGPDN grid.

5.3.3. **CGP Developmental Neuron (CGPDN) Setup**

The CGPDN is arranged in the following manner for the checkers experiments:

1. Each player’s CGPDN has neuron and its branches located in a 4x4 grid.
2. Initial number of neurons is 1.
3. Maximum number of dendrites is 5.
4. Maximum number of dendrite branches is 200, and axon branches is 100.
5. Maximum branch state-factor is 7.
6. Maximum soma state-factor is 3.
7. Mutation rate is 5%.
(8) Maximum number of nodes per chromosome is 200.
(9) Maximum number of moves is 20 for each player.
(10) Chromosome length is 800 integers (200 nodes x (3 connections per node + Node function)).

Here, we chose to represent potential and other parameters by 32-bit integers.

5.3.4. Fitness Calculation

The fitness of each agent is calculated at the end of the game using the following equation:

\[ \text{Fitness} = A + 200(K_P - K_O) + 100(M_P - M_O) + N_M, \]

Where \( K_P \) represents the number of kings, and \( M_P \) represents number of men (normal pieces) of the player. \( K_O \) and \( M_O \) represent the number of kings and men of the opposing player. \( N_M \) represents the total number of moves played. ‘A’ is 1000 for a win, and zero for a draw. To avoid spending much computational time assessing the abilities of poor game playing agents we have chosen a maximum number of moves. If this number of moves is reached before either of the agents win the game, then \( A = 0 \), and the number of pieces and type of pieces decide the fitness value of the agent.

5.3.5. Experimental Setup for CGPDN evolved against MCP

The CGPDN agent plays against a minimax based checker program (MCP) \(^1\). The initial population of five agents, each starts with a small randomly generated initial network and randomly generated genotypes. The genotype corresponding to the agent with the highest fitness at the end of the game is selected as the parent for the new population. Four offspring formed by mutating the parent are created. Any ability to learn acquired by an agent is obtained through the interaction and repeated running of program encoded by the seven chromosomes within the game scenario.

We always allowed the MCP to make the first move. The updated board is then applied to an agent’s CGPDN. The potentials representing the state of the board are applied to CGPDN using the axo-synapse (AS) chromosome. The agent CGPDN is run which decides its move. The game continues until it is stopped. It is stopped if all its or its opponent players are taken, or if the agent or its opponent can not move anymore, or if the allotted number of moves allowed for the game have been taken.

5.3.6. Results and Analysis

We have evolved agents against MCP in a number of evolutionary runs for 1500 generations and plotted it in figure 11. From the fitness graph, it is evident that the agent plays poorly at the early stage of evolution, but as the evolution progresses, the agent starts playing increasingly better and after 1250 generations, it begins to beat the opponent by three and four pieces margin. MCP is using minimax at ply level of 5. The agent plays with a different strategy every time and finally manages to beat the opponent. The agent receives signals from the board and produces moves accordingly, but as evolution progresses, the agent begins to “understand” the board and play better. This is evident from the fitness graph shown in figure 11. The fact that the agent is using a single neuron as a computational system and still manages to beat a program based on human intelligence is impressive.

Table 1 shows a game played between the well evolved agent and MCP. This is presented to demonstrate the level of play that the two players play. Figure 13

\(^1\) The checkers program, chkit.dll is available at http://www.fierz.ch/download.php
Figure 11. Fitness of CGPDN based player against MCP shows various stages of the game along with the corresponding neuron structure updated as a result of game scenario. Figure 12 shows the variation in the number of axon and dendrite branches of the CGP neuron during the game. Table 1 and figure 13 shows the complete game, the game start with black (MCP) making the first move by moving its piece forward from square 12 to 15. The updated board is applied as input to the CGPDN causing white (CGPDN) to move a piece forward from square 21 to 17 as a result of signal received from CGPDN to motor neuron. The motor neuron receives signals using virtual dendrite branches distributed in the CGPDN Grid. Initially the neuron has a small branching structure as evident from the first neuron image in figure 13 (Row 2, Column 1). Mutual exchange of pieces occur at various stages of the game and the neural branching structure continues to develop. Move 16 shows a double jump by the white piece, while making a king in the process. The important breakthrough occurs when black makes a blunder at the previous move causing white to not only take two black pieces in one move but also for its piece to become a king so that it can move both in forward and backward direction. Figure 13 show the move on the third row and last column. At this stage the CGPDN has the maximum dendrite branching structure so it can sense the signal from the board through its branches and act accordingly as evident from figure 13 and figure 12. The game continues until the allotted number of moves (40) are taken with white (CGPDN) having one king and a piece advantage over black (MCP).

5.3.7. Generality and Robustness

In order to assess the agent’s general game playing ability we have conducted a range of experiments by allowing the agent to play against five different opponents with various playing strengths. The neuron inside the agent starts with a random branching structure that continues to develop during the game while the opponent uses an evolved genotype that stays fixed during the course of the game. The agent play against completely new opponents that it had never played before during the course of evolution. The opponent’s level of play is evident by the number of generations for which it is evolved. It beats the 50th generation agent by one king and two pieces within 76 moves. An agent evolved for 100 generations also by one king and two pieces but in 83 moves, the 1000th generation agent by one king in 111 moves and finally the 1200th generation by one piece and a king in 120 moves. In the final case, the agent lost the game to a 1300 generations evolved player by one king and 4 pieces in 59 Moves. It is worth mentioning that the agent was trained (evolved) to play forty moves. It never played a game beyond forty moves.
Gul Muhammad Khan and Julian F. Miller

Table 1. The first 40 moves of a game between a highly evolved player (white) against MCP (black)

<table>
<thead>
<tr>
<th>S.No</th>
<th>Black Move</th>
<th>White Move</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>B1 12 - 15</td>
<td>W2 21 - 17</td>
</tr>
<tr>
<td>2</td>
<td>B3 10 - 13</td>
<td>W4 17 - 10</td>
</tr>
<tr>
<td>3</td>
<td>B5 5 - 14</td>
<td>W6 23 - 20</td>
</tr>
<tr>
<td>4</td>
<td>B7 1 - 5</td>
<td>W8 25 - 21</td>
</tr>
<tr>
<td>5</td>
<td>B9 14 - 19</td>
<td>W10 29 - 25</td>
</tr>
<tr>
<td>6</td>
<td>B11 5 - 10</td>
<td>W12 20 - 16</td>
</tr>
<tr>
<td>7</td>
<td>B13 10 - 13</td>
<td>W14 28 - 23</td>
</tr>
<tr>
<td>8</td>
<td>B15 19 - 28</td>
<td>W16 32 - 23</td>
</tr>
<tr>
<td>9</td>
<td>B17 13 - 17</td>
<td>W18 16 - 12</td>
</tr>
<tr>
<td>10</td>
<td>B19 7 - 16</td>
<td>W20 23 - 19</td>
</tr>
<tr>
<td>11</td>
<td>B21 15 - 20</td>
<td>W22 24 - 15</td>
</tr>
<tr>
<td>12</td>
<td>B23 11 - 20</td>
<td>W24 22 - 18</td>
</tr>
<tr>
<td>13</td>
<td>B25 8 - 12</td>
<td>W26 26 - 22</td>
</tr>
<tr>
<td>14</td>
<td>B27 17 - 26</td>
<td>W28 30 - 21</td>
</tr>
<tr>
<td>15</td>
<td>B29 9 - 13</td>
<td>W30 18 - 9</td>
</tr>
<tr>
<td>16</td>
<td>B31 2 - 5</td>
<td>W32 9 - 2</td>
</tr>
<tr>
<td></td>
<td></td>
<td>W33 2 - 11</td>
</tr>
<tr>
<td>17</td>
<td>B34 20 - 23</td>
<td>W35 27 - 20</td>
</tr>
<tr>
<td>18</td>
<td>B36 16 - 23</td>
<td>W37 22 - 18</td>
</tr>
<tr>
<td>19</td>
<td>B38 12 - 16</td>
<td>W39 11 - 14</td>
</tr>
<tr>
<td>20</td>
<td>B40 16 - 20</td>
<td>W41 19 - 15</td>
</tr>
</tbody>
</table>

Table 2. Results of evolved agents playing against various opponents not seen during evolution

<table>
<thead>
<tr>
<th>Game Number</th>
<th>Winning Margin of CGPDN Agent</th>
<th>Level of opponent</th>
<th>Number of Moves to win</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2 pieces and 1 king</td>
<td>50</td>
<td>76</td>
</tr>
<tr>
<td>2</td>
<td>2 pieces and 1 king</td>
<td>100</td>
<td>83</td>
</tr>
<tr>
<td>3</td>
<td>1 king</td>
<td>1000</td>
<td>111</td>
</tr>
<tr>
<td>4</td>
<td>1 piece and 1 king</td>
<td>1200</td>
<td>120</td>
</tr>
<tr>
<td>5</td>
<td>lost by 1 king and 4 pieces</td>
<td>1300</td>
<td>59</td>
</tr>
</tbody>
</table>

during the course of evolution. From the results shown in table 2 it is evident that as the level of play of the opponent increases, the winning margin decreases, thus demonstrating clearly that we are able to evolve a genotype that allows a neuron to produce a structure that can play checkers intelligently.

5.4. Training the network over a series of games

We conducted experiments to ascertain whether the learning capability of the CGPDN network would be enhanced if it is evaluated over a series of games (5) against the MCP. So in every game the agent starts with a developed network from previous (unless it is the first game in which case it begin with a random network) game and is allowed to continue to develop during the five game series. We have calculated the average fitness over five games and based on this we promote agents from one generation to the next. Agents were evolved for two thousand generations.

In this case we evaluated the performance of evolved agents over a larger series
of games sequence (500). A well evolved agent from generation 2000 is taken and is allowed to play against an agent from 50th generation. The rules of the game are set such that both the agents are allowed to play 20 moves each. The 50th generation agent always begins playing with a same initial random network, the one it is trained on, whereas the 2000th generation agent continues with the developed network it had at the end of the previous game. The 2000th generation agent plays as white while the 50th generation agent plays as black. The genotype of the agents is kept the same throughout this experiment, only the architecture develops (neurites change) and shape (neurite branches continue to shrink and grow) during the course of game.

At the end of each game we have calculated the fitness of both the agents and plotted them against each other. Figure 15 shows the fitness variations of the two agents calculated at the end of every game. Figure 14 shows the fitness averaged over five consecutive games. The 2000th fitness represented by a continuous line is always above zero, showing clearly that its performance is better throughout 500 games, even though its network continually develops and changes. Figure 16 shows the accumulated fitness graph of the well evolved agent over 500 games. From these graphs it is evident that although the network changes in every game, the network maintains its integrity of getting higher fitness over the less evolved agent, demonstrating that it does not forget how to play checkers at a better level. However, figure 15 shows some peaks at various stages, these are the cases when the highly evolved agent beats the opponent within 20 moves. This is very interesting, as while being evolved, the agent was never able to beat its opponent within 20 moves, as it is trained against a high skilled checker program (MCP). But during development stage when it is allowed to play more than 5 games it is able to do so. As it continues to develop and play without evolution, its ability to beat the opponent within 20 moves seems to increase as is shown by the average fitness remaining above the x-axis in Figure 14.

From rigorous analysis, we found that almost every time a new game is started, the network although different (through development) repeats its initial moves. This causes it to make two double jumps taking four pieces of the opponent during
initial stage of the game. This is interesting behaviour as the opponent always starts with the same initial structure and it will repeat the same moves if the developed agent (well evolved) also does. A number of games are studied starting from game 100 (when the network appears to stabilize), the agent seems to repeat its first 8 moves almost every time and this causes the agent to take two double jumps over the opponent giving it an extra advantage. The developed agent does not know when one game ends and another begins, yet it makes the same initial move with a different network, forcing the opponent to repeat the same mistakes, thus causing opponent to lose the game. This suggests that the agent responds to change in board positions and is able to make the same moves with a different network. This also demonstrates that stable behaviour can be obtained when CGPDN is changing.

6. Conclusion

We have described a neuro-inspired developmental model of neuron with biological realistic morphology and used Cartesian genetic programming to evolve a multi-chromosome genotype. We have evaluated the learning potential of this model in the context of solving complex mazes and learning how to play games (checkers) through experience. Artificial agents were provided with this model as their signal processing and decision making system. The genotypes of the agents are evolved for the desired learning capabilities from generation to generation, while the neural structure grows and changes in response to the agent behaviour and interactions with the environment. The learning that occurs during the process of solving mazes
Figure 14. Graph showing the average fitness variation of a well evolved agent (white) playing five games of checkers against a less evolved agent (black).

Figure 15. Graph showing the fitness variation of a well evolved agent (white) against a less evolved agent (black) and playing games is stored in the neural structure and weights of the agent, whereas the capability of learning is obtained solely through evolution by evolving the genotype. The eventual aim is to see if it is possible to evolve a network that can learn by experience. This model is biologically more plausible than previous ANN models and demonstrates interesting learning behaviour.

In the case of checkers, we have investigated the evolution of checkers playing agents that are controlled by a single developmental neuron and demonstrated
that it can play intelligently and beat a human intelligence based agent by a large margin. We have also tested the CGPDN based agent for its generality. It beats the low level players with large margins in less time but tends to have problems beating high level players. This is a clear demonstration of its learning capabilities. We have also trained the network on a series of five games and tested it for experiential learning against a less trained network in a 500 games scenario in which the highly evolved agent was allowed to continue to develop while the less evolved agent always started each game from the same initial random neural structure. The results obtained at the end of the game series demonstrated that network (CGPDN) is capable of maintaining its integrity of winning the game, even when its architecture is updated and is also able to find a default minimum structure which is suitable for it. In further analysis, we found that the network starts repeating similar moves each time a new game is started. This demonstrates that these agents are capable of carrying out intelligent action repeatedly in the board environment, even when its network is different.

This work has a great potential of evolving systems that can learn continuously and will be explored in a range of intelligent systems scenario and applications. However, the model we have described is admittedly rather complex. Despite this it is an enormous simplification of biological neural processes. The model is compartmental with seven evolved programs largely responsible for the behaviour. This means that at least in principle, some of these aspects could be fixed in some way, so that the relative importance of different aspects for learning can be assessed. This remains for the future.

References


REFERENCES


REFERENCES


### Appendix A. Information Processing in the Network

Information processing in CGPDN starts with the following three steps:

1. Produce a random CGPDN (the default structure)
2. Produce initial genotypic population, with each population member consisting of seven chromosomes.
3. Specify the number of inputs and outputs of CGPDN, and distribute them at random locations in the network.

In order to produce a random network the following parameters need to be specified:

1. Maximum number of branches ($N_{b_{\text{max}}}$)
2. Maximum number of dendrites ($N_{d_{\text{max}}}$)
3. Max Neuron state-factor ($N_{s_{f}}$)
4. Max branch state-factor ($B_{s_{f}}$)
5. Mutation rate ($\mu$)
6. Dimensions of 2D space for neuron (Number of rows ($N_{\text{row}}$) and columns ($N_{\text{col}}$))
7. Neuron and branch life threshold ($H_{\text{min}}$), offspring threshold ($H_{\text{max}}$).
8. Neuron and branches health, weight and potentials reduction factors ($\sigma_{H_{ab}}$, $\sigma_{H_{as}}$, $\sigma_{W_{ab}}$, $\sigma_{W_{as}}$, $\sigma_{P_{ab}}$, $\sigma_{P_{as}}$, and $\sigma_{P_{s}}$)

A CGP Developmental neuron is produced with a random number of dendrites and axon branches. Each dendrite is assigned with a number of branches. This neuron is placed at a random location with its branches distributed in a 2-dimensional CGPDN grid. An initial value of health is assigned to both soma and branches. A random value of weight is assigned to soma and, both dendrite and axon branches. Branches are also assigned with random resistance values. The thresholds for all the operations must be specified.

The second step, genotype initialization consists of specifying the following:

1. Number of offspring, ($\lambda$) (Evolutionary strategy $1 + \lambda$)
2. Number of nodes per chromosome
3. The set of node functions and the number of connections per node.

A population of random genotype is produced using the above specifications, and then evolved to get the desired network behaviour.

In the third step we need to specify the initial number of inputs and outputs of the system and their corresponding locations in the CGPDN grid.
Figure A1. A diagram showing how external inputs (as potentials) are introduced into the CGPDN by executing axo-synapse program. The external input is applied to the axo-synapse program (AS) together with the potentials on neighbouring dendrite branches. After the AS program is run the axo-synapse potential and the potentials on neighbouring dendrite branches are updated.

After producing the initial population the rules for information processing the network are specified as follows:

1. Input from the environment should be applied to the network at the start of the processing.
2. The network is run for five cycles ($N_{cycles}$) before reading the output.
3. The potential of soma and branches ($\sigma_{P_{db}}$, $\sigma_{P_s}$ and $\sigma_{P_{as}}$) is reduced by 10% after every cycle.
4. After five cycles of the network, the health and weights of the soma and branches are reduced by 10%.
5. The soma threshold potential ($\eta_{th}$) is also reduced by double the reduction factor of soma potential after every cycle.
6. The state-factor of all the branches and soma are reduced by one unit after every cycle, to allow them to move toward activity.

After specifying the rules for information processing in the network, we can start running the network. First of all, input is applied to the CGPDN network using the following steps:

1. Find the location of each input.
2. Select the active dendrite branches at that location
3. Bias the input axo-synaptic branch potential. Biasing means duplicating their value in input vectors, equal to number of active dendrite branches, in this case).
4. Apply potentials of active dendrite branches and biased input potential to the program encoded in axo-synapse electrical chromosome (AS).
5. The axo-synapse electrical chromosome program updates the values of the dendrite branch potentials (as shown in figure A1)
6. Repeat the same process at each input branch.
After applying the input to CGPDN, run it for a number of cycles (5 in this case). The following steps are performed to run the network for one cycle:

(1) Start processing a neuron by using the following steps:
   a) Start processing each dendrite connected to a neuron.
   b) At each dendrite select all the active dendrite branches attached to it, and apply their potential values along with biased soma potential to the CGP dendrite electrical processing chromosome. This produces their updated values as output. Biasing means applying the soma potential the same number of times as the number of active dendrite branches.
   c) Process each branch potential based on the resistance, weight and health values using Eqn. 1.
   d) After this process if any branch is active, its life cycle is run by applying its resistance, weight and health as input to CGP dendrite branch life cycle chromosome. This updates these values, and depending on the updated value of resistance the decision to move the branch from the current location is taken. The health decides whether the branch should produce offspring, die or remain the same. Producing offspring means the creation of another branch at the same location with an initial health, and random weight and resistance values. If the branch dies it is removed from the dendrite.
   e) The same process is repeated for all the dendrites and their corresponding branches. After processing all the dendrites, the average value of potentials of all the dendrites is taken, which in turn is the average value of all the active dendrite branches attached to them. This potential and the soma potential are applied as inputs to the CGP soma electrical processing chromosome. This produces the updated value of the soma potential ($\bar{P}$) as output.
   f) This updated soma potential is processed using health (H) and weight (W) of the soma using Eqn. 2.
   g) After processing, the soma potential is compared with the soma threshold potential, if it is higher than the soma threshold, then the soma fires. This means that the soma potential is set to the maximum value. The soma threshold potential and soma state-factor are set to their maximum values (Maximum values are variable and user-defined). This keeps the soma inactive for a number of cycles.
   h) If the soma fires, its life cycle is run and also the potential is transferred to other neurons using axo-synaptic branches by running the program encoded in axo-synaptic electrical processing chromosome (AS).
   i) If the soma does not fire, its state-factor is adjusted based on the value of the processed potential as described earlier.
   j) If the soma life cycle is run, it takes the health and weight of soma as input and produces the updated values as output.
   k) If the soma fires the signal needs to be transferred to other neurons. This is done as follows: the axo-synaptic electrical processing chromosome is run (AS), in the active axon branches (as shown in Fig. A2). Select all the active dendrite branches in the vicinity of each active axon branch and take their potential values, and apply them together with a biased soma potential (equal to number of active branches) as inputs to the CGP electrical processing chromosome. The chromosome produces the updated potentials of all the dendrite branches, along with axo-synaptic potentials. After this process the axo-synaptic
Figure A2. Axosynaptic potential transfer to the neighbouring dendrite branches, showing soma, two axo-synapse branches, a grid square and a number of dendrite branches attached to other neurons and their corresponding potentials, and a weight processing chromosome. This shows the situation just before the axo-synapse program (AS) in top right grid square is executed. After this the potential at this axo-synapse and the potentials at the neighbouring dendrites branches will be updated.

Figure A3. Weight processing of the neighbouring dendrite branches. The diagram shows soma (S), two axo-synapse branches (AS), a grid square and a number of dendrite branches attached to other neurons and their corresponding weights. A weight processing chromosome is shown highlighted in the grid square. The weight processing program (WP) has just been executed and updated the weights of all axo-synapses and dendrite branches in the grid square. \( w_i \) shows the updated weights of axo-synapse and dendrite branches.

potential is processed using Eqn. 3.

After getting the processed potential the program encoded by the weight processing chromosome is run. This takes the weights of the active dendrite branches in the vicinity of the axon branch and its axo-synaptic weight as input and produces the updated values as output (as shown in figure A3).

The axo-synaptic potential is assigned to the dendrite branch whose weight is the maximum after weight processing as shown in figure A4.

l) Also after the axo-synaptic electrical processing, if the potential of the axo-synaptic branch is raised above a certain level, it is kept active and its life cycle is run.

m) The life cycle of axo-synapse takes health and resistance of the axon as input and produces their updated values. The change in resistance of the branch is compared with a threshold value to decide whether the branch should move or stay at the same location. Also the health
Figure A4. Transfer of potential to highest weight after weight processing. The diagram shows the soma (S), two axo-synapse branches (AS), a grid square and a number of dendrite branches attached to other neurons and the weight processing chromosome (WP). After the WP is run a branch is identified with the highest weight ($W_i$). The axo-synapse potential is subsequently transferred to the corresponding dendrite branch.

of the branch is checked and if it is above the offspring threshold, it results in the production of another branch at the same location, with an initial health and random weight and resistance. If the health falls below the life threshold it dies and is removed from the axon.

n) The same process is repeated in all the axon branches

After running the network for five cycles, the output is read from the output branches. The output branches are affected by the network processes, through their updated potential values.

After completing the task the following steps are performed:

1) The network fitness is assessed.
2) The genotype with highest fitness is selected.
3) Using mutation, new offspring chromosomes are produced.