Request Confirmation Networks: A Cortically Inspired Approach to Neuro-Symbolic Script Execution

Request Confirmation Networks: A cortically inspired approach to neuro-symbolic script execution

Katherine Gallagher

A Thesis in the Field of Software Engineering

for the Degree of Master of Liberal Arts in Extension Studies

Harvard University

May 2018

Abstract

This thesis examines Request Confirmation Networks (ReCoNs), hierarchical spreading activation networks with controlled top-down/bottom-up recurrency that are inspired by constraints of cortical activity during execution of neuro-symbolic sensorimotor scripts. ReCoNs are evaluated in the context of the Function Approximator, a showcase implementation that calculates a function value from a handwritten image of the function. Background is provided on biological and artificial neural networks, with emphasis on biomimetic approaches to machine learning.

Dedication

To my parents, for their infinite love and support.

To Joseph, who has explored the mind.

To Laura, who believed me.

To Jennifer, for lighting the way.

To Joscha, a fellow stranger in a strange land.

To Seb, who made me myself, and without whom this could not have been.

Acknowledgments

I would first like to express my immense appreciation for the Program for Evolutionary Dynamics and all its members. Special thanks to May Huang for her invaluable support, and to Dr. Martin Nowak, who chose to add my dynamic to his program's evolution and to whom I will forever be grateful.

I would also like to recognize Harvard Extension School, especially Dr. Jeff Parker and Dr. Sylvain Jaume for their guidance in the thesis process, and Prof. Ben Gaucherin, who catalyzed my path at Harvard.

Sincere thanks as well to Dr. Adam Marblestone for lending his expertise, and to Priska Herger, both for her original work on Request Confirmation Networks and her enormously appreciated tolerance of, and responsiveness to, my Slack messages.

Finally, my unending gratitude to Dr. Joscha Bach, for his friendship, mentoring, generosity with his time and brilliance, endlessly fascinating conversations, and apparently limitless patience.

Table of Contents

List of Tables

List of Figures

Chapter 1.

Introduction

"...we'll show that you can build a mind from many little parts, each mindless by itself."

- Marvin Minsky, *The Society of Mind*

This thesis presents Request Confirmation Networks (ReCoNs), a biologically-inspired neural network architecture proposed to simulate neuro-symbolic script representation and execution in the cerebral cortex.

1.1. Motivation

Machine learning, the field of computer science concerned with giving machines the ability to autonomously draw increasingly accurate predictions from prior experience, is currently experiencing a renaissance on a scale that has prompted forecasts of nearly boundless future growth (Knight, 2016). Significant advances are rapidly being made across machine learning specialties, from strategic game play (Silver et al., 2017) to medical diagnostics (Rajpurkar et al., 2017; Weng, Reps, Kai, Garibaldi & Qureshi, 2017). However, these developments have yet to lead to what some consider to be "the holy grail" of computer science (CNBC, 2016): artificial general intelligence (AGI).

The concept of AGI, the capacity of a machine to perform all intellectual tasks at or beyond a human level, is predicated on the assumption that computation can replicate or emulate biological intelligence. This premise is supported by proponents of the view that "[b]rains are, at a fundamental level, biological computing machines" (Cox & Dean, 2014, p. R921), and borne out in practice by disciplines such as computational neuroscience, which treat the brain as an information processing structure with dynamics that can be reliably predicted by mathematics and fully described in computable algorithms.

Although it remains unclear whether closely imitating neuroanatomy and/or neural function is a necessary prerequisite for intelligence, or if there may be alternative physiological configurations that result in intelligent behavior, computer science has converged on artificial neural networks (ANNs), systems that bear structural and functional resemblance to biological neural networks and neural activity, as a valuable component in machine learning and a compelling contender for the foundation of an AGI architecture (Fernando et al., 2017).

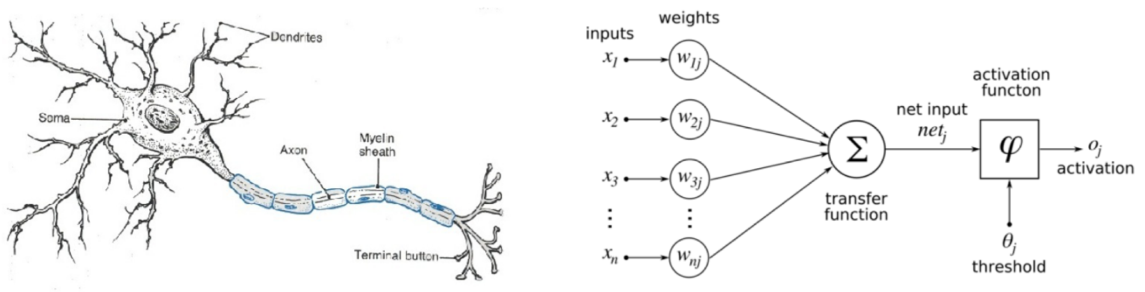


Fig. 1.1. Comparison of a biological and artificial neuron

Left: A diagram of a biological neuron (Carlson, 1992). Right: A diagram of an artificial neuron (Burgmer, 2005).

Research correlating biological neural network structure to its emergent function appears to support the premise that simulating biological structure may lead to simulated biological function (Hermundstad, Brown, Bassett & Carlson, 2011; Reimann et al., 2017). Basic neuronal activity has already been artificially replicated with increasing specificity at the level of individual neurons (Simon et al., 2015), and there are ongoing attempts to reverse engineer algorithms from cortical tissue (Cepelewicz, 2016). However, despite a growing body of data, much about the brain and its inner workings remains unclear (Gorman, 2014; Lam, 2016).

The persistent opacity surrounding the details of neural activity itself lends incentive to the search for AGI. While AGI may be a desirable goal on its own merits, it is also likely that the quest to imbue machines with intelligence will result in a better understanding of the human mind and the mechanisms of cognition. Simulating cognitive functions may reveal details about the internal architectures from which they emerge; this assumption has already led to ANNs becoming a generally accepted model for studying neural information processing.

In the pursuit of AGI via computational neural emulation, rather than attempting to faithfully recreate from the ground up biological structures whose subroutines, dependencies, and other possible complicating factors are still largely obscure to us, it might be possible to artificially recreate brain function at a higher level of abstraction. To this end, ReCoNs offer a cortically inspired solution to computationally implementing neuro-symbolic sensorimotor scripts.

The current generation of perceptual machine learning systems approximates hierarchical feature detection in sensory areas. As in biological systems, this is a likely building block in a greater cognitive architecture that can dynamically reconfigure the hierarchies, facilitate motivational learning, and drive symbolic abstraction, sensory-motor integration, and specific higher-level functionality such as self-reflection, language, and social cognition. However, current machine learning systems are generally feedforward networks that learn to approximate continuous functions after being trained with the chain rule in a narrowly defined arena, which results in systems that may demonstrate proficiency in a specific task, sometimes even exceeding human performance, but cannot translate knowledge across domains, and do not demonstrate the universal, adaptive, online learning found in biology.

This has been a primary point of contention against so-called "connectionist" networks in the historic debate between "connectionist" and "symbolic" AI (Sun, 1999). The former consists of the large networks of simple units connected by tunable links that have become synonymous with modern machine learning, whereas the latter refers to networks that focus on developing symbolic representations and associations that can serve as the basis for general knowledge acquisition. Although the symbolic paradigm has largely been eclipsed by the enormous success of connectionist systems, it offers some advantages that may prove integral to the construction of artificial learning systems that more closely emulate the flexibility of biological learning, such as the capacity to represent recurrencies, composition operators, and grammars.

The symbolic approach attempts to mimic the logical foundation that enables biological systems to autonomously configure basic neuro-symbolic representations of acquired knowledge into complex hierarchies capable of representing any combination of information. In the brain, these hierarchies can be activated *bottom-up*, based on perceptual input from sensory modalities, or *top-down*, based on knowledge or directed attention (Buschman & Miller, 2007; Gilbert & Sigman, 2007). The cooperative interaction and feedback between bottom-up and top-down processing has been shown to occur at nearly every level of cortical sensory processing, making feedforward mechanisms alone unlikely to account for either the brain's flexible and invariant pattern recognition in changing environments or its corresponding behavioral responses (Gilbert & Sigman, 2007).

ReCoNs unite connectionist and symbolic approaches in a top-down/bottom-up hierarchical spreading activation network that can be used to extend a neural learning system beyond feature detection to self-driven experimentation and modification of its acquired representations without the need for a central control architecture. By providing a constrained recurrency framework in which an ANN can symbolically represent, test, and internally reconfigure its own learned hypotheses, ReCoNs may be able to serve as building blocks in a general learning network with the ability to not just learn and respond to sensory input based on predetermined rules, but to locally formulate and execute plans based on assimilated knowledge.

ReCoNs are therefore proposed both specifically as a computational interpretation of sensorimotor script execution and, more generally, as a possible foundational component in bridging the current neural network-based machine learning systems into a more general learning architecture resembling that of the cerebral cortex.

1.2. Outline

Chapter 2 offers a brief history of the origins of machine learning, examines the biological foundations of ANNs, discusses representative types of ANNs and their applications, and surveys machine learning approaches designed to emulate cortical activity.

Chapter 3 establishes the neurological basis for ReCoNs, specifies ReCoN subcomponents and their functions, and outlines the flow of activation through the network from initial request to final confirmation or failure. The chapter concludes with a study of the pilot ReCoN implementation described in Bach and Herger (2015).

Chapter 4 provides a system overview and implementation details of the Function Approximator, a showcase application of ReCoNs. Using a combination of a pre-trained multilayer perceptron (MLP) and a ReCoN, the system takes handwritten algebraic functions as input, identifies their component symbols, and outputs the value of the function. This chapter introduces MESH, a graphical user interface (GUI) for neural networks, which adds a visualization component to the Function Approximator.

Chapter 5 presents and analyzes the results of implementing the Function Approximator, and evaluates ReCoNs in the context of other biomimetic machine learning approaches as a potential model for sensorimotor script execution in the cerebral cortex.

Chapter 6 summarizes the findings of this thesis and suggests possible future directions for ReCoN development.

Chapter 2.

Background

This chapter includes a brief history of the origins of machine learning and ANNs, a basic explanation of biological neural networks, an overview of various types of ANNs and their applications, and a current survey of biomimetic neural network approaches to machine learning.

2.1. The origins of machine learning

The origin of the field of machine learning might be traced back to the Dartmouth Summer Research Project on Artificial Intelligence, a workshop held during the summer of 1956 and organized by John McCarthy, Marvin Minsky, Nathaniel Rochester, and Claude Shannon. Starting from the premise that all elements of intelligence can be precisely defined, and therefore are capable of being simulated by a machine, the organizers proposed to "find how to make machines use language, form abstractions and concepts, solve kinds of problems now reserved for humans, and improve themselves" (McCarthy, Minsky, Rochester & Shannon, 1955, p. 2).

Soon thereafter, in 1957, Frank Rosenblatt developed the perceptron, a linear classification algorithm he devised as a model of information storage and organization in the brain (Rosenblatt, 1958). Building upon the simplified logical specifications for a biological neuron introduced by Warren McCulloch and Walter Pitts in 1943 (McCulloch

& Pitts, 1943), the perceptron was first implemented on an IBM mainframe and could distinguish between left and right after training cards printed with squares (Yovits, 1993, p. 341). It was later adapted for image recognition as the Mark 1 Perceptron, a custom-built machine that demonstrated the ability to categorize patterns based on geometric similarity (Hay, Lynch & Smith, 1960).

Shortly following the debut of Rosenblatt's perceptron, one of the earliest recorded usages of the term "machine learning" appeared in a manuscript written by a Dartmouth workshop attendee, Arthur Samuel. The paper, which describes Samuel's landmark work on the first computational learning system for the game of checkers, defines machine learning as the practice of "programming computers to learn from experience" without explicit instructions on how to do so, and draws a sharp distinction between systems designed to learn only specific tasks and "the neural-net approach". Although Samuel deemed the latter less efficient than the former, he claimed that it "should lead to the development of general-purpose learning machines" (Samuel, 1959, p. 535).

Not all machine learning systems employ neural networks, but those that do have garnered significant attention in recent years thanks to a rapid cycle of setting and exceeding benchmarks, notably including human performance, across a variety of arenas (Eckersley, Nasser et al., 2017). Some of the most successful algorithms involve deep learning, which harnesses multiple layers of artificial neurons. Although Samuel's prediction that neural networks would lead to AGI has yet to be conclusively proven, it remains a compelling theory nearly six decades later.

2.2. Biological neural networks

The neuron is the basic processing unit of a biological brain. A neuron typically consists of a cell body, called the "soma", dendrites, and an axon (see Figure 1.1).

In general, dendrites are short fibers that receive incoming signals from other neurons and relay them to the soma, while the axon, a single long fiber, transmits outgoing messages from the soma to other neurons or body tissues across communication interfaces called synapses. When the aggregate of the input, both excitatory and inhibitory, received by the neuron exceeds its activation threshold, the neuron “fires”, and an electrical impulse, also known as an action potential, is discharged to the axon as output and travels across a synapse to the receiving cell. Synapses can be either excitatory, encouraging the signal to be passed on, or inhibitory, dampening the signal.

According to de Garis, Shuo, Goertzel & Ruiting (2010, p. 3), "[t]he human brain has about 100 billion neurons, with each neuron connecting to roughly 10,000 others, with each synapse firing at maximum of about 10 bits per second; hence the total bit processing rate is of the order of 10^{16} bits per second." This electrochemical activity translates into brain function.

The anatomy of the human brain is highly modular, with distinct anatomical regions specializing for particular behaviors. Distinguishing patterns of information flow within these regions have prompted suggestions that they evolved to provide algorithms that solve different computational problems; for instance, large numbers of recurrent connections may imply short-term memory storage, whereas regions like the thalamus, that have activation from a variety of other areas moving through them, appear to perform information routing (Marblestone, Wayne & Kording, 2016). Signals sent and received by firing neurons in each of these areas result in corresponding function.

To illustrate a specific example: the largest part of the human brain is the cerebrum, which is divided into two hemispheres. Each hemisphere is subdivided into four major "lobes", regions specialized for various functions: the frontal lobe is commonly associated with attention and planning, the parietal lobe with sensory integration and language, the occipital lobe with visual processing, and the temporal lobe with translating sensory input into meaningful information (Ackerman, 1992).

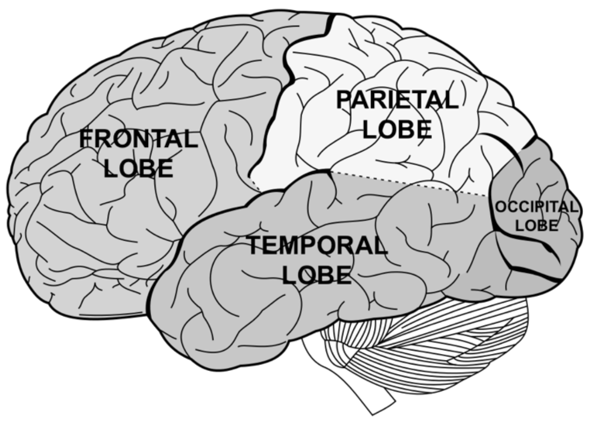


Fig. 2.2. The lobes of the cerebrum

The frontal, parietal, temporal and occipital lobes have all been shown to specialize for different brain functions. (Carter, n.d.)

Neurons in the primary visual cortex (V1), located in the occipital lobe, will fire in response to visual stimuli in their receptive fields. A V1 neuron will be most responsive to particular set of input for which it is "tuned", such as orientation or color. Activation is transmitted from V1 through higher-layer visual processing areas, which respond to more complex properties, like geometric shapes and motion, and objects, like faces. By sequentially activating the hierarchical networks that together encode the features of a particular object, neural firing patterns result in the recognition of visual input.

2.3. ANNs in machine learning

ANNs consist of “artificial neurons”, functions that sum weighted inputs to produce activations in a fashion roughly analogous to biological neurons (see Figure 1.1). Machine learning systems pass input data through layers of these nodes connected by weighted links. Similar to the hierarchical feature processing in the visual cortex, each successive layer represents an element of the data at progressively higher levels of abstraction (LeCun, Bengio & Hinton, 2015; Belinkov et al. 2017).

2.3.1. The artificial neuron

The McCulloch-Pitts model is widely credited as the first formal definition of an artificial neuron. Based on observations from theoretical neurophysiology, McCulloch and Pitts (1943) expressed neural activity in terms of propositional logic, resulting in a description of a linear threshold neuron capable of handling only binary input and output. To emulate synaptic contribution, inputs can be either excitatory (+1) or inhibitory (-1). If the sum of all inputs exceed a given threshold value T, the neuron output will be 1, else output will be 0.

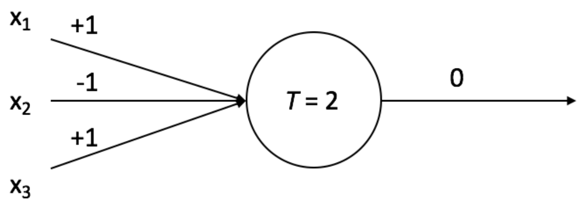


Fig. 2.3.1. McCulloch-Pitts neuron

x_1, x_2, and x_3 represent weighted inputs +1, -1, and +1 respectively. When summed, they do not exceed the threshold T (1 < 2), so the neuron output is 0.

This simple neuron can represent the linear logical operators NOT, AND, and OR, and can also be layered into more complex logic gates, as exhibited by its extension into the first artificial neural network implementation, Rosenblatt's perceptron.

Although the McCulloch-Pitts model has been superseded by more sophisticated and efficient algorithms with decimal input and output and non-linear transfer functions, the basic architecture of an artificial neuron remains recognizably derivative of this early prototype.

2.3.2. Learning with ANNs

In a general sense, ANNs "learn" by updating the weights of the links that connect neurons. Weights are coefficients by which each input passed along a link is multiplied before arriving at its target neuron, and determine how impactful a particular input will be in the neuron's calculation. Methods of training neural networks can be classified into three main learning paradigms: reinforcement learning, supervised learning, and unsupervised learning.

In reinforcement learning systems, an agent learns via repeated interactions with its environment. Each action performed by the agent generates a cost, which the agent attempts to minimize over time, in pursuit of maximizing a long-term reward. The trial-and-error mapping of environmental situations to actions based on an anticipated numeric reward imbues reinforcement learning systems with a semblance of goal-driven behavior. This characteristic resemblance to the behavior of living systems has prompted various parallels to be drawn between reinforcement learning and biology, especially in regard to the role of the dopamine system in the brain for reward-dependent learning in mammals (Montague, Eagleman, McClure & Berns, 2006).

Supervised learning offers the ability to precisely correct inferences made by a neural network by providing an "answer key" of accurate labels along with the input data. Teaching the names of everyday objects to young children is a common example of supervised learning in humans: a depiction of the object, or the object itself, is indicated to the child along with the pronunciation of the object's name, and the child is expected to repeat the pronunciation. This establishes an association between the object and its name in the child's mind at a symbolic level; by learning and storing a symbolic representation of the characteristic features that compose a tree, a child does not have to be shown an example of every kind of tree (pine, palm, etc.) in order to eventually predict that a rooted trunk with branches indicates the presence of a tree, even if the specific configuration has not previously been encountered.

Part of the process of properly forming associations is error correction. If a child points to a cat and calls it a dog, that error should be corrected by its parent or other supervisor so that the child doesn't continue building inaccurate representations based on this mistake. Similarly, supervised learning ensures that incorrect predictions made by an ANN are immediately remedied during training by providing feedback in the form of a loss function, which calculates a penalty to the system based on the difference between the expected output and the prediction.

In contrast, an unsupervised learning network converges on a function based on statistical regularities in the input data without relying on labels to guide or correct its conclusions. A similar principle appears to be at work in the brain, where learning is governed by changes in the synaptic connections between neurons (Martin, Grimwood & Morris, 2000). The generally accepted Hebbian theory of neural learning, often paraphrased as "neurons wire together if they fire together" (Löwel & Singer, 1992, p. 211), states that if a presynaptic cell is repeatedly involved in the excitation of a postsynaptic cell, the strength of the synapse between them will increase, resulting in a higher likelihood of the postsynaptic cell firing if the presynaptic cell is active. In the same way that an ANN represents acquired knowledge through updated link weights, the neuroplasticity of a biological neural network thereby depends on synaptic weight updates.

A potentially problematic characteristic of unsupervised learning is that these correlations are reinforced without regard to error. Unlike the supervised and reinforcement learning paradigms, which provide feedback on the value of particular associations, unsupervised learning strengthens connections solely on the basis of correlated firing between the neurons they connect, whether or not that firing or the resultant relation is always valid or useful.

2.3.3. Types of ANNs

Despite the parallels drawn between artificial and biological neural networks, it should be established that ANNs generally differ considerably from biology in that, whereas biological neurons map temporal and spatial inputs to discrete activation values which are in turn mapped to synaptic outputs, an artificial neuron computes an activation of a normalized weighted real-valued sum. As features are added and structures reconfigured for performance optimization in specific tasks, ANNs deviate to varying degrees further from their biological foundations.

The ANN adaptions summarized in this section were selected to provide a representative cross-section of the diverse formats and capabilities of contemporary artificial networks, as well as a very basic understanding of some of the foundational principles of the networks outlined in Section 2.4. They have discrepant claims to biological plausibility, and are by no means intended to be a complete representation of the field.

2.3.3.1. Multilayer Perceptron (MLP)

The single-layer perceptron developed by Rosenblatt was the first and most basic of neural network algorithms. Weighted inputs are fed directly into a row of output nodes, which sum the received input and fire if it exceeds their threshold. The difference between the actual output and the target output is typically used to adjust the weights and train the network.

Although this feedforward structure is capable of linear binary and multiclass classification, as Marvin Minsky and Seymour Papert famously pointed out, it cannot be trained to recognize pattern classes that aren't linearly separable and is subsequently unable to represent the exclusive-OR (XOR) operator (Minsky & Papert, 1969/2017). MLPs, perceptron architectures that incorporate one or more hidden layers, offer a solution to this problem.

The formal definition of a perceptron prescribes a threshold activation function, as seen in the McCulloch-Pitts model, but the term "multilayer perceptron" has become generally accepted to encompass architectures of neurons with arbitrary activation functions (the neurons in the MLP implementation discussed in Chapter 4 of this thesis employ a rectifier (ReLU)).

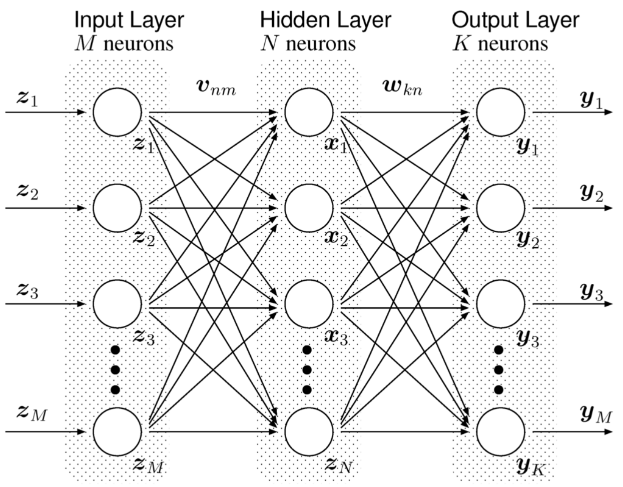


Fig. 2.3.3.1. A multilayer perceptron

A structural diagram of an MLP with a single hidden layer. (Isokama, Nishimura & Matsui, 2012)

Figure 2.3.3.1. displays the structure of a generic fully-connected MLP. Input data is fed into the first layer of neurons, which send their output to each neuron in the next layer. Activation is subsequently passed through all hidden layers until it reaches the output layer, where it can be compared against the label and the loss function can compute the appropriate feedback, which is distributed back through the network via backpropagation.

Backpropagation has emerged as an integral technique to assign credit to individual neurons for their role in the final output of an ANN. Backpropagation leverages gradient descent, an optimization algorithm that seeks the parameters that best minimize an objective function. In the case of backpropagation, the function being minimized is the network loss function. The derivative of the loss is calculated with respect to the network weights, where the derivative with respect to each weight is dependent on the derivatives of the weights in the following layer.

Despite its widespread adoption and proven utility in ANNs (the MLP classifier discussed in this thesis employs backpropagation), it is generally accepted that it is unlikely that the brain implements an exact version of the backpropagation algorithm (Bengio, Lee, Bornschien, Mesnard & Lin, 2015; O'Reilly & Munakata, 2000). Recent work suggests that biological neural networks instead approximate backpropagation solely through local Hebbian plasticity, with certain neurons encoding the difference between actual activity and predicted activity, and propagating these prediction errors through the network (Whittington & Bogacz, 2017).

2.3.3.2. Convolutional Neural Network (CNN)

Although traditional MLPs can be useful for image recognition, their fully-connected nature makes scaling to handle high dimensional input computationally prohibitive, and they are not robust to distortion or variance in input data. CNNs are a subclass of MLPs that provide solutions to these issues and have thereby come to dominate the field of image analysis: CNN architectures were the first to achieve learning-based face detection (Vaillant, 1994), currently hold the performance record on multiple image databases (Benenson, 2017), and are widely employed in technology found in self-driving cars (Tian, Pei, Jana & Ray, 2017), where feature detection and recognition is critical.

The predecessor to CNNs, the neocognitron, was inspired by Nobel Prize-winning research performed in the 1960s by David Hubel and Torsten Wiesel at Harvard Medical School, which indicated that a neuron in the visual cortex is sensitive to stimuli only within a small, specific region of the visual field, termed its "receptive field" (Fukushima, 1980). Based on their findings, Hubel and Wiesel proposed an information processing model of the visual cortex that relied on two types of specialized cells for hierarchical feature extraction: "simple cells" that learn to respond to a particular feature within their receptive field, and "complex cells" that integrate these features into composite representations (Hubel & Wiesel, 1961).

In the neocognitron, a multilayered network capable of unsupervised pattern recognition, complex cells were connected to multiple simple cells in the preceding layer that extracted the same feature at different positions. The complex cells would respond if activation was received from any one of these simple cells. By alternating layers of complex cells, which effectively blurred small differences in identified features, with layers of simple cell feature extractors, simple cells at higher layers learned to accommodate distortions and translations in the features to which they were responsive, resulting in the shift-invariance characteristic of CNNs.

The first CNN was introduced by LeCun et al. (1998). LeNet-5, a 7-layer CNN trained by backpropagation, extracted features in "convolutional layers", corresponding to the simple cell layers of the neocognitron, while "subsampling layers", corresponding to the complex cell layers of the neocognitron, blurred local pattern distortions. To combine features learned in the lower layers, the last convolutional layer was fully connected to the output layer, essentially uniting the neocognitron with an MLP.

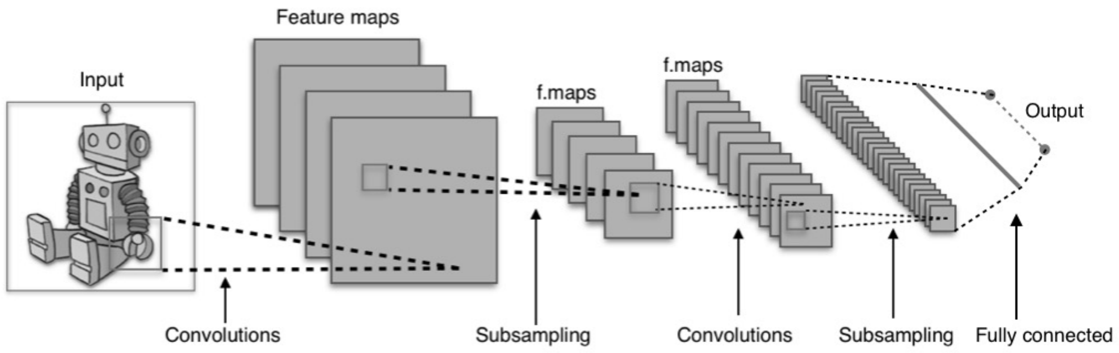


Fig. 2.3.3.2. A convolutional neural network

A filter, represented here by a small square, moves over the pixels in the input image to create feature maps, which are subsampled by pooling layers into more feature maps. The process repeats until the data exits the fully connected output layer. (Aphex34, 2015)

In a typical CNN (depicted in Fig. 2.3.3.2.), a matrix of weights, also known as a "filter" or "kernel", is convolved over an input image, represented as a larger matrix of pixel values. As the filter moves over the input, covering each pixel at least once, the dot product of the pixel values and the weights are translated into "feature maps", matrixes that capture regions with salient features. Different filters extract different features relevant for accurate network output. Training the CNN, usually via backpropagation, updates the filter weight values, which improves the network's identification of regions that are significant for feature extraction.

Optionally following the convolutional layers in a CNN, subsampling, or "pooling", layers perform a similar process to reduce the spatial size of the input. A favored algorithm for these layers is "max pooling", which uses the max value within the filter matrix as the value for the corresponding position in the feature map.

The last layer of a CNN is generally fully-connected, and forces the data into the number of output classes desired.

2.3.3.3. Recurrent Neural Network (RNN)

Unlike feedforward networks such as MLPs and CNNs, where input data moves through the network in only one direction, RNNs rely on directed cycles to achieve data persistence and process arbitrary input sequences.

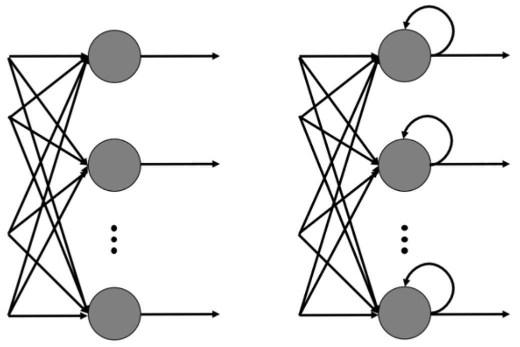


Fig. 2.3.3.3.1. Recurrent vs. feedforward network

Left: A hidden layer in a feedforward network; Right: A recurrent hidden layer, where the output of a node is recycled as its input.

Long Short-Term Memory networks (LSTMs) are a particularly notable type of RNN whose ability to form long-term dependencies has led to field-defining applications in speech recognition and natural language processing. An LSTM unit typically contains a "memory" cell, which holds a value for an arbitrary period of time, and a series of "gates", which manage state. These gates are similar to traditional neurons and control whether or not, and how much, information should be allowed into and out of the cell.

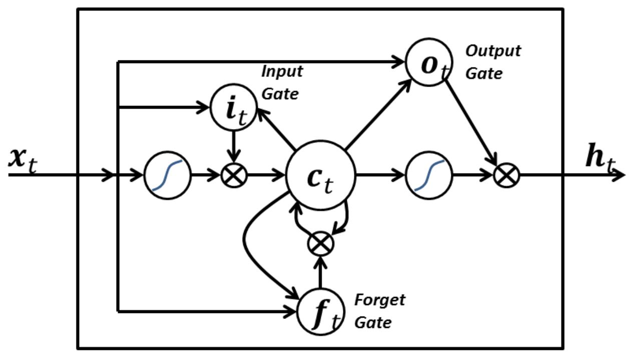


Fig. 2.3.3.3.2. Long Short-Term Memory

A "peephole" LSTM unit showing the flow of data through input, output, and forget gates and a central memory cell (BiObserver, 2015).

In a "peephole" LSTM (Fig. 2.3.3.3.2.), an input gate *i* controls new values entering a memory cell *c*, a forget gate *f* determines how much value remains in the cell, and the output gate *o* calculates the output of the unit based on the cell activation. Their respective gate activations i_t, o_t, and f_t at a given time step *t* are based on c_{t-1}. This gated recurrency of an LSTM allows it to maintain a representation over time of its input sequences without interference from novel stimuli.

2.4. Biomimetic ANNs

Computational neural network implementations can be considered on a spectrum between two poles: 1) systems that maximize machine learning performance without particular regard for biological plausibility, and 2) biologically faithful interpretations focused on providing neurobiological insight, with varying degrees of direct applicability to machine learning.

The majority of commercial neural network architectures are positioned firmly into the first category. Research and Development divisions may take inspiration from biological systems, but trading algorithm designers and self-driving car manufacturers are governed primarily by practical application. In the final implementation, minimal or no consideration is given to preserving correlation between program structure and brain architecture, or program function and brain activity.

The second category is largely composed of models of neurosystems developed for academic and medical purposes. de Garis, Chen, Goertzel, and Ruiting (2010) provide a comprehensive review of several large-scale brain simulations, including the Blue Brain Project (Markram, 2006), that display the complex dynamics of brain regions but do not yield intelligent behaviors. Much of the work performed under the recently declared International Brain Initiative (Human Brain Project, 2017) also falls primarily into this category; a major goal of its founding member institutions is to directly measure and map electrical and chemical activity to generate neural circuit diagrams that can be compiled into a cohesive, definitive model of the brain (Amunts et al., 2016; National Institutes of Health, 2014).

As ReCoNs are a cortically inspired solution to a machine learning problem, comparable approaches are found along the spectrum between these two classes: biomimetic learning systems based on cortical anatomy and physiology.

2.4.1. Current approaches to biomimetic learning

In 2010, a survey of then-notable biologically inspired cognitive architectures[1] concluded that "it [was] not yet possible to tell whether emulating the brain on the architectural level is going to be enough to allow rough emulation of brain function"

[1] This survey incidentally includes MicroPsi, the cognitive architecture used as a foundation for the pilot ReCoN implementation discussed in Bach and Herger (2015) and later in this thesis.

(Goertzel, Lian, Arel, De Garis & Chen, 2010, p. 30). Subsequent research has contributed to an increasing body of evidence suggesting that this is, in fact, likely (Eliasmith et al., 2012; Reimann et al., 2017).

Given that brain is our only present example of general intelligence, and considering that the first successes in machine learning directly resulted from attempts to simulate its structure and function, it appears sensible to pursue AGI by continuing these efforts. This sentiment was echoed in 2014 by prominent computer scientist Yoshua Bengio, who noted that "[a]s far as I know we haven't found AI yet, so any inspiration we're getting from biology is worth taking in and…see[ing] if there's some computational or mathematical principles that we can use" (Hernandez, 2015).

This section surveys current approaches to machine learning that intentionally attempt to recreate biological neural function with biomimetic artificial neural networks. It is important to note that these approaches are by no means mutually exclusive, and can be considered as potentially complementary angles from which to unpack the intricacies of the brain and cognition.

2.4.1.1. Reverse-engineering the cortex

A founding member of the newly formed International Brain Initiative, the U.S.-based massively multi-institutional Brain Research through Advancing Innovative Neurotechnologies (BRAIN) initiative seeks to develop new technologies to map the brain at multiple scales and expose the relationship between brain function and behavior (BRAIN Initiative, n.d.). Although the majority of this work appears to fall outside the narrowly defined scope of biomimetic learning systems, one subprogram in particular is relevant to consider.

In 2016, under the heading of the BRAIN initiative, the Intelligence Advanced Research Projects Agency (IARPA) announced that it would be delegating $100 million to the Machine Intelligence from Cortical Networks program (MICrONS), which intends to reverse-engineer machine learning algorithms from a cubic millimeter of rodent visual cortex (Cepelewicz, 2016). Three independent teams from Harvard University, Carnegie Mellon University, and Baylor College of Medicine were selected to pursue this line of inquiry, each using a different approach: translating electron microscopy analysis into 3D mapping of neuronal connections; using genetic markers to map neural circuitry; and testing hypotheses about the development of communication between different neural circuit components, respectively (Singer, 2016).

2.4.1.2. Spiking Neural Network (SSN)

Certain biomimetic machine learning systems, SSNs, have focused on emulating the spiking nature of biological neural networks. Biological neurons have a "membrane potential", the voltage difference between the neuron's interior and its exterior environment that gates its inclination to fire, and transmit information via short voltage increases known as "spikes". Unlike MLP neurons, which propagate analog activation at each network step, SSN neurons have an internal threshold representing membrane potential that is elevated by discrete incoming spikes, and only fire when the threshold value is reached. The time elapsed between spikes, the frequency of the spikes, and other spike-related variables offer additional dimensions for encoding information in the network. SSNs are thereby able to replace biologically implausible backpropagation with biologically plausible spike-timing dependent plasticity, which locally adjusts link weights based on the relative timing between a neuron receiving activation and firing (Diehl & Cook, 2015; Taherkhani, Belatreche, Li & Maguire, 2014).

SSNs have exhibited sufficiently significant neurobiological parallels as to be used to replicate *in vitro* features in when studying biological neural networks (Brette et al., 2007; Maheswaranathan, Ferrari, VanDongen & Henriquez, 2012). This correlation has also been displayed in task execution; Spaun, a large-scale SSN with 2.5 million spiking neurons, demonstrated a range of behaviors associated with cognitive functions (Eliasmith et al., 2012), and an SSN-based virtual insect successfully learned to avoid environmental obstacles to reach a target location without prior priming for navigation (Zhang, Xu, Henriquez & Ferrari, 2013).

Despite these achievements, and despite the assertion that temporal coding in SSNs may enable a single spiking neuron to achieve the same compute as hundreds of hidden neurons in a sigmoidal network (Maass, 1997), SSNs have to date found limited application in machine learning. This can be traced to their relative complexity in implementation, large processing demands for biologically plausible models, and lower performance on benchmarks compared to other types of ANNs (Sengupta, Ye, Wang, Liu & Roy, 2018). However, recent advances in SSN optimization may reignite interest in their machine learning applications (Sengupta et al., 2018), and the event-driven nature of SNNs makes them an appealing candidate for neuromorphic hardware (Merolla et al, 2014; Lee, Delbruck & Pfeiffer, 2016), discussed in Section 2.4.1.5.

2.4.1.3. Hierarchical Temporal Memory (HTM)

HTM is a theory of intelligence developed by Palm Pilot inventor and Numenta co-founder Jeff Hawkins that serves as the basis for machine learning technology. As in spiking models, HTM is similarly fundamentally time-based: it parses a stream of data to discover patterns and anomalies, and uses hierarchical caching of these learned sequences to generate future predictions.

In a HTM system, unlabeled data flows through an encoder into a spatial pooler, which uses a "winners take all" approach to map only the top 2% of neurons maximally activated by the input to a Sparse Distributed Representation (SDR), an array of bits where each bit index has a specific semantic meaning encoding a feature of the input data. If the same bit is active in two SDRs, it indicates semantic similarity in the corresponding inputs. In an approximation of unsupervised Hebbian learning in the brain, when a bit becomes active, it forms connections to neighboring bits that were active in the previous time step. Upon recurrence of the sequential activation pattern, the connections are strengthened, otherwise the connections are weakened and eventually dropped. The system thereby continuously learns to predict the next sequence by tuning a "connection permanence" scalar that indicates whether a particular connection is closer to becoming permanent (1) or being dropped (0) (Numenta, 2012).

To more accurately depict the modularity of the cortex, which is discussed further in Section 3.1., the HTM model was recently extended into laterally connected vertical layers of nodes, an endeavor to approximate pyramidal neurons in cortical columns (Hawkins, Ahmad & Cui, 2017).

When evaluated against four other popular techniques in its specialty arena, continuous sequence learning, HTM achieved comparable prediction accuracy to state-of-the-art systems, demonstrated fast recovery after sequence changes, and did so without requiring either periodic retraining or external input to determine the higher-order data structure, unlike the other systems against which it was matched (Colyer 2017; Cui, Ahmad & Hawkins, 2016). As of this writing, HTM has not set any machine learning benchmarks, but remains potentially promising in the eyes of those who anticipate that AGI must necessarily emulate cortical architecture; in April 2015, IBM dedicated a research group of around 100 people, the “Cortical Learning Center”, to further explore HTM algorithms (Simonite, 2015).

2.4.1.4. Capsule networks

In 2011, machine learning pioneer Geoffrey Hinton introduced "capsules", groups of neurons whose outputs represent different instantiation parameters of the same entity (Hinton, Krizhevsky & Wang, 2011). Capsule design was inspired by techniques in computer vision and graphics rendering, which, in contrast to the scalar output of traditional feature detecting neurons, use vectors of outputs to represent learned features, and can thereby explicitly encode "pose", the spatial relationship between features.

While traditional CNNs achieve shift-invariance through sub-sampling, which summarizes activity across neurons and discards these spatial relationships between higher-order features, each capsule in a capsule network learns to recognize multiple orientations of a single visual entity. The output for each capsule is a vector of parameters specifying the variation of a detected entity from its implicit canonical representation of that entity, the length of which represents a probability prediction that the entity is present in the capsule's domain. Capsule networks are therefore able to represent exact spatial relationships of higher-order features and learn to recognize wholes as sums of their parts.

A "dynamic routing" algorithm is used to determine the most consistent interpretation of information between competing capsules. An activated capsule sends output to all possible parent capsules in the next layer, and calculates for each a "prediction vector", the product of its own output and a weight matrix. A large product induces top-down feedback that increases the contribution the capsule makes to that parent, while decreasing it for the others. (Sabour, Frosst & Hinton, 2017)

As noted in Sabour, Frosst & Hinton (2017), capsule networks and dynamic routing have some claim to biological plausibility based on models of pattern recognition in the visual cortex. Specifically, the preservation of structural relationships between input features by encapsulation, the use of capsules as the fundamental units of each network layer, and the dynamic routing between capsules are reminiscent of the modularity and activation patterns found in the cortex, discussed further in Section 3.1.

2.4.1.5. Neuromorphic hardware

The brain is generally estimated to operate on 12 W to 20 W (Clancy, 2017; Jabr, 2012), whereas the IBM Sequoia supercomputer has less processing power and consumes 7.9 MW (Docksai, 2017), and in 2005, one simulated second of a detailed 100 billion neuron model of the brain took 50 days on a Beowulf cluster of 27 3GHz processors (Izhikevich, 2005). The brain's computing power, coupled with its energy efficiency, has led to growing attention on neuromorphic hardware, biologically inspired devices capable of improving performance when running large scale ANNs and other computationally demanding programs. Some neuromorphic architectures also seek to better approximate the qualities displayed in biological systems by incorporating features not found in traditional processing units, such as chaotic elements proposed as being integral to the complex dynamics demonstrated by the brain (Kumar, Strachan & Williams, 2017).

Neuromorphic hardware implementations can be analog, digital, or hybrid platforms. The memristor, a circuit element that emulates the plasticity and timing dependence of synapses in the brain (Strukov, Snider, Stewart & Williams, 2008), is "ubiquitous" in neuromorphic systems due to their energy efficiency and strong claims to biological plausibility (Schuman et al., 2017). Two particularly famous neuromorphic architectures, the University of Manchester's SpiNNaker and IBM's TrueNorth, are both digital, massively parallel systems that do not use memristors: SpiNNaker consists of approximately 57,000 nodes of 18 ARM9 cores each that can model up to a billion neurons and a trillion synapses in real time (Painkras et al., 2013), and TrueNorth is a 64 μW 5.4-billion-transistor silicon chip with 4096 neurosynaptic cores each simulating 256 programmable neurons (Merolla et al., 2014).

A self-organized mesh of silver nanowires, 2 square millimeters in size, at UCLA is perhaps an especially unusual neuromorphic device. Formed by pouring silver nitrate onto tiny copper spheres, the resulting nanowire mesh was exposed to sulfur gas to induce the formation of silver sulfide. When voltage is applied, silver ions are pushed out of the silver sulfide and form silver filaments that act as switches for the current; reversing the current shrinks the filaments, turning off the switch (von Bubnoff, 2017). This mesh has demonstrated emergent logical behaviors, such as prediction of statistical trends after training, and is inherently fast, with tens of thousands of state changes per second (von Bubnoff, 2017; Scharnhorst, Woods, Teuscher, Stieg & Gimzewski, 2017).

Chapter 3.

Request Confirmation Networks

ReCoNs are hierarchical dynamic ANNs designed to simulate the representation and execution of sensorimotor scripts without the need for a central control architecture, a function that might be realized biologically by the modularity of the cerebral cortex. ReCoNs were first introduced in Bach and Herger (2015), which provides a complete formal specification and results from a pilot implementation.

3.1. The neurological inspiration for ReCoNs

The outer layer and largest region of the human cerebral cortex, the neocortex, is thought to consist of over 180 distinct areas (Glasser et al., 2016), functionally defined regions composed of millions of neurons organized into basic units of approximately 80 - 100 neurons each. These units, known as cortical minicolumns, average 30 - 60 µm in diameter (Buxhoeveden & Casanova, 2002; Jones, 2000) and extend through the six main layers of the cortical sheet.

Neurons within a minicolumn are responsive to similar object features (Horton & Adams, 2005), have common outputs, and are vertically interconnected. The overlapping processing chains between neurons in a minicolumn form circuits that map arrays of inputs to arrays of outputs (Mountcastle, 1997), prompting assertions from neuroscientists that they "may well constitute a fundamental computational unit of the cerebral cortex" (Cruz et al., 2005, p. 322).

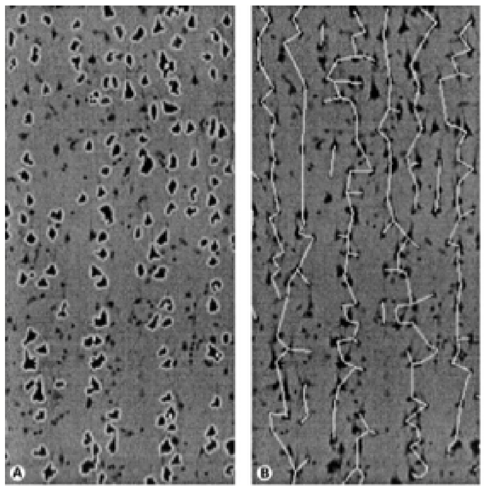


Fig. 3.1. Cortical minicolumns

Left: Large pyramidal neurons (outlined in white) in the cortex; Right: Euclidean spanning trees (white) denoting minicolumns composed of these neurons (Buxhoeveden, Switala, Litaker, Roy & Casanova, 2001)

Minicolumns form maps by default, but can also be recruited into sequences and scripts (Cosentino, Chute, Libon, Moore & Grossman, 2006), and learn how to bind to each other to form dynamic processing hierarchies. They are often stacked across areas into receptive fields, where individual units receive activation from a corresponding group of units in neighboring areas. Various lines of research suggest that the inhibition of this activation is crucial to understanding the function of a minicolumn (Buxhoeveden & Casanova, 2002): whereas lateral excitation between minicolumns has been shown to cause neurons to develop related afferent connections, reciprocal lateral inhibition produces dissimilar afferent connections, thereby causing neighboring minicolumns to encode for different receptive fields (Favorov & Kelly, 1994).

A long-held theory of biological learning proposes that the primary role of the cortex is unsupervised learning through prediction (Yger & Harris, 2013). The balance of excitation and inhibition between minicolumns results not only in orderly yet diverse receptive fields, but also in "a variety of stimulus feature-extracting properties" (Favorov & Kelly, 1994, p. 408) that would seem consistent with this interpretation. A 2004 study of a model of inhibitorily coupled minicolumns further appears to support this premise by demonstrating that the minicolumns were able to self-organize via Hebbian plasticity into selective receptive fields that allowed them to serve as classifiers for input patterns (Lüke & von der Malsburg, 2004).

The body of literature supporting the role of minicolumns as a fundamental information processing module has made them a popular choice for simulation in biologically inspired ANN architectures (for examples, see Azam, 2000; Johansson & Lansner, 2007; Koene & Hasselmo, 2005; Lüke & von der Malsburg, 2004; Martinet, Sheynikhovich, Benchenane & Arleo, 2011; Richert, Fisher, Piekniewski, Izhikevich & Hylton, 2016; Sandberg, Lansner, Petersson & Ekeberg, 2002). Of the approaches to ANNs previously discussed in this thesis, as mentioned in Section 2.4.1., two explicitly purport to model cortical columns: HTM (Section 2.4.1.3.) and Hinton's capsule networks (Section 2.4.1.4.).

3.2. ReCoN Structure

ReCoNs are auto-executable networks of stateful nodes with typed links that perform neural computations and script execution by passing activation along the links in a controlled hierarchical sequence termed "request confirmation". They are implemented within the MicroPsi formalism (Section 3.3.1).

3.2.1. Nodes and links

According to Lamb (2000), "if a neuron is analogous to a logic gate, a cortical (mini)column is more analogous to a small subroutine". In this sense, a ReCoN node can be considered as an approximation of a cortical minicolumn, representing a subroutine within the ReCoN hierarchy.

Each ReCoN node is a state machine with one of eight states, {*inactive, requested, active, suppressed, waiting, true, confirmed, failed*}. The state of a node is determined by its activation level, which results from the activation of connected nodes in the preceding time step. This internode state dependency allows for the simulation of excitatory and inhibitory relationships, such as those of neighboring minicolumns.

Activation	State	Meaning
< 0	*failed*	will remain until requesting ends
< 0.01	*inactive*	will change to prepared when requested
< 0.3	*preparing*	inhibits neighbors, changes to suppressed
< 0.5	*suppressed*	inhibits neighbors, changes to requesting when no longer inhibited
< 0.7	*requesting*	starts requesting, changes to pending
< 1	*pending*	continues requesting, will either change to confirmed or failed
>=1	*confirmed*	will remain until requesting ends

Table 3.2.1. ReCoN node states and corresponding activation levels

Nodes are connected to each other by weighted links. They receive input from incoming links via "slots" and transmit output on outgoing links via "gates". The ReCoN nodes used in the implementation discussed in Chapter 4 have five input slots and five output gates, one of each type *gen* (general), *por* (left), *ret* (right), *sub* (below), and *sur* (above). Each link between ReCoN nodes is either of a type *sub/sur,* which connects a parent node to a child node, or *por/ret*, which connects a predecessor node to a successor node and indicates a lateral relationship. Gates of type *gen*, representing a general association, are available to connect ReCoN nodes to nodes of other types.

3.2.2. Script execution

Section 2.2. outlined the spreading of activation through cortical areas in response to visual input, resulting in recognition of the viewed object. This sequence can be considered in terms of bottom-up execution, originating from environmental stimuli and ending in the activation of the symbolic internal representation of the input. Conversely, a similar sequence could be executed top-down by directly activating the upper hierarchy of the network encoding the object, thereby calling the symbolic representation of the object to mind without extending all the way through the primary visual cortex. This top-down activation has been implicated as the underlying process in voluntary memory recall (Miyashita & Hayashi, 2000; Tomita, Ohbayashi, Nakahara, Hasegawa & Miyashita, 1999) and dream imagery, which involves the higher processing networks for visual memory but not the low-level perceptual networks (Solms, 1997).

Sensory circuits at early stages of processing, like those in the primary visual cortex, are shared by pathways that govern both action and perception (Goodale &

Milner, 2013). This allows them to be harnessed into schemas, higher-order neuro-symbolic complexes composed of many subnetworks of varying function and complexity. Schemas represent objects or events and organize categories of information and relationships between them (DiMaggio, 1997); scripts are a category of sensorimotor schema that encode key sequences of events and the relevant actions necessary to fulfil behavioral routines and activities (Barnett et al., 2007).

The deliberate top-down initiation of a script, as in the intentional moving of an arm or imagining of an object, has been attributed to activity in the prefrontal cortex (Deiber et al., 1991; Frith, Friston, Liddle & Frackowiak, 1991), an area associated with goal-directed behavioral planning and task management (Koechlin, Basso, Pietrini, Panzer & Grafman, 1999; Tanji & Hoshi, 2001). To execute a cognitive process or an action, activation flows from its initial stimulation in the prefrontal cortex through the relevant schematic components, continuing either until the objective has been successfully achieved, or until the sequence is interrupted or fails.

ReCoNs offer a possible model for how these schemas and sensorimotor scripts are represented and executed in the cortex. The execution of a ReCoN script begins with a *request* signal to its root node, signified by activating its *sub* slot. The script execution tests a hypothesis, represented by the root node and defined in the script itself, which will return a binary truth value.

Parent nodes request confirmation from their child nodes via *sub* links, and receive in return a *wait* signal, followed by either confirmation or failure of the request via *sur* links. A successor node that requires confirmation from predecessor nodes before executing its own subsequence is prevented from becoming prematurely active by an *inhibit request* signal received via *por* links until that confirmation is available, and will send back an *inhibit confirm* signal via *ret* links while its sequence executes, indicating to its predecessors that calculation is in progress.

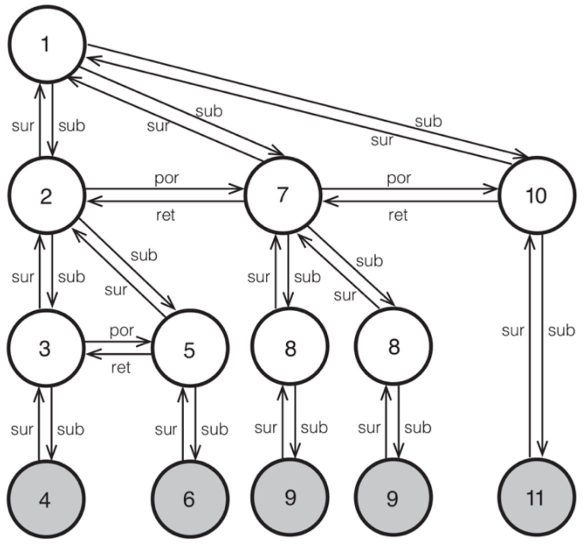


Fig. 3.2.4. Example ReCoN script execution

Activation of the root node (1) sends requests for computation top-down through the network via sub links to check the validity of its hypothesis or perform actions. Confirmation or failure percolates bottom-up via sur links. (Bach & Herger, 2015)

Calculation is finished when the last element of the sequence changes its state to *confirmed* and the confirmation propagates back to the root node, or upon interruption or failure.

The reciprocal inhibition of ReCoN nodes provides discrete on-off control that acts as a decentralized gating mechanism. Inhibition has been shown to play a similar role in biology, as noted in Section 3.1., as well as in the significance of the inhibitory outputs of the basal ganglia in gating the thalamus (Goldberg, Farries & Fee, 2013) or in the instantiation of pathway-specific gating (Yang, Murray & Wang, 2016).

3.3. Prior ReCoN implementation

A pilot ReCoN implementation is outlined in Bach and Herger (2015), which describes results obtained from allowing a MicroPsi agent to explore a Minecraft world, using ReCoNs to recognize previously visited locations.

3.3.1. MicroPsi

MicroPsi (Bach, 2003) is a cognitive architecture designed to provide a framework in which to run and test ANN-based agents. It extends Dietrich Dörner’s PSI theory, which models the mind as an information processing architecture and sets computational formalisms for the interaction between cognitive, motivational, and emotional processes (Dörner & Güss, 2013). The current iteration of MicroPsi, MicroPsi2, offers a graphical editor and runtime system for ANNs and agent environments.[2]

An overview of the architecture of a typical MicroPsi agent is displayed in Figure 3.3.1. Perceptual sensors provide environmental input to the agent, while other sensors monitor simulated "urges", like "hunger" and "tiredness", and affect the agent's motivation, which governs the priority it places on potential near-future actions.

Short-term memory provides a local perceptual space and associated representations, whereas long-term memory stores knowledge acquired from the agent's history of interacting with its environment. The exchange between these two forms of memory is managed by a memory maintenance module.

[2] The MicroPsi Editor Shell (MESH) is discussed in Section 4.3. of this thesis.

Action is decided and executed within the behavior script space, which encompasses the "internal behaviors" analogous to higher cognitive processes. A meta-management system allocates processing resources to the internal behavior processes and memory maintenance based on the current situation and state of the agent.

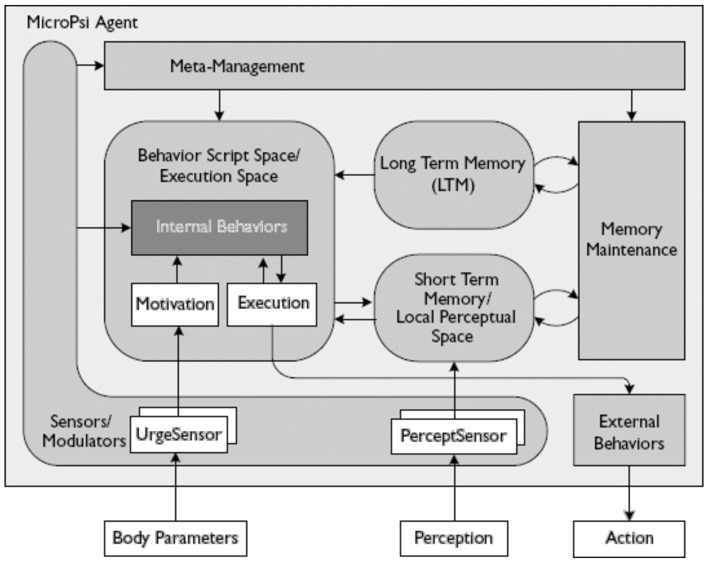


Fig. 3.3.1. MicroPsi agent architecture

An agent receives external and internal stimuli from perceptual and body urge sensors, respectively. Actions are based on previously acquired knowledge and governed by motivation. Memory maintenance manages the exchange between long- and short-term memory, and meta-management coordinates resources between subsystems. (Bach, 2009)

3.3.2. Active perception with ReCoNs

In the active perception task discussed in Bach and Herger (2015), learned features of the Minecraft environment encountered by a MicroPsi agent were encoded by ReCoN link weights. Hypotheses represented by the ReCoN root nodes were based on an autoencoder, an ANN whose output is a reconstruction of its input, that captured the features of an environmental scene; the original feature input and the autoencoder output are shown in Figure 3.3.2. Higher-level nodes encode object representations and geometric relations between objects, whereas nodes further down the hierarchy encode for lower-level compositional visual features.

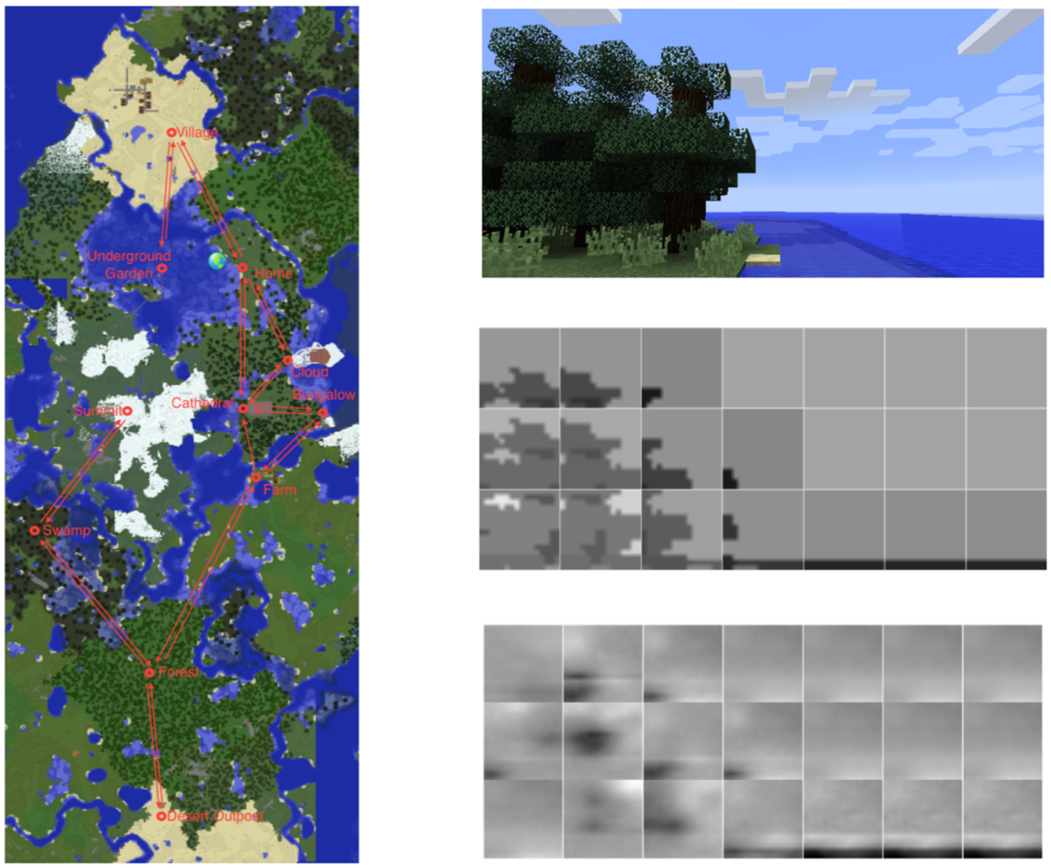


Fig. 3.3.2. Action space and input sample of a ReCoN-based agent in Minecraft

Left column: The red line shows the path of the agent through the Minecraft environment; Right column: Training data from the Minecraft client as seen in a graphical user interface (top), visual input to the agent (center), and features learnt by the agent (bottom) (Bach & Herger, 2015)

When an agent traveled into a new space, it would check its existing hypotheses to see if it recognized the environment as a specific combination of stored features. If the agent had not previously encountered that space, all of its existing hypotheses failed, and it would form a new model built from a combination of the sub-hypotheses that were confirmed by local features of that environment.

The agent was tested by allowing it to freely tour a fixed set of Minecraft locations. Once the agent had formed sufficient internal representations to be able to successfully identify all locations, it stopped generating new hypotheses. This indicated that the ReCoNs had worked as anticipated and were able to represent all features necessary for the agent to recognize any previously encountered environment.

3.3.3. Revisiting ReCoNs

Although results from the active perception ReCoN implementation appeared promising, no further work was performed. The source code for the implementation has since either been lost or deleted.

The ReCoNs discussed in the following chapters of this thesis are a reconstruction of the original networks. They were built based on the details available in Bach and Herger (2015), instructions provided by Joscha Bach, and a code extract discovered in the MicroPsi2 MESH application that specifies the ReCoN node activation thresholds and corresponding states defined in Table 3.2.1. They are implemented within a general framework for neural networks built from scratch to be roughly compatible with MicroPsi2 in order to preserve the option for future extension into that architecture.

A stand-alone program designed to serve as an illustrative proof-of-concept for the recreated ReCoNs is detailed in Chapter 4.

Chapter 4.

Function Approximator

The Function Approximator was created as a basic example of ReCoN operation and proof of concept. The system accepts images of handwritten algebraic functions as input and outputs the function value. It consists of a function parser to segment the handwritten function into its individual symbols, an *n*-layer perceptron pre-trained on handwritten digits 0-9 and the mathematical operators +, -, ×, and ÷ to classify the symbols, a ReCoN that is custom-built based on the array of recognized symbols to represent the identified function, and a stacked calculator to execute the function.

4.1. Biological function approximation

The ability to approximate numerosity, the magnitude of a stimulus, and to perform elementary arithmetic, is found in both humans and animals (Woodruff & Premack, 1981). Humans also associate discrete numeric values with perceptual representations, such as the verbal "nine" and visual Arabic numeral "9", and lab-trained rhesus macaques have demonstrated the ability to do the same, even completing rudimentary addition tasks using visual symbols (Livingstone et al., 2014).

The neurobiological foundations of numerical competence are distributed across a network in the parietal and frontal lobes (Nieder, 2016). Numerous studies have shown that the intraparietal sulcus (IPS), a region that runs horizontally across the surface of the parietal lobe, plays a principal role in both non-symbolic and symbolic numerical processing, including arithmetic approximation with Arabic numerals (Cantlon, Brannon, Carter & Pelphrey, 2006; Dehaene, Spelke, Pinel, Stanescu & Tsivkin, 1999). Monkeys trained to associate numerosity with Arabic numerals and young children developing familiarity with number symbols show elevated activity in the prefrontal cortex (PFC) compared to the IPS when processing symbolic numbers; this effect fades over time as children gain numeric fluency and may signify the PFC mapping non-symbolic to sub-symbolic representations (Nieder & Dehaene, 2009). As IPS neurons respond earlier in hierarchical number processing than those in the PFC, and, unlike PFC neurons, are sensitive to perceptive modality, it is thought that the IPS may extract quantitative information before it is processed by the PFC in a goal-directed manner (Nieder, 2016).

The PFC, parietal, premotor, posterior temporal, and subcortical areas have all been implicated in arithmetic calculation, differing based on calculation task (Arsalidou & Taylor, 2011). Recent research has indicated that in simple digit addition, digit comprehension occurs in the left superior temporal area in the temporal lobe, is executed in the left inferior parietal area, located directly below the IPS, followed by recognition in the corresponding regions in the right hemisphere (Iijima & Nishitani, 2017).

Given the difficulty of disentangling the exact neurobiological mechanisms of variably encoded numeric representation during calculation from the instrumental processes required for both arithmetic and non-arithmetic operations (Gruber, Indefrey, Steinmetz & Kleinschmidt, 2001), there is currently no consensus model of numeric representation in calculative tasks (Eger, 2016). The ReCoN structures generated by the Function Approximator are therefore not intended to be a precise depiction of neurobiology, but are rather proposed as a simple and generally translatable abstraction of hierarchical excitation and inhibition between cortical modules during neuro-symbolic script execution.

4.2. System Overview

A function image is first parsed into separate images of its digits and mathematical operators. These subcomponent images are prepped according to MNIST preprocessing directives (LeCun, 1998) and fed into an MLP of *n* layers, where *n* >= 2. The MLP is a symbolic classifier pre-trained on a combination of MNIST data and the algebraic operators +, -, ×, and ÷, adapted from a Kaggle handwritten math symbols dataset (Nano, 2016). The MLP output is a prediction of the symbol depicted in each image; these predictions are gathered in an array representing the original function.

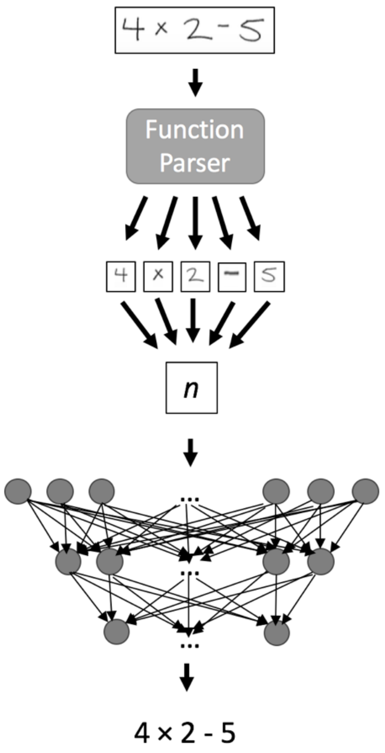


Fig. 4.2.1. Sequence diagram of function parsing and classification

An image of a handwritten function (4×2-5), is parsed into five 28x28 pixel images depicting its respective subcomponent symbols. Each of these five images is fed into a pre-trained MLP classifier, which outputs a classification prediction for its symbol.

The array of symbols identified by the MLP is used to build a ReCoN that embodies the predicted function in its topology, with the root node symbolizing the numeric function value. Activation of a ReCoN root node initiates the subroutine represented by the ReCoN. In the case of the Function Approximator, the subroutine is the execution of the predicted function and comparison of the resulting numeric value against the correct value label of the input handwritten function. The ReCoN uses a stack-like data object, referred to herein as "the stack", to perform the mathematical operations of the function and return a final value.

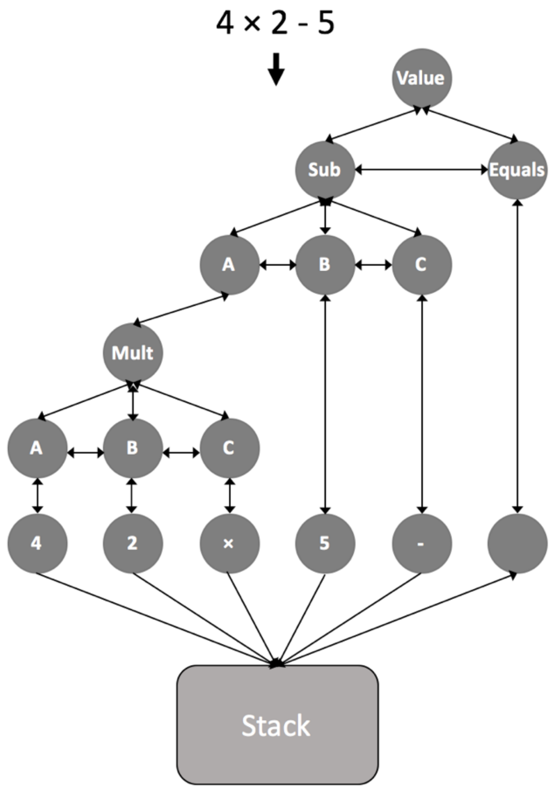


Fig. 4.2.2.	Diagram of ReCoN and stack

The array of digits and algebraic operators output by the MLP classifier is used to generate a ReCoN that represents and executes the predicted function by pushing and pulling values to and from a stack-like data object.

4.3. Implementation

The Function Approximator is a "recipe", or predefined program, that runs on a stateful neural network architecture. This architecture provides the framework for both the MLP and ReCoN, and can be extended into any neural network configuration. A full specification of this architecture is included as Appendix A.

4.3.1. Datasets

Two datasets, MNIST and a Kaggle handwritten math symbols dataset (Nano, 2016), were combined to create the training data for the MLP classifier. The handwritten functions used as input data for the Function Approximator can be adapted from any source, provided that they follow the parameters defined here.

4.3.1.1. MNIST

The MNIST database is commonly used for training image processing systems. It consists of 60,000 training images and 10,000 testing images depicting handwritten greyscale digits 0-9, as well as respective labels for each set.

MNIST images are normalized to fit within a 20x20 pixel box, which is then centered inside a 28x28 pixel canvas. Pixel values are 0-255, with 0 indicating background (white) and 255 indicating foreground (black). The MNIST images and their one-hot encoded labels are packaged as multidimensional matrices in IDX file format.

Fig. 4.3.1.1. Sample MNIST digits

Four images from MNIST database. From left to right: 5, 0, 4, 1. (TensorFlow, 2015)

4.3.1.2. Handwritten math symbols dataset

The original Kaggle database (Nano, 2016) from which the mathematical operators used here were derived consists of sets of 45x45 pixel images depicting 78 different symbols commonly used in mathematical and scientific publications. Only the sets for +, -, ×, and ÷ are relevant to this implementation.

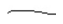
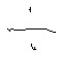
Name	Symbol	Number of files	Sample image
Addition	+	25,113	
Subtraction	-	33,998	
Multiplication	×	3,252	
Division	÷	764	

Table 4.3.1.2. Original math symbol sets from Nano (2016)

Given that all four algebraic operators can be expected to appear in any random function with equal probability, it was necessary to trim the larger sets significantly so as not to unduly bias the MLP toward symbols that appeared more often in its training data, while also maintaining a balanced ratio of the total number of mathematical operators to the total number of MNIST images. Images visually determined to not resemble their respective symbols were manually removed as well.

All images were size normalized and centered to match the MNIST dataset. The image filenames are prefixed with the symbol depicted in the image, which is used to generate a one-hot encoded label. As with MNIST, the images and labels are packaged as multidimensional arrays.

Initial results for a combined MNIST and math symbols training set (with 20% of the overall symbols retained for testing) of 4606 addition symbols, 6363 subtraction symbols, 2,444 multiplication symbols and 567 division symbols indicated a strong bias against the underrepresented division symbol, so an additional 240 division symbols were handwritten, scanned, and included, which appeared to slightly improve performance over most training runs.

4.3.1.3. Combined MNIST and math symbols

A single dataset containing both the MNIST and math symbol images is generated by zipping each set with their respective labels, combining the two zipped sets into a single array, randomly shuffling the array contents, and unzipping the shuffled array into an images matrix and a labels matrix.

4.3.1.4. Handwritten functions

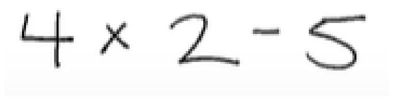


Fig. 4.3.1.4. Sample handwritten function

The functions used as input for testing the Function Approximator were handwritten by members of the Harvard Program for Evolutionary Dynamics on white paper with black ink, scanned, and uploaded as individual PNG images.

The image filenames serve as the function labels and are therefore manually named to reflect the function and its evaluation; for example, Figure 4.3.1.4 would be named "4*2-5=3.png". This is done so that, after the function is parsed into its individual symbols, each symbol image file can be automatically labeled with the correct symbol as its filename, and the final output of the Function Approximator can be compared against the correct function value.

4.3.2. Function Parser

The Python image processing library scikit-image is used to divide a PNG image of a handwritten function into PNG images of its component symbols.

The image is converted from RGB to greyscale, then thresholded using Otsu's algorithm, which calculates the optimal division between a binary distribution of foreground and background pixel values such that the internal variance of both classes is minimal. The image is masked using the calculated threshold, and distinct continuous foreground areas are identified.

Each foreground area is bounded and cropped, employing sufficient padding to ensure that the non-continuous dots in the division symbols are not identified as independent components and bounded separately. The cropped images are saved in a folder named for the function, as derived from the function filename. In order to serve as labels and conserve the left-right order of appearance of the symbols in the function, image filenames are in the form "*index* | *identity*.png", where *index* is the position of the symbol in the function and *identity* is the symbol itself. For example, the filename for the first of the five images generated from Figure 4.3.1.4 would be "0 | 4.png".

To prepare the images according to MNIST preprocessing directives (LeCun, 1998), they are first trimmed of all excess whitespace. The trimmed symbols are then pasted onto the center of a square background; if a symbol's width *w* exceeds its height *h*, the canvas dimensions are *w*x*w*, whereas a symbol taller than it is wide is pasted on a canvas of dimension *h*x*h*. Finally, the square image is resized to 20x20 pixels and centered on a blank canvas on 28x28 pixels.

4.3.3. MLP

The first of the two neural networks employed by the Function Approximator is a fully-connected *n*-layer feed-forward perceptron, where *n* >= 2, layer_0 has 784 nodes, layer_{n-1} has 14 output nodes, and any optional hidden layers can be of variable size. As networks increase in size and prediction accuracy, there is a corresponding increase in processing demands and execution time. All MLP nodes are of the type "register", which are defined as having one input slot of type *gen* and one output gate of type *gen*.

<u>4.3.3.1. Pre-trained MLPs</u>

The Function Approximator offers the choice between three pre-trained MLPs of sizes [784, 14], [784, 60, 14], and [784, 240, 60, 14]. These were created by saving the link weights of MLPs trained with backpropagation on the combined MNIST and math symbols database. The Function Approximator MLP is generated by specifying a network of the same size as an available pre-trained network and initializing the network's links with the saved weights.

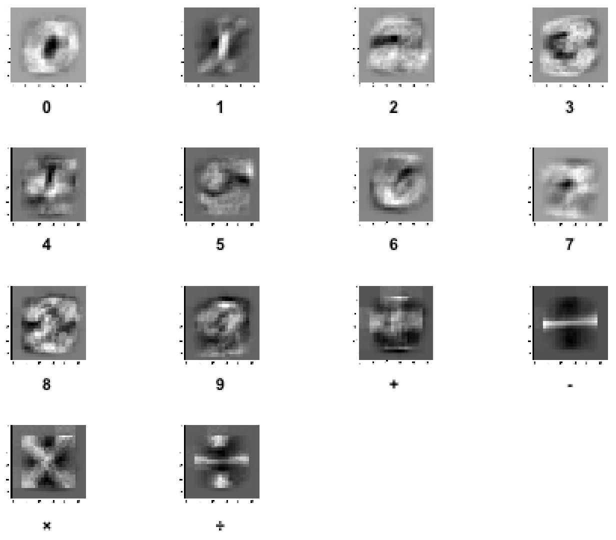


Fig. 4.3.3.1. Learned symbol representations of a pre-trained MLP

Images of symbolic representations learned by the pre-trained 2-layer MLP [784, 14], created by translating the 784 link weights for each output node into pixel values. Large weights correspond to pixels identified as important to recognizing a symbol; white pixels indicate large positive weights (a foreground pixel here would strongly imply this symbol) and black pixel values indicate large negative weights (a foreground pixel here would strongly imply a different symbol).

If an MLP is desired with dimensions other than those of the three provided pre-trained networks, new networks can be trained and saved for future use by running the "Classifier" recipe on its own.

4.3.3.2. MLP flow sequence

After the file parser has processed a handwritten function into a folder containing images of the constituent symbols, each successive image file in the folder is handed to the MLP. A 28x28 pixel image is converted to a flattened array of 784 (28×28) pixel values, which are used as the input activation for the slots of the 784 layer_0 nodes.

A step function governs the spread of activation through the MLP. Activated nodes, defined as nodes with any slot whose activation > 0, are called by the step function to pass their activation values held in their slots to their gates, where the activation is subject to the gate function.

With the exception of output layer nodes, which have identity functions, all MLP node gates employ a ReLU function, where $f(x) = \begin{cases} 0, & x \leq 0 \\ x, & x > 0 \end{cases}$. This function returns 0 for activations equal to or less than 0 and an identity function otherwise, and was selected due to its slight advantage in performance (1-2% improvement in classification accuracy) over the hyperbolic tangent function (tanh).

The step function then calls for activation values to be sent from the gate of a layer$_n$ node to the slots of all layer$_{n+1}$ nodes to which it is connected via a link. The activation is multiplied by the link weight before arriving in the layer$_{n+1}$ target slot, which sums all activations received from its origin nodes in layer$_n$.

The step function repeats until activation has traveled from the MLP input layer through all hidden layers into the last layer of the network, where the activation in each of the 14 output node gates is collected into an array. The 14 array indices represent the 14 possible predictions, with indices 0-9 reserved for the corresponding digits 0-9 and indices 10-13 denoting +, -, ×, and ÷, respectively.

A softmax function, given by $\sigma(x_j) = \frac{e^{x_j}}{\sum_i e^{x_i}}$, is applied to the array to force all 14 values to sum to 1. The resultant largest array value indicates the strongest probability that the input image depicts the symbol represented by its index, and is therefore selected as the prediction of the MLP for that image.

4.3.4. ReCoN

The array of predicted symbol values returned by the MLP as a result of processing a handwritten function image is used to construct a ReCoN that represents the function in its structure.

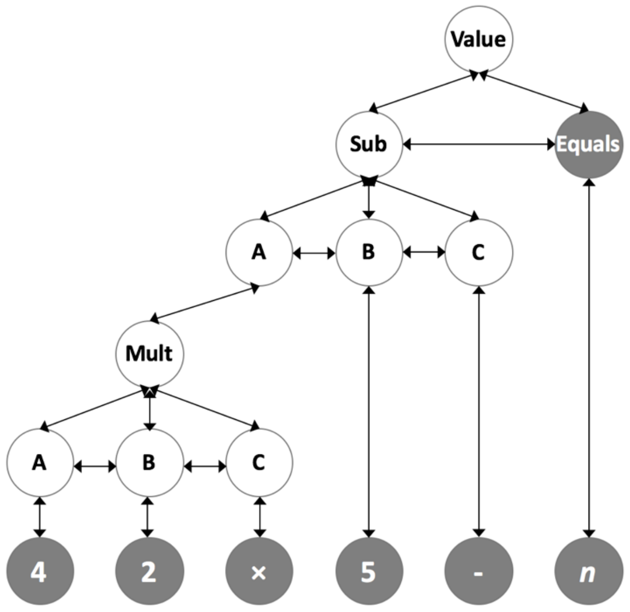


Fig. 4.3.4. Example ReCoN structure

The identified function 4×2-5 generates this ReCoN. The topmost node, or "root node", has two "child" nodes: the left branch represents the function's hierarchical operations, and the right branch represents the function value. Grey indicates "leaf" nodes.

The activation flows through the network hierarchy according to the ReCoN specifications, finally activating each of the last layer "leaf" nodes in sequence. When a leaf node receives activation, if it holds a value, it pushes that value onto the stack (see Figure 4.2.2. and Figure 4.3.5). If the value pushed is a mathematical operator, the prior two values on the stack are accepted as its operands, and the result of the operation is pushed to the top of the stack.

Only the last leaf node does not hold a value. When it receives activation, it pulls its value from the top of the stack. This value is handed to its parent node and compared against the function label. If all symbols have been correctly classified, the function will be properly executed and return the numeric value specified by the function's label, otherwise the value returned by the ReCoN will be incorrect.

Confirmation of the root node's request is not predicated on a correct or incorrect function value; request confirmation depends only on the correct sequence of activation spreading through the network to return a value, rather than failing or timing out before a value can be returned.

This may at first seem counterintuitive, but the internal confirmation of a hypothesis in a biological neural network does not necessarily indicate that it accurately reflects reality, only that the script has executed satisfactorily. Hemispatial neglect, the loss of awareness in one side of the field of vision caused by damage to the corresponding cerebral hemisphere, provides an extreme example of this principle; when asked to draw a clock, individuals with hemispatial neglect will often complete only half of the clock face (Chen & Goedert, 2012). In this case, it could be said that the request to draw the clock has been confirmed, as the individual believes that the script governing clock drawing has been successfully executed, but the confirmation is inaccurate.

4.3.5. Stack

In this implementation, the stack is responsible for executing the calculations represented by the ReCoN. It functions as a stacked (Reverse Polish) calculator, receiving input values from activated leaf nodes, performing operations in sequence, and pushing the results to the top. The top result is pulled into the last leaf node when it receives activation.

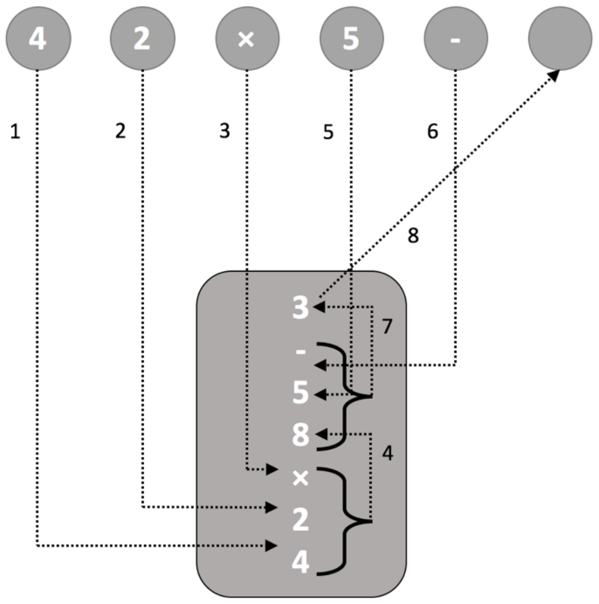


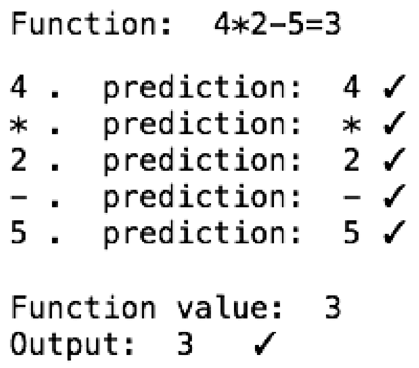
Fig. 4.3.5. Sequence diagram of stack execution

The stack execution sequence of the function 4×2-5. The stack calculates 8 from 4×2 before 5 and - are pushed on top. 3 is then calculated from 8-5, and pulled into the last leaf node.

This implementation provides functionality only for positive input integers in functions that are limited to the four basic algebraic operators and executed left-to-right, but the stack calculator could be revised to respect order of operations without altering the ReCoN structure or training data. Other possible alterations and extensions are discussed in Section 5.1.

4.3.6. Output

When run from a command line interface (CLI), the printed output of the Function Approximator provides the function label, the label for each symbol and the classifier's prediction for that symbol, the correct value for the function, and the ReCoN output with a ✓ indicating a correct function evaluation and a ☒ indicating an incorrect function evaluation, as shown in Figure 4.3.6.1.



```
Function:  4*2-5=3

4 .  prediction:  4 ✓
* .  prediction:  * ✓
2 .  prediction:  2 ✓
- .  prediction:  - ✓
5 .  prediction:  5 ✓

Function value:  3
Output:  3   ✓
```

Fig. 4.3.6.1. Function Approximator CLI output

Command line output shows results for the classification and evaluation of 4×2-5

4.4. MESH

MESH is a graphical editor developed by MicroPsi Industries for instantiating MicroPsi agent environments and running neural network agents. It provides a graphical user interface (GUI) in which the Function Approximator component architectures and activation spreading can be visualized in 3D.

The implementation described here is visualized in MESH version 0.2.4; as of this writing, a later version is publicly available for download on the MicroPsi Industries website (http://www.micropsi-industries.com).

4.4.1. MESH Classifier

The MLP classifier described in 4.2.3. consists of 784 input nodes, *n* layers of hidden nodes, and 14 output nodes. While this arrangement can be modeled in MESH, there are some drawbacks, as demonstrated in Figure 4.4.1.1.

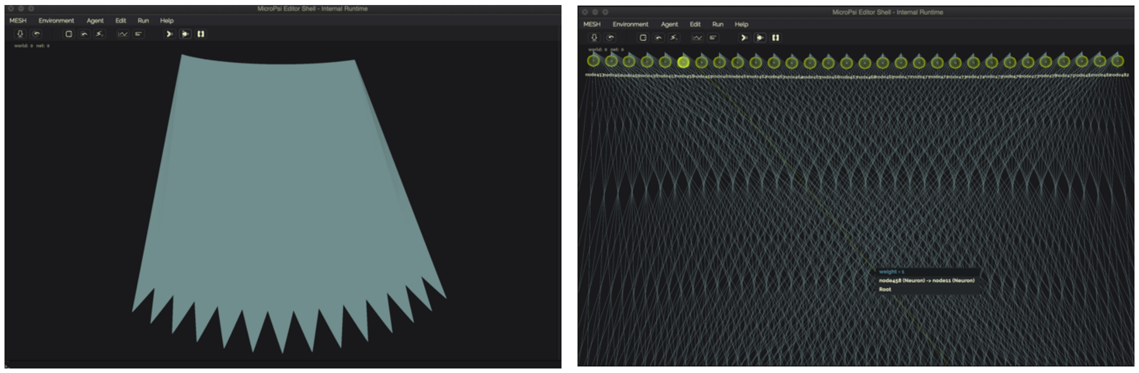

MicroPsi Editor Shell - Internal Runtime
MESH
Environment
Agent
Edit
Run
Help
MicroPsi Editor Shell - Internal Runtime
MESH
Environment
Agent
Edit
Run
Help

Fig. 4.4.1.1. MLP Classifier in MESH without flow modules

A 2-layer MLP with 784 input nodes and 14 output nodes. Left: an overview of the network; Right, a zoomed-in view with one of the links highlighted.

Although the zoomed-in view nicely reveals the level of visual detail available in MESH, the overlap of 10,976 links is too convoluted to provide an informative representation of the entire network. More importantly, the processing power and time required to render any additional hidden layers is prohibitive for a standard-build personal computer.

A workaround can be applied in the form of flow modules. Flow modules are individual computational units that act as functional groups of nodes and together form a flow graph. Flow modules are not only visually compact, they require significantly less processing power for MESH to render and run than the equivalent number of individual nodes.

The MLP classifier in Fig. 4.4.1.2. was adapted for use from a MicroPsi Industries demo agent built to classify the MNIST dataset.

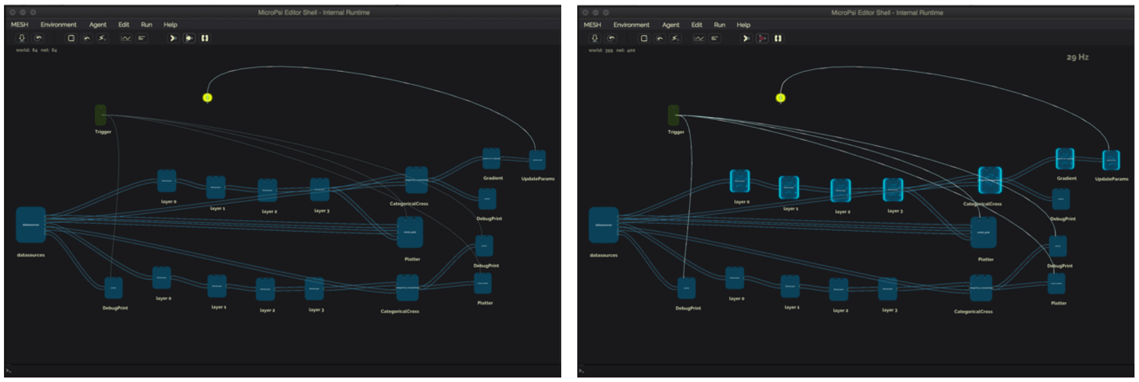


Fig. 4.4.1.2. MLP Classifier in MESH with flow modules

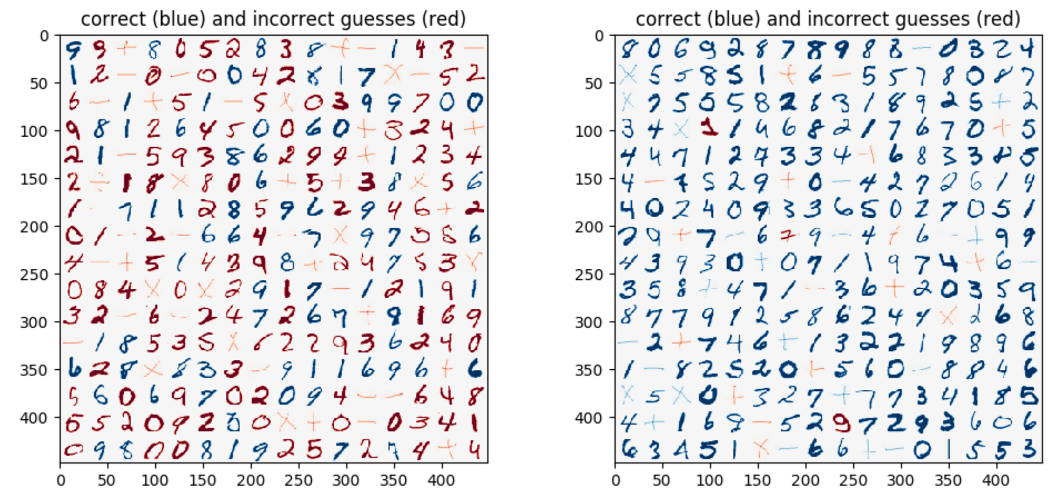
Left: The MLP Classifier is displayed here with a datasource, four flow modules, a shallow copy of the flow modules, and printers and plotters for visual output; Right: Activation flows from the datasource through the MLP layers and into the printers, plotters, and modules that handle network parameters to update for the next epoch.

The MLP is displayed with a top row of four flow modules, labeled as layers 0 through 3 and representing a network with 784 input nodes, 3 hidden layers of 150, 100, and 50 nodes, and 14 output nodes. Each of the four flow modules has 784 inputs and "hidden" dimensions of 150, 100, 50, and 14, respectively, and the flow graph they form is fully connected. When the last flow module receives activation, all flow graph calculations are performed in a single time step.

Beneath the top row of flow modules is a visually identical row of flow modules. This is the shallow copy of the flow graph, which is connected to the validation data and computes the network's loss function using categorical cross entropy, which calculates $H(p,q) = -\sum_x p(x)log(q(x))$, where p is the target vector and q is the vector of predictions output by the flow graph.

In addition to the datasource module, which represents the combination MNIST and math operations dataset, also pictured are a trigger module and execution node to induce activation in the network, as well as various debuggers, printers, plotters, and various administrative units required to implement gradient descent and update the network parameters accordingly. In MESH, the activation flow through the network is represented by a glowing animation (see Figure 4.4.1.3).

Images of the classification results generated by the printers and a chart of the loss function derived from the plotters are saved to a folder for review.



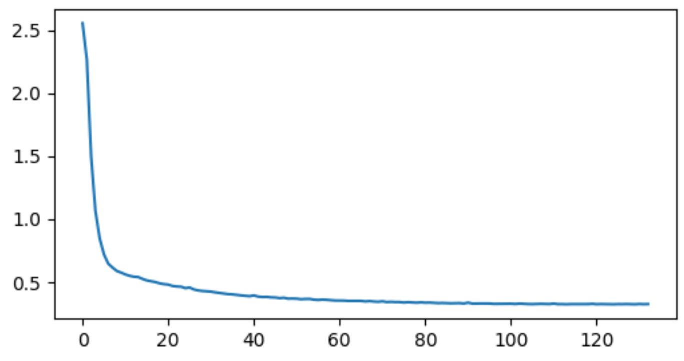


Fig. 4.4.1.4. MESH Classifier results and loss function

Top: Left: Results from the first training epoch; Right: Results from a later epoch. Bottom: Final graph of the classifier loss function

4.4.2. MESH ReCoN

A ReCoN can be modeled and run as a stand-alone network in MESH. In this case, activation is simulated by providing *sub*-activation to the ReCoN root node.

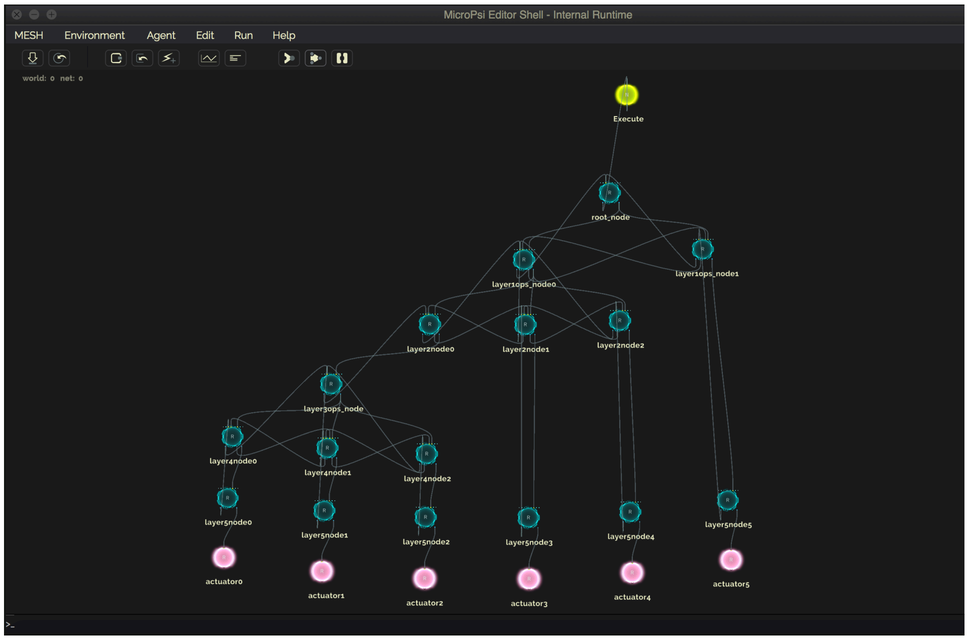
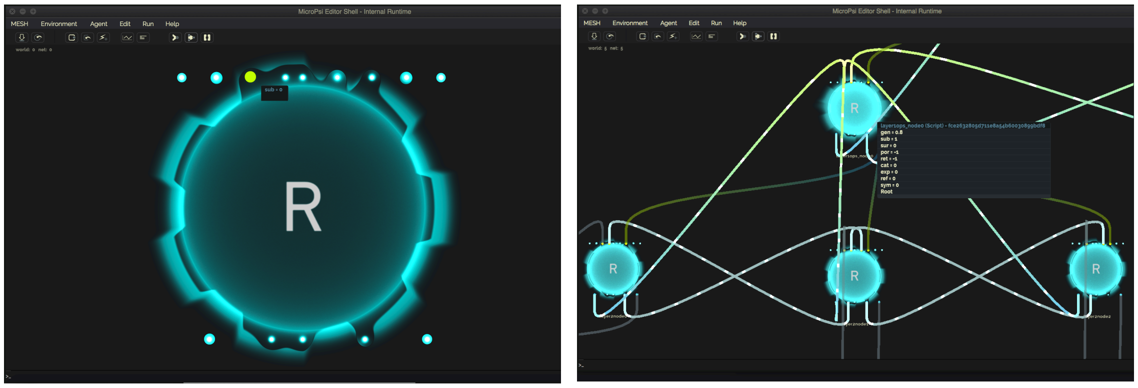


Fig 4.4.2.1. ReCoN in MESH

ReCoN nodes are shown in blue. An execution node, in yellow, simulates activation to the network by sending sub-activation to the ReCoN root node. Actuator nodes, in pink, provide the sur-activation necessary for the last layer nodes to respond affirmatively to a request for confirmation.

ReCoN nodes are denoted by spheres with a row of smaller circles above, representing slots, and below, representing gates (Fig. 4.4.2.2., left). ReCoN nodes in MESH have four types of slots and gates not included in the nodes used in the Function Approximator ReCoNs, but are made available to represent more types of relationships between nodes. These additional relationships are not used for the networks discussed in this thesis, but are important to understanding the potential uses for ReCoNs and are given further consideration in Section 5.2.

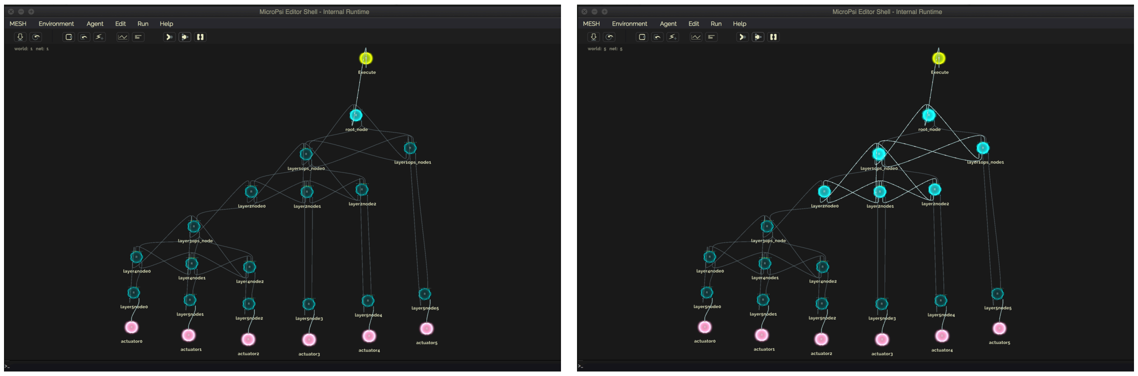


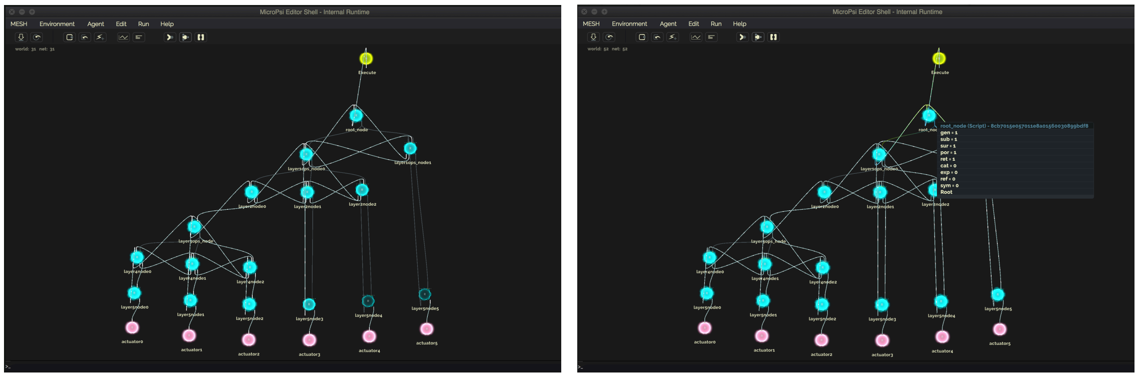
layer1ops_node0 (Script) - fce2632805d711e8a54b60030899bdf8
gen = 0.8
sub = 1
sur = 0
por = -1
ret = -1
cat = 0
exp = 0
ref = 0
sym = 0
Root

Fig. 4.4.2.2. MESH ReCoN node display

Left: A ReCoN node with the sub slot highlighted; Right: Highlighting a node displays its current level of each activation type. The stronger "glow" of the top node indicates a higher level of activation than its three child nodes.

The MESH GUI allows the controlled script execution characteristic to ReCoNs to be easily followed via the animated "glow" depicting activation spreading through the network hierarchy. Nodes have three possible levels of "glow", corresponding to low, medium, and high levels of activation, respectively. Fig. 4.4.2.3. displays sequential excerpts from the activation flow through a ReCoN.





```
root_node (Script) - 8cb7015e057011e8a01560030899bdf8
gen = 1
sub = 1
sur = 1
por = 1
ret = 1
cat = 0
exp = 0
ref = 0
sym = 0
Root
```

Fig. 4.4.2.3. ReCoN activation spreading in MESH

Top left: Activation moves from the execution node to the ReCoN root node. Actuator nodes hold their activation until requested by the nodes in the last layer of the ReCoN; Top right: Activation flows into the first and second layers. The successor node in the first layer is suppressed by its predecessor from activating its child node in the last layer; Bottom left: Activation reaches the last layer nodes, which confirm their parents' requests in sequence; Bottom right: Activation has propagated fully through the network and back up to the root node, which displays activation values of 1, corresponding to "confirmed".

The first network step (tracked in the top left corner of the display), introduces activation from the execution node to the ReCoN root node. The actuator nodes are also active, but hold their activation until they are requested by the nodes in the last layer of the ReCoN (Fig. 4.4.2.3., top left).

In the next step, as it receives continued activation from the execution node, the root node activation changes from .2, *preparing*, to .6, *requesting*, prompting activation to flow from the root node into the first layer and for the first layer nodes to change their respective activation values to .2 accordingly. This induces a *wait* signal to be sent via the *por* link connecting the first layer nodes, which sets successor node activation to .4, *waiting*, while the activation of the predecessor node moves to .6, *requesting*, and activation is released from the predecessor node into the second layer (Fig. 4.4.2.3., top right).

A parent node continually requesting confirmation from its children will change its activation to .8, *pending*, until its request is either confirmed or denied. The activation level of a suppressed successor node will remain at .4, corresponding to *waiting*, until its predecessor receives the response requested from its child nodes and lifts the *wait* signal. This activation gating prevents the successor node from sending activation to its children before the predecessor node has received confirmation from its subnetwork.

When activation reaches the last layer nodes of the ReCoN, if they are not connected to sources that yield positive *sur*-activation, symbolizing grounded input representation, they will respond with an activation value of 0, indicating a *failed* state. This failure propagates backwards through the network, and the root node's request would be unconfirmed. In this implementation, the last layer ReCoN nodes are connected to actuator nodes that provide the activation necessary for the last layer nodes to confirm their parents' requests. Each root node therefore changes its state in sequence to *confirmed* with an activation value of 1, and this activation is sent back through the hierarchy (Fig. 4.4.2.3., bottom left) until all requested confirmation has been received.

Once activation has propagated fully through the network and back up to the root node, the activation value of the root node is 1 and the hypothesis it represents is considered confirmed (Fig. 4.4.2.3, bottom right).

Chapter 5.

Results and Analysis

This section provides analysis and evaluation of the Function Approximator implementation in general and of ReCoNs specifically, both within the context of the Function Approximator and, more broadly, as a potential model for cortical activity. It concludes with a discussion of the contributions made by this thesis.

5.1. Function Approximator

The Function Approximator recipe and the neural network architecture on which it runs were written in Python, a popular language for machine learning. Python was chosen not only to allow leveraging of the SciPy ecosystem components, including NumPy for matrix manipulation and sci-kit image for image parsing, but to ensure language compatibility with MicroPsi2, which was necessary for the MESH visualization and desirable for possible future extensions.

The quality and quantity of available data proved to be a not insignificant obstacle in training the MLP classifier. The selected Kaggle dataset of handwritten mathematical operator symbols was the only suitable publicly accessible option, and unfortunately (as discussed in Section 4.3.1.2.) did not contain enough multiplication or division symbols to avoid undue system prejudice towards those operators. Additionally, the stroke width of the Kaggle symbols is uniformly significantly narrower than that of most MNIST symbols, which likely also contributed to training bias. Despite the practical complications involved in compiling an ideal dataset, if greater classification accuracy had been required, it may have been preferable to do so. However, as the Function Approximator was built as a showcase and proof-of-concept for ReCoNs, optimizing the performance of the MLP classifier was not a primary concern; the classifier achieved 96% accuracy with 4 layers of size 784, 240, 60, 14 respectively, which was adequate for purpose.

The Function Approximator, as implemented, leaves significant room for improvements and extensions. The MLP could incorporate any number of classic optimization techniques to enhance performance, or be replaced by a CNN. An improved classifier would facilitate the inclusion of more symbols in the training set without a prohibitive drop in accuracy, opening possibilities for additional function operations. If symbols for left "(" and right ")" parentheses were added, they could be used to indicate modifications to the stack calculator's left-to-right function execution; alternatively, or additionally, the system could be reorganized to process functions via order of operations.

At the moment, although the stack calculator can technically handle multiple-digit numbers, as well as negative and decimal values, the ReCoNs built from the identified function arrays are structured to represent single-digit positive integers only. The choice to severely limit the capabilities of the Function Approximator was made to maintain the ReCoN structure as a simple and generally translatable abstraction of cortical activity, rather than including specific functionalities, such as bit shift representation for multiple-digit values, that would expand the Function Approximator but obscure or detract from the neurobiological model.

5.2. ReCoNs

The Function Approximator (re-)establishes that ReCoNs, as specified in Bach & Herger (2015), can be used to hierarchically represent and execute neuro-symbolic scripts, characterized here by algebraic functions, and that this can be performed entirely locally, without the need for a central control architecture. However, this implementation is merely a first step in exploring the full potential offered by the request confirmation model of cortical activity.

As discussed in Section 4.1., the ReCoNs generated by the Function Approximator are not implied to be a portrayal of numerical calculation in the brain, but are instead intended to be a generalization of a proposed model of hierarchical cortical excitation and inhibition during neuro-symbolic script execution. Algebraic functions were determined to be a straightforward and easy-to-follow illustrative example, as opposed to attempting to depict the ambiguities inherent in more complex scripts, such as making a meal or planning a trip, that could equally be represented by this structure. It should also be noted that the ReCoN nodes, as implemented, might be considered more analogous to a unified collection of cortical minicolumns, or possibly a "macrocolumn" (Mountcastle, 1997), than to a single minicolumn; this abstraction could be unraveled if desired into a larger and more granular simulation, but is useful here for clarity and conciseness in the model.

The Function Approximator serves as a foundational proof of concept, but does not make use of a number of characteristics inherent to ReCoNs that significantly extend their functionality. For instance, beyond the *sub*, s*ur*, *por*, *ret*, and *gen* gates used in the Function Approximator, ReCoNs have the capacity to encode additional neuro-symbolic associations, further expanding the options for hierarchical representations. In the same way that *sub*/*sur* denotes parent/child and *por*/*ret* denotes predecessor/successor, *cat*/*exp* can be used to denote category/exemplar, where the former is a category and the latter is an instance of that category, and *sym*/*ref* to denote symbol/referent, where the former is a symbol and the latter is a concept represented by that symbol; for example, *sym*/*ref* is useful for linking an object representation with words for that object. These typed links enable ReCoNs to structurally encode the complex relationships embedded in higher-order scripts and schemas.

In contrast to the ReCoNs described in Bach & Herger (2015), which learned to recognize previously visited environments, the link weights of the Function Approximator ReCoNs remain static, and the networks do not engage in any type of learning. Extending the Function Approximator ReCoNs into a more granular structure with adequate numbers of nodes, as suggested above, and imposing a learning rule, would allow them to be similarly translated into learning networks. In this case, the stacked calculator could be replaced entirely by a ReCoN trained to perform arithmetic calculations on the numeric values represented in its subnetworks.

In the Function Approximator, bottom-up activation is used solely to confirm an already made request, but could also serve as stimulus for a directed decision to send top-down activation through a relevant network. For example, in a reinforcement learning environment, the presence of a food item would send bottom-up activation through a network in a ReCoN-based agent that recognizes the object as edible, and, depending on the agent's environment, simulated urges ("hunger", "tiredness", etc.), and other parameters, may inspire the top-down activation of networks that perform the actions necessary to consume the food. Alternatively, if the agent is hungry, it can send top-down activation to search for the presence of food in its environment, and receive bottom-up activation when it has been located. This modulation of complementary top-down/bottom-up activation by a sensory-driven motivational system connected to a reward architecture is a natural next step for ReCoN development.

5.3. Contributions

ReCoNs offer a unique combination of certain structural and functional features that may prove useful to the field of biomimetic neural networks.

The Function Approximator illustrates the proposed role of request confirmation in top-down and bottom-up excitation and inhibition between cortical modules when executing hierarchical neuro-symbolic sensorimotor scripts. Top-down activation of the "root node" of a ReCoN-embodied script leads to its execution, whereas perceptual input sends activation bottom-up through related networks. Child and predecessor nodes inhibit their parent and successor nodes, respectively, while awaiting confirmation from their own subnetworks. This constrained recurrency ensures an ordered spread of activation through the hierarchy and enables local control.

ReCoNs are inherently state-based networks, and, in contrast to those ANNs that model neurons as pure functions with link weight as the single parameter, ReCoN links and nodes are designed such that they can act as stateful machines unto themselves, accepting any number of parameters. This is especially relevant when considering recent experimental results that reveal that each biological neuron functions as a collection of independent threshold units, with output dependent on stimulation location and origin

(Sardi, Vardi, Sheinin, Goldental & Kanter, 2017); with trivial modification, ReCoNs are capable of accommodating these variables. ReCoNs are able to model multiple types of output signals, which offers the opportunity to simulate both spikes and bursts, or oscillatory and non-oscillatory channels, or multiple cell types.

When compared to existing biomimetic approaches, in some respects ReCoNs appear to inhabit a comparable space to capsule networks and the cortical column implementation of HTM, which similarly purport to model cortical activity and account for the recurrency integral to the function of biological cortical circuits (Gilbert & Sigman, 2007; Shu, Hasenstaub & McCormick, 2003). While it is difficult at this stage to compare the Function Approximator ReCoNs, which are rudimentary prototypes that do not perform learning, with these more sophisticated learning networks, some differences are apparent. Unlike capsule networks, which are specialized to address the limitations of CNNs in visual recognition, ReCoNs are not structured to perform batch learning, nor must they necessarily optimize a single objective function for a specific task, characteristics that seem to align them more closely with HTM (Ahmad, 2017).

However, in contrast to HTM (and to capsule routing), ReCoNs are designed to represent objects and concepts at a neuro-symbolic level, encoding partonomic associations between symbols with typed links and constrained hierarchical activation spreading.

Chapter 6.

Summary & Conclusions

This thesis proposes ReCoNs, hierarchical neuro-symbolic spreading activation networks first introduced in Bach & Herger (2015), as a candidate model for the execution of sensorimotor scripts in the cortex. It provides background on both biological and artificial neural networks, and surveys existing biomimetic machine learning approaches against which ReCoNs can be contextualized. The neurological foundations, history, and a full specification of ReCoNs are given, followed by a detailed description and visualization of the Function Approximator, a showcase implementation for the ReCoNs. Finally, an evaluation and analysis is offered for both the Function Approximator and the ReCoNs, along with an exploration of the contributions that ReCoNs make to the field of biomimetic machine learning.

6.1. Directions for future research

The ReCoNs built for this thesis were constructed with an eye toward eventual deployment in MicroPsi2; section 5.2. mentions that a natural next step for ReCoN development would be to unlock their learning capacity and use them as the basis for an agent with motivation and reward systems that can freely explore simulated environments, which can be accomplished within the existing MicroPsi2 architecture.

Another possible avenue of exploration would be to incorporate ReCoNs with one or more other types of neural networks, such as Generative Adversarial Networks

(GANs), essentially employing ReCoNs as a locus of control for more specialized functionalities.

6.2. Conclusion

This thesis represents a preliminary step toward establishing the usefulness of ReCoNs and the request confirmation model of spreading activation; further work is necessary to determine their full potential, both as a technology and as a candidate for modeling cortical activity. However, Chapter 2 of this thesis covers a number of field-defining applications inspired by prior architectures. ReCoNs may prove to be a valuable contribution in their own right, but it is hoped regardless that they will serve as groundwork for future research into cortical activity and biomimetic learning systems.

Appendix A.

Code Resources

The following is a reproduction of an iPython notebook[3] documenting the foundational neural network architecture on which the Function Approximator runs.

NODENET

This document describes Nodenet, a state-based neural network generator that creates linked networks of nodes. A network can be of any structure; configuration specifications come from predefined scripts ("recipes").

The recipe examined here constructs a simple multilayer perceptron (MLP) with three fully-connected layers built for classification of a combination of MNIST and images of addition, subtraction, multiplication and division symbols selected from a Kaggle handwritten math symbols dataset.

```
def build_nodenet(nodenet):

        # enter nodes per layer
        network_dimensions = [784, 60, 14]

        node_data = config.generate_node_data(network_dimensions)
        config.add_nodes(nodenet, node_data)

        link_data = config.generate_link_data(nodenet)
        config.link_nodes(nodenet, link_data)

        return network_dimensions

if __name__ == "__main__":

        nodenet = Nodenet()
        network_dimensions = build_nodenet(nodenet)
```

The desired number of nodes per layer is entered in the `network_dimensions` array. The `[784, 60, 14]` specifications above will build a 3-layer network with 784 nodes in the input layer, 60 nodes in the hidden layer, and 14 nodes in the output layer.

Looking more closely at `build_nodenet()`, we see that it first generates data about nodes from the `network_dimensions`, then uses that data to add nodes to the nodenet:

```
node_data = config.generate_node_data(network_dimensions)
config.add_nodes(nodenet, node_data)
```

Once the nodes have been added to the nodenet, link data is generated from information about the nodenet, then used to create links between the nodes:

[3] The notebook, along with code for the Function Approximator, can be found at https://github.com/kvgallagher/nodenet

```
link_data = config.generate_link_data(nodenet)
config.link_nodes(nodenet, link_data)
```

NODES

Nodes are fundamental network units, generally arranged in layers, that perform transfer functions on one or more aggregated inputs. They have a unique UID, a name, an activation value, a node function (described later), and vectors of slots and gates.

```
class Node:
    def __init__(self, name=None, slot_vector=None, gate_vector=None, node_function=None, activation=0):
        self.uid = uuid4()
        self.name = name
        self.node_function = node_function if node_function is not None else self._default_node_function
        self.slot_vector = slot_vector
        self.gate_vector = gate_vector
        self.activation = activation
```

Nodes receive input from the nodes they are connected to in the previous layer via slots:

```
class Slot:
    def __init__(self, name=None, activation=0):
        self.name = name
        self.activation = activation
```

Slots have an activation value representing their received input and a name that indicates their type. The type of a slot indicates a neuro-symbolic relationship between it and the origin gate from which it receives activation.

Although the slots used in the MLP recipe described here are uniformly of type *gen*, indicating a general associative relationship, other network structures use more types, such as *sub*/*sur*, denoting a parent/child relationship, or *por*/*ret*, denoting a predecessor/successor relationship.

```
SLOT_TYPES = {
    "gen": [
        "gen",
        0
    ]
}
```

Nodes send their output activation to the nodes they are connected to in the next layer via gates:

```
class Gate:
    def __init__(self, name, activation, parameters=None, gate_function=None):
        self.name = name
        self.activation = activation
        self.parameters = parameters if parameters else {}
        self.gate_function = {
                        "default": self._default_gate_function,
                        "output": self._output_gate_function
                }[gate_function]
```

Like slots, gates also have an activation value and a name indicative of their type (gate types are identical to slot types). Gates can additionally accept parameters, such as threshold value, which will prevent a gate from releasing its activation unless or until it meets or exceeds the threshold.

The gate function performs the transformation of the node input. The functions specified here are rectifier (ReLU)), used in the MLP for all nodes except for the output layer nodes, which have an identity function.

```
    def _default_gate_function(self, activation):
        self.activation = 0 if activation < 0 else activation # ReLU

    def _output_gate_function(self, activation):
        self.activation = activation
```

Nodes can also be of different types, which are defined by the number and types of their slots and gates. The nodes used in the MLP are uniformly of type *register*, which has a single *gen* slot and a single *gen* gate.

```
NODE_TYPES = {
    "register": [
        [ "gen" ],
        [ "gen" ]
    ],
    "sensor": [
        [ "gen" ]
    ]
}
```

LINKS

Links connect a gate of an origin node to a slot of a target node in the next layer.

```
class Link:
        def __init__(self, origin_node, origin_gate, target_node, target_slot, weight=None):
                self.uid = uuid.uuid4()
                self.origin_node = origin_node
                self.origin_gate = origin_gate
                self.target_node = target_node
                self.target_slot = target_slot

                # OPTIMAL WEIGHT INITIALIZATION
                self.weight = weight if weight is not None else random.uniform(-.125, .125)
```

Links have a unique UID, an origin node and gate, a target node and slot, and a weight value that acts as a coefficient by which the activation traveling over the link is multiplied.

The weights of links in the network are key to its learning capacity. As the network receives feedback on its predictions, weights are tuned via backpropagation (discussed in RUN) according to the difference between the predicted value and the actual value. This adjusts the relative influence of the origin output on the target input.

In the MLP, links are initialized randomly but in a uniform distribution within a constrained range of -.125 to .125, which was determined to be optimal for learning.

BUILD NODENET

When `build_nodenet()` is called, it uses the number of layers and number of nodes per layer specified in `network_dimensions` to generate a list of layers, each a list of the name and type for each node assigned to that layer.

```
def generate_node_data(network_dimensions):
        return [[["layer"+str(layer_index)+"node"+str(n), "register"]
        for n in range(0, num_nodes)] for layer_index, num_nodes in enumerate(network_dimensions)]
```

This list is then used to create layers of actual node objects and add them to the nodenet:

```
def add_nodes(nodenet, node_data):
        nodenet.add_layers(node_factory(node_data))
```

`node_factory` creates the nodes, specifying gates with ReLU functions for all nodes other than those in the last layer, which are created with identity gate functions:

```
def node_factory(node_data):
    if len(node_data) > 0:
        layers = []
        for layer_index, layer in enumerate(node_data):
            if layer_index < len(node_data)-1:
                node = [Node(node[0], *_get_slots_and_gates(*NODE_TYPES.get(node[1]))) for node in layer]
                layers.append(node)
            else:
                node = [Node(node[0], *_get_output_slots_and_gates(*NODE_TYPES.get(node[1]))) for node in layer]
                layers.append(node)
        return layers
    else:
        raise ValueError("please pass in node data or number of nodes to create")
```

The `get_(output)_slots_and_gates` functions call `slot_factory` and `gate_factory`, which create the requested slots and gates, respectively.

```
def slot_factory(slot_names):
        # return a list of slots, generated from slot types based on a list of names
        return [Slot(*SLOT_TYPES.get(name)) for name in slot_names]
```

```
def gate_factory(gate_names, is_output_node):
    # return a list of gates, generated from gate types based on a list of names
    if is_output_node == False:
        return [Gate(*GATE_TYPES.get(name) + ["default"]) for name in gate_names]
    else:
        return [Gate(*GATE_TYPES.get(name) + ["output"]) for name in gate_names]
```

The list of lists (representing layers) of nodes is added to the nodenet, and links are created between the nodes:

```
def generate_link_data(nodenet):
        return [[[{"origin": [origin_node.name, "gen"], "target": [target_node.name, "gen"]}
        for origin_node in nodenet.layers[layer_index-1]] for target_node in layer]
        for layer_index, layer in enumerate(nodenet.layers) if 0 < layer_index < len(nodenet.layers)]
```

`generate_link_data` generates a list of dicts that contain the information needed to create each link: the name of its origin node and gate and its target node and slot. In the MLP, the *gen* slot of every node in a layer is connected to the `gen` gate of every node in the preceding layer.

The link data is then used to create layers of links and add them to the nodenet:

```
def link_nodes(nodenet, link_data):
        nodenet.links_list = [[[(create_link(nodenet, link))
        for link in node] for node in layer] for layer in link_data]

def create_link(nodenet, link_data):
        origin_node = nodenet.node_dict[link_data.get("origin")[0]]
        origin_gate = origin_node.get_gate(link_data.get("origin")[1])
        target_node = nodenet.node_dict[link_data.get("target")[0]]
        target_slot = target_node.get_slot(link_data.get("target")[1])

        return Link(origin_node, origin_gate, target_node, target_slot)
```

The nodenet itself now has a unique UID, a name ("nodenet"), a list of layers of nodes (`self.layers`) and a list of links (`self.links_list`) that connect them, and an unlayered dict of nodes that is helpful for administrative purposes (`node_dict`).

Additionally, the nodenet has a `learning_rate`, set to .05, which tells the network how much it should change its weights after each prediction, and a `RATE_DECAY`, which modifies the learning rate over time so that the network weights fluctuate more at first, as it starts to "figure out" its parameters, and slowly change less and less as it receives feedback and begins to settle on weights that make better predictions.

```
class Nodenet:
        def __init__(self, name = None):
                self.uid = uuid4()
                self.name = name
                self.learning_rate = .05
                self.RATE_DECAY = .0001
                self.node_dict = {}
                self.layers = []
                self.links_list = []
```

SET ACTIVATION

The nodenet is now an MLP ready to classify our dataset, which consists of images of handwritten digits (0-9) and addition, subtraction, multiplication, and division symbols, for a total of 14 categories (10 digits + 4 symbols).

The dimensions of `[784, 60, 14]` were set for a reason: the output layer has 14 nodes, each representing a possible digit or symbol that an image could depict. The images in the dataset are 28px x 28px = 784px total. Each of the 784 nodes in the input layer will receive a single pixel from the image.

The images are represented as multidimensional arrays of pixel values with accompanying one-hot encoded arrays of labels, where the first 10 indices represent the digits 0-9 and the last 4 indices represent the mathematical operators.

The images are fed one by one into the nodenet:

```
for i, image in enumerate(images):

                config.set_activation(nodenet, image)
                control.run(nodenet, labels[i], i, run_type)
```

`set_activation` first flattens the multidimensional image into a single array of pixel values, then normalizes the values between 0 and 1, and sets the slot activation of each of the 784 input nodes to be equal to the pixel at the corresponding index in the flattened array:

```
def _flatten_image(image):
        return [pixel for row in image for pixel in row]

def set_activation(nodenet, image):
        flattened_image = _flatten_image(image)

        for i, node in enumerate(nodenet.layers[0]):

                pixel = flattened_image[i] * (1/255) # normalize values between [0,1]
                slot = node.get_slot("gen")
                slot.activation = pixel
```

RUN

```
def run(nodenet, target_output, image_index, run_type):
        output = _step_function(nodenet) # softmax output

        _pretty_print(output, target_output, image_index)

        if run_type == "train":
                _update_weights(nodenet, output, target_output, image_index)

        _zero_nodes(nodenet)
        _zero_gates(nodenet)
```

With the input activation set, the nodenet is ready to predict the image that the activation represents.

The end result of `run()` is a one-hot encoded array of 14 values. The highest value indicating the strongest possibility that the image depicts the digit or symbol represented by that index. This array is the output of a step function:

```
def _step_function(nodenet):

        for i, layer in enumerate(nodenet.layers):
                _net_function(nodenet)
                _link_function(nodenet)

                # fetch output from last layer
                if i == len(nodenet.layers)-1:
                        output = [gate.activation for node in layer for gate in node.gate_vector]

        return _softmax(output) # apply softmax
```

For each layer in the nodenet, `_step_function()` calls a `net_function()`:

```
def _net_function(nodenet):
        node_dict = nodenet.node_dict

        for node in node_dict.values():
                for slot in node.slot_vector:
                        if slot.activation != 0:
                                node.node_function(slot.activation)
                                slot.activation = 0
```

The `net_function()` calls a `node_function()` for all of the nodes in the nodenet that received activation in the preceding time step. In turn, the node function for each node passes the received activation to its gates via their gate function:

```
def _default_node_function(self, activation):
# calls all gate functions, passes value from slot
    for gate in self.gate_vector:
        gate.gate_function(activation)
```

The gate function performs its transform on the activation, and `_step_function()` calls `_link_function()`, which checks all links in the nodenet to see if their origin gate received activation. If so, the origin gate activation is multiplied by the link weight and the product is sent to the target slot of the node in the next layer.

```
def _link_function(nodenet):
        for layer in nodenet.links_list:
                for node in layer:
                        for link in node:
                                if link.origin_gate.is_active():
                                        _send_activation_to_target_slot(link)
```

When activation reaches the last layer of nodes, `_step_function()` gathers the activation from each node into an array and applies a softmax function, which creates a probability distribution by forcing all values in the array to sum to 1. The index of the highest value in the softmax array indicates the strongest probability that the input image depicts the digit or symbol represented by that index.

```
def _softmax(output):
        exp_output = np.exp(output - np.max(output))
        return exp_output / exp_output.sum()
```

TRAINING

At first, the nodenet output is meaningless; it requires feedback on its predictions to improve. This feedback is calculated by taking the difference of the prediction and the actual desired output, and feeding it backwards through the network via backpropagation.

Backpropagation proportionally distributes "credit" for successful predictions or "penalty" for inaccurate predictions in the form of updates to the link weights. As stated in LINKS, link weights determine how much impact a particular node will have on the calculation of the nodes it is connected to in the next layer.

`_update_weights()` handles the error calculation and credit assignment for the last layer of links:

```
def _update_weights(nodenet, output, target_output, image_index):
        output_links = nodenet.links_list[len(nodenet.links_list)-1]
        learning_rate = _decay_learning_rate(nodenet)
        error_array = target_output - output

        # calculate and set weights for each link to output nodes
        for node_index, output_node in enumerate(output_links):
                error = error_array[node_index]

                for i in range(len(output_node)):
                        link = output_node[i]

                        # store weighted errors on link origin nodes
                        link.origin_node.activation += link.weight * error * _relu_deriv(link.origin_gate.activation)

                        link.weight += learning_rate * link.origin_gate.activation * error

        if len(nodenet.links_list) > 1:
                _backprop(nodenet, learning_rate)
```

A one-hot encoded `error_array` is created from the difference between the `target_output` label and the nodenet prediction. For each node in the last layer, the weights of all links that connect to that node are multiplied by the value in the error array at the node index, as well as the derivative of the ReLU function applied by the gates of the connected nodes in the preceding layer.

If the network has hidden layers, `_update_weights()` calls `_backprop()` to send the error signal back through the rest of the network:

```
def _backprop(nodenet, learning_rate):
        i = len(nodenet.layers)-2

        while i > 0:
                prior_layer_links = _get_prior_layer_links(nodenet, i)

                for node_index, node in enumerate(nodenet.layers[i]):
                        links = prior_layer_links[node_index]

                        for link in links:
                                error = link.target_node.activation

                                # store weighted errors on link origin nodes
                                link.origin_node.activation += link.weight * error * _relu_deriv(link.origin_gate.acti
vation)

                                link.weight += learning_rate * link.origin_gate.activation * error
                i -= 1
```

Over time, the nodenet weights converge on values that result in accurate predictions.

Appendix B.

Sample ReCoN Activation Sequence

The activation sequence of a ReCoN computing a 2-operator function (4×2-5).

```
START

root_node 0.2 – preparing
root_node 0.6 – requesting
root_node 0.8 – pending

layer1ops_node0 0.2 – preparing
layer1ops_node1 0.2 – preparing

layer1ops_node0 0.6 – requesting
layer1ops_node1 0.4 – suppressed

layer1ops_node0 0.8 – pending
layer2node0 0.2 – preparing
layer2node1 0.2 – preparing
layer2node2 0.2 – preparing

layer2node0 0.6 – requesting
layer2node1 0.4 – suppressed
layer2node2 0.4 – suppressed

layer2node0 0.8 – pending
layer3ops_node 0.2 – preparing

layer3ops_node 0.6 – requesting

layer3ops_node 0.8 – pending
layer4node0 0.2 – preparing
layer4node1 0.2 – preparing
layer4node2 0.2 – preparing

layer4node0 0.6 – requesting
layer4node1 0.4 – suppressed
layer4node2 0.4 – suppressed

layer4node0 0.8 – pending
layer5node0 0.2 – preparing
layer5node0 0.6 – requesting
layer5node0 0.8 – pending
layer5node0 1 – confirmed

layer4node0 1 – confirmed

layer4node1 0.6 – requesting

layer4node1 0.8 – pending
layer5node1 0.2 – preparing
layer5node1 0.6 – requesting
layer5node1 0.8 – pending
layer5node1 1 – confirmed

layer4node1 1 – confirmed

layer4node2 0.6 – requesting

layer4node2 0.8 – pending
layer5node2 0.2 – preparing
layer5node2 0.6 – requesting
layer5node2 0.8 – pending
layer5node2 1 – confirmed

layer4node2 1 – confirmed

layer3ops_node 1 – confirmed

layer2node0 1 – confirmed

layer2node1 0.6 – requesting

layer2node1 0.8 – pending
layer5node3 0.2 – preparing
layer5node3 0.6 – requesting
layer5node3 0.8 – pending
layer5node3 1 – confirmed

layer2node1 1 – confirmed

layer2node2 0.6 – requesting

layer2node2 0.8 – pending
layer5node4 0.2 – preparing
layer5node4 0.6 – requesting
layer5node4 0.8 – pending
layer5node4 1 – confirmed

layer2node2 1 – confirmed

layer1ops_node0 1 – confirmed

layer1ops_node1 0.6 – requesting

layer1ops_node1 0.8 – pending
layer5node5 0.2 – preparing
layer5node5 0.6 – requesting
layer5node5 0.8 – pending
layer5node5 1 – confirmed

layer1ops_node1 1 – confirmed

root_node 1 – confirmed

END
```

References

Ackerman, S. (1992). *Discovering the brain*. National Academies Press.

Ahmad, S. (2017, December 18). Capsules Close-up: Comparing the latest deep learning advance with HTM Sensorimotor Theory [Web log comment]. Retrieved from https://numenta.com/blog/2017/12/18/comparing-capsules-with-htm/

Amunts, K., Ebell, C., Muller, J., Telefont, M., Knoll, A., & Lippert, T. (2016). The Human Brain Project: creating a European research infrastructure to decode the human brain. *Neuron*, *92*(3), 574-581.

Aphex34 (Artist). (2005). *Typical cnn* [Digital image]. Reproduced under CC BY-SA 4.0. Retrieved from Wikimedia Commons website: https://commons.wikimedia .org/w/index.php?curid=45679374

Arsalidou, M., & Taylor, M. J. (2011). Is 2+ 2= 4? Meta-analyses of brain areas needed for numbers and calculations. *Neuroimage*, *54*(3), 2382-2393.

Azam, F. (2000). *Biologically inspired modular neural networks.* (Doctoral dissertation, Virginia Tech).

Bach, J. (2003). *The MicroPsi Agent Architecture*. Proceedings of ICCM-5, International Conference on Cognitive Modeling, Bamberg, Germany, 15-20

Bach, J. (2009). *Principles of synthetic intelligence PSI: an architecture of motivated cognition* (Vol. 4). Oxford University Press.

Bach, J., & Herger, P. (2015). Request confirmation networks for neuro-symbolic script execution. In T.R. Besold, A.D. Garcez, G.F. Marcus & R. Miikkulainen (Eds.), *COCO'15 Proceedings of the 2015th International Conference on Cognitive Computation: Integrating Neural and Symbolic Approaches*, *1583* (pp. 43-51). Aachen: CEUR-WS.

Barnett, D., Bauer, A., Bell, S., Elliott, N., Haski, H., Barkley, E., ... & Mackiewicz, K. (2007). Preschool intervention scripts: Lessons from 20 years of research and practice. *The Journal of Speech and Language Pathology–Applied Behavior Analysis*, *2*(2), 158.

Belinkov, Y., Màrquez, L., Sajjad, H., Durrani, N., Dalvi, F., & Glass, J. (2017). Evaluating Layers of Representation in Neural Machine Translation on Part-of-Speech and Semantic Tagging Tasks. In *Proceedings of the Eighth International Joint Conference on Natural Language Processing (Volume 1: Long Papers)* (Vol. 1, pp. 1-10).

Benenson R. (2017) Classification datasets results [Github Repository] Retrieved from http://rodrigob.github.io/are_we_there_yet/build/classification_datasets_results.html#4d4e495354

Bengio, Y., Lee, D. H., Bornschein, J., Mesnard, T., & Lin, Z. (2015). Towards biologically plausible deep learning. *arXiv:1502.04156*.

Bergmer, C. (Artist). (2005). *A diagram of an artificial neuron* [Digital image]. Reproduced under CC BY-SA 4.0. Retrieved from Wikimedia Commons website: https://commons.wikimedia.org/wiki/File:ArtificialNeuronModel_english.png.

BiObserver (Artist). (2015) *Long short-term memory units* [Digital image]. Reproduced under CC BY-SA 4.0. Retrieved from Wikimedia Commons website: https://commons.wikimedia.org/wiki/File:Long_Short_Term_Memory.png

BRAIN Initiative. (n.d.). *The Brain Initiative Mission*. Retrieved from http://www.braininitiative.org/mission/

Brette, R., Rudolph, M., Carnevale, T., Hines, M., Beeman, D., Bower, J. M., ... & Zirpe, M. (2007). Simulation of networks of spiking neurons: a review of tools and strategies. *Journal of Computational Neuroscience*, *23*(3), 349-398.

von Bubnoff, A. (2017, September 20). A Brain Built From Atomic Switches Can Learn. *Quanta Magazine.* Retrieved from https://www.quantamagazine.org/a-brain-built-from-atomic-switches-can-learn-20170920/

Buschman, T. J., & Miller, E. K. (2007). Top-down versus bottom-up control of attention in the prefrontal and posterior parietal cortices. *Science*, *315*(5820), 1860-1862.

Buxhoeveden, D. P., & Casanova, M. F. (2002). The minicolumn hypothesis in neuroscience. *Brain*, 125(5), 935-951.

Buxhoeveden, D. P., Switala, A. E., Litaker, M., Roy, E., & Casanova, M. F. [Digital image] (2001). Lateralization of minicolumns in human planum temporale is absent in nonhuman primate cortex. *Brain, Behavior and Evolution*, *57*(6), 349-358. Copyright © 2001 Karger Publishers, Basel, Switzerland.

Cantlon, J. F., Brannon, E. M., Carter, E. J., & Pelphrey, K. A. (2006). Functional imaging of numerical processing in adults and 4-y-old children. *PLoS Biology*, *4*(5), e125.

Carlson, N.A. (1992). *Foundations of Physiological Psychology*. Needham Heights, Massachusetts: Simon & Schuster.

Carter, H.V. (Artist). (n.d.). *Principal fissures and lobes of the cerebrum viewed laterally.* [Digital image]. Reproduced under public domain. Retrieved from Wikimedia Commons website: https://commons.wikimedia.org/wiki/File:Lobes_of_the_brain_NL.svg.

Cepelewicz, J. (2016, March 8). The U.S. Government Launches a $100-Million "Apollo Project of the Brain". *Scientific American*. Retrieved from http://www.scientificamerican.com/article/the-u-s- government-launches-a-100 -million-apollo-project-of-the-brain/

Chen, P., & Goedert, K. M. (2012). Clock drawing in spatial neglect: A comprehensive analysis of clock perimeter, placement, and accuracy. *Journal of Neuropsychology*, *6*(2), 270-289.

Clancy, K. (2017, February 15). A Computer to Rival the Brain. *The New Yorker.* Retrieved from https://www.newyorker.com/tech/elements/a-computer-to-rival-the-brain

Colyer, A. (2017, October 24). Continuous online sequence learning with an unsupervised neural network model. *The Morning Paper.* Retrieved from https://blog.acolyer.org/2017/10/24/continuous-online-sequence-learning-with-an -unsupervised-neural-network-model/

Cox, D. D., & Dean, T. (2014). Neural networks and neuroscience-inspired computer vision. *Current Biology*, *24*(18), R921-R929.

CNBC (Producer). (2016, June 1). Bill Gates on Artificial Intelligence [Video file]. Retrieved from https://www.cnbc.com/video/2016/06/01/bill-gates-on-artificial -intelligence.html

Cosentino, S., Chute, D., Libon, D., Moore, P., & Grossman, M. (2006). How does the brain support script comprehension? A study of executive processes and semantic knowledge in dementia. *Neuropsychology* , 20(3), 307.

Cruz, L., Buldyrev, S. V., Peng, S., Roe, D. L., Urbanc, B., Stanley, H. E., & Rosene, D. L. (2005). A statistically based density map method for identification and quantification of regional differences in microcolumnarity in the monkey brain. *Journal of Neuroscience Methods*, *141*(2), 321-332.

Cui, Y., Ahmad, S., Hawkins, J. (2016). Continuous Online Sequence Learning with an Unsupervised Neural Network Model. *Neural Computation* 28 (11): 2474-2504.

Dehaene, S., Spelke, E., Pinel, P., Stanescu, R., & Tsivkin, S. (1999). Sources of mathematical thinking: Behavioral and brain-imaging evidence. *Science*, *284*(5416), 970-974.

Deiber, M. P., Passingham, R. E., Colebatch, J. G., Friston, K. J., Nixon, P. D., & Frackowiak, R. S. J. (1991). Cortical areas and the selection of movement: a study with positron emission tomography. *Experimental brain research*, *84*(2), 393-402.

Diehl, P. U., & Cook, M. (2015). Unsupervised learning of digit recognition using spike-timing-dependent plasticity. *Frontiers in computational neuroscience*, *9*, 99.

DiMaggio, P (1997). "Culture and cognition". *Annual Review of Sociology*. 23: 263–287.

Docksai, R. (2017, August 10). 'Brain-Like' Supercomputers Could Enable Better Defense Decision-Making. *U.S. Department of Defense News*.

Dörner, D., & Güss, C. D. (2013). PSI: A computational architecture of cognition, motivation, and emotion. *Review of General Psychology*, *17*(3), 297.

Eckersley, P., Nasser, Y., et al. (2017). EFF AI Progress Measurement Project. Retrieved from https://www.eff.org/ai/metrics

Eger, E. (2016). Neuronal foundations of human numerical representations. In *Progress in brain research* (Vol. 227, pp. 1-27). Elsevier.

Eliasmith, C., Stewart T. C., Choo X., Bekolay T., DeWolf T., Tang Y., Rasmussen, D. (2012). A large-scale model of the functioning brain. *Science*. Vol. 338 no. 6111 pp. 1202-1205.

Favorov, O. V., & Kelly, D. G. (1994). Minicolumnar organization within somatosensory cortical segregates: I. Development of afferent connections. *Cerebral Cortex*, *4*(4), 408-427.

Fernando, C., Banarse, D., Blundell, C., Zwols, Y., Ha, D., Rusu, A. A., ... & Wierstra, D. (2017). Pathnet: Evolution channels gradient descent in super neural networks. *arXiv:1701.08734*.

Frith, C. D., Friston, K. J., Liddle, P. F., & Frackowiak, R. S. (1991, June). Willed action and the prefrontal cortex in man: a study with PET. In *Proc. R. Soc. Lond. B* (Vol. 244, No. 1311, pp. 241-246). The Royal Society.

Fukushima, K. (1980). Neocognitron: A self-organizing neural network model for a mechanism of visual pattern recognition. *Biol. Cybernetics* 36, 193-202.

de Garis, H., Chen, S., Goertzel, B., & Ruiting, L. (2010). A world survey of artificial brain projects, Part I: Large-scale brain simulations. *Neurocomputing*, 74(1), 3-29.

Gilbert, C. D., & Sigman, M. (2007). Brain states: top-down influences in sensory processing. *Neuron*, *54*(5), 677-696.

Glasser, M. F., Coalson, T. S., Robinson, E. C., Hacker, C. D., Harwell, J., Yacoub, E., ... & Smith, S. M. (2016). A multi-modal parcellation of human cerebral cortex. *Nature*, *536*(7615), 171-178.

Goertzel, B., Lian, R., Arel, I., De Garis, H., & Chen, S. (2010). A world survey of artificial brain projects, Part II: Biologically inspired cognitive architectures. *Neurocomputing*, *74*(1), 30-49.

Goldberg, J. H., Farries, M. A., & Fee, M. S. (2013). Basal ganglia output to the thalamus: still a paradox. *Trends in neurosciences*, *36*(12), 695-705.

Goodale, M., & Milner, D. (2013). *Sight unseen: An exploration of conscious and unconscious vision*. OUP Oxford.

Gorman, J. (2014). Learning How Little We Know About the Brain. *The New York Times*. Retrieved from https://www.nytimes.com/2014/11/11/science/learning-how-little-we-know-about-the-brain.html?mtrref=www.google.com

Gruber, O., Indefrey, P., Steinmetz, H., & Kleinschmidt, A. (2001). Dissociating neural correlates of cognitive components in mental calculation. *Cerebral cortex*, *11*(4), 350-359.

Hay, J. C., Lynch, B. E., & Smith, D. R. (1960). *Mark I Perceptron Operators' Manual* (No. VG-1196-G-5). Cornell Aeronautical Laboratory, Inc. Buffalo, NY.

Hawkins, J., Ahmad, S., & Cui, Y. (2017). Why Does the Neocortex Have Layers and Columns, A Theory of Learning the 3D Structure of the World. *bioRxiv*, 162263.

Hermundstad, A., Brown, K., Bassett, D., & Carlson, J. (2011, November). Structural drivers of function in information processing networks. In *Signals, Systems and Computers (ASILOMAR), 2011 Conference Record of the Forty Fifth Asilomar Conference on* (pp. 837-841). IEEE.

Hernandez, D. (2015, February 26). The thinking computer that was supposed to colonize space. *Splinter*. Retrieved from https://splinternews.com/the-thinking-computer-that-was-supposed-to-colonize-spa-1793845705

Hinton, G. E., Krizhevsky, A., & Wang, S. D. (2011, June). Transforming auto-encoders. In *International Conference on Artificial Neural Networks* (pp. 44-51). Springer, Berlin, Heidelberg.

Horton, J. C., & Adams, D. L. (2005). The cortical column: a structure without a function. *Philosophical Transactions of the Royal Society of London B: Biological Sciences*, *360*(1456), 837-862.

Hubel, D. H., & Wiesel, T. N. (1962). Receptive fields, binocular interaction and functional architecture in the cat's visual cortex. *The Journal of physiology*, *160*(1), 106-154.

Human Brain Project. (2017, December 12). *World's Brain Initiatives Move Forward Together* [Press release]. Retrieved from https://www.humanbrainproject.eu/en/follow-hbp/news/worlds-brain-initiatives-move-forward-together/

Iijima, M., & Nishitani, N. (2017). Cortical dynamics during simple calculation processes: a magnetoencephalography study. *Clinical Neurophysiology Practice*, *2*, 54-61.

Isokawa, T., Nishimura, H., & Matsui, N. (2012). Quaternionic multilayer perceptron with local analyticity. *Information*, *3*(4), 756-770.

Izhikevich, E.M. (2005). Simulation of Large-Scale Brain Models [Web log comment]. Retrieved from www.nsi.edu/users/izhikevich/interest/index.htm.

Jabr, F. (2012). Does thinking really hard burn more calories. *Scientific American*, *18*.

Johansson, C., & Lansner, A. (2007). Towards cortex sized artificial neural systems. *Neural Networks*, *20*(1), 48-61.

Jones EG. (2000). Microcolumns in the cerebral cortex. *Proc Natl Acad Sci* 97(10): 5019–21.

Koechlin, E., Basso, G., Pietrini, P., Panzer, S., & Grafman, J. (1999). The role of the anterior prefrontal cortex in human cognition. *Nature*, *399*(6732), 148.

Koene, R. A., & Hasselmo, M. E. (2005). An integrate-and-fire model of prefrontal cortex neuronal activity during performance of goal-directed decision making. *Cerebral Cortex*, *15*(12), 1964-1981.

Knight, W. (2016, December 7). AI Winter Isn't Coming. *MIT Technology Review.* Retrieved from https://www.technologyreview.com/s/603062/ai-winter-isnt -coming/

Kumar, S., Strachan, J. P., & Williams, R. S. (2017). Chaotic dynamics in nanoscale NbO 2 Mott memristors for analogue computing. *Nature*, *548*(7667), 318.

Lam, V. (2016). "We know very little about the brain": Experts outline challenges in neuroscience. *Scope*. Retrieved from http://scopeblog.stanford.edu/2016/11/08 /challenges-in-neuroscience-in-the-21st-century/

Lamb, S. (2000). The Columnar Level. In *Language and Brain: Neurocognitive Linguistics* (An Adaptive Neural Network: the Cerebral Cortex). Retrieved from http://www.ruf.rice.edu/~lngbrain/Farh/col.html

LeCun, Y. (1998). The MNIST database of handwritten digits [Data file]. Available from http://yann. lecun. com/exdb/mnist/.

LeCun, Y., Bengio, Y., & Hinton, G. (2015). Deep learning. *Nature*, *521*(7553), 436.

Lee, J. H., Delbruck, T., & Pfeiffer, M. (2016). Training deep spiking neural networks using backpropagation. *Frontiers in neuroscience*, *10*, 508.

Livingstone, M. S., Pettine, W. W., Srihasam, K., Moore, B., Morocz, I. A., & Lee, D. (2014). Symbol addition by monkeys provides evidence for normalized quantity coding. *Proceedings of the National Academy of Sciences*, *111*(18), 6822-6827.

Löwel, S., & Singer, W. (1992). Selection of intrinsic horizontal connections in the visual cortex by correlated neuronal activity. *Science*, *255*(5041), 209-212.

Lücke, J., & Malsburg, C. V. D. (2004). Rapid processing and unsupervised learning in a model of the cortical macrocolumn. *Neural Computation*, *16*(3), 501-533.

Maass, W. (1997). Networks of spiking neurons: the third generation of neural network models. *Neural networks*, *10*(9), 1659-1671.

Maheswaranathan, N., Ferrari, S., VanDongen, A. M., & Henriquez, C. (2012). Emergent bursting and synchrony in computer simulations of neuronal cultures. *Frontiers in computational neuroscience*, *6*, 15.

Marblestone, A. H., Wayne, G., & Kording, K. P. (2016). Toward an integration of deep learning and neuroscience. *Frontiers in Computational Neuroscience*, *10*, 94.

Markram, H. (2006). The blue brain project. *Nature Reviews Neuroscience*, *7*(2), 153-160.

Martin, S. J., Grimwood, P. D., & Morris, R. G. (2000). Synaptic plasticity and memory: an evaluation of the hypothesis. *Annual review of neuroscience*, *23*(1), 649-711.

Martinet, L. E., Sheynikhovich, D., Benchenane, K., & Arleo, A. (2011). Spatial learning and action planning in a prefrontal cortical network model. *PLoS computational biology*, *7*(5), e1002045.

McCarthy, J., Minsky, M. L., Rochester, N., & Shannon, C. E. (1955). A Proposal for the Dartmouth Summer Research Project on Artificial Intelligence. Retrieved from http://jmc.stanford.edu/articles/dartmouth/dartmouth.pdf

McCulloch, W. S., & Pitts, W. (1943). A logical calculus of the ideas immanent in nervous activity. *The bulletin of mathematical biophysics*, *5*(4), 115-133.

Merolla, P. A., Arthur, J. V., Alvarez-Icaza, R., Cassidy, A. S., Sawada, J., Akopyan, F., ... & Modha, D. (2014). A million spiking-neuron integrated circuit with a scalable communication network and interface. *Science*, *345*(6197), 668-673.

Minsky, M., Papert, S. A., & Bottou, L. (2017). *Perceptrons: An introduction to computational geometry*. MIT press.

Miyashita, Y., & Hayashi, T. (2000). Neural representation of visual objects: encoding and top-down activation. *Current opinion in neurobiology*, *10*(2), 187-194.

Montague, P. R., Eagleman, D. M., McClure, S. M., & Berns, G. S. (2006). Reinforcement Learning: A Biological Perspective. *Encyclopedia of Cognitive Science*.

Mountcastle, V. B. (1997). The columnar organization of the neocortex. *Brain: a journal of neurology*, *120*(4), 701-722.

Nano, X. (2016). Handwritten math symbols dataset [Data file]. Available from Kaggle: https://www.kaggle.com/xainano/handwrittenmathsymbols

National Institutes of Health (2014). *NIH embraces bold, 12-year scientific vision for BRAIN Initiative* [Press release]. Retrieved from https://www.nih.gov/news -events/news-releases/nih-embraces-bold-12-year-scientific-vision-brain-initiative

Nieder, A. (2016). The neuronal code for number. *Nature Reviews Neuroscience*, *17*(6), 366.

Nieder, A., & Dehaene, S. (2009). Representation of number in the brain. *Annual review of neuroscience*, *32*, 185-208.

Numenta (2012). Modeling Data Streams Using Sparse Distributed Representations [Video file]. Retrieved from http://www.cortical.io/resources_media.html#video -303

O'Reilly, R. C., & Munakata, Y. (2000). *Computational explorations in cognitive neuroscience: Understanding the mind by simulating the brain*. MIT press.

Painkras, E., Plana, L. A., Garside, J., Temple, S., Galluppi, F., Patterson, C., ... & Furber, S. B. (2013). SpiNNaker: A 1-W 18-core system-on-chip for massively-parallel neural network simulation. *IEEE Journal of Solid-State Circuits*, *48*(8), 1943-1953.

Rajpurkar, P., Irvin, J., Zhu, K., Yang, B., Mehta, H., Duan, T., ... & Lungren, M. P. (2017). CheXNet: Radiologist-Level Pneumonia Detection on Chest X-Rays with Deep Learning. *arXiv:1711.05225*.

Reimann, M. W., Nolte, M., Scolamiero, M., Turner, K., Perin, R., Chindemi, G., ... & Markram, H. (2017). Cliques of Neurons Bound into Cavities Provide a Missing Link between Structure and Function. *Frontiers in Computational Neuroscience*, *11*, 48.

Richert, M., Fisher, D., Piekniewski, F., Izhikevich, E. M., & Hylton, T. L. (2016). Fundamental principles of cortical computation: unsupervised learning with prediction, compression and feedback. *arXiv:1608.06277*.

Rosenblatt, F. (1958). The Perceptron: A Probabilistic Model for Information Storage and Organization in the Brain. *Psychological Review*, v65, No. 6, pp. 386–408.

Sabour, S., Frosst, N., & Hinton, G. E. (2017). Dynamic routing between capsules. In *Advances in Neural Information Processing Systems* (pp. 3859-3869).

Sandberg, A., Lansner, A., Petersson, K. M., & Ekeberg. (2002). A Bayesian attractor network with incremental learning. *Network: Computation in neural systems*, *13*(2), 179-194.

Samuel, A. L. (1959). Some studies in machine learning using the game of checkers. *IBM Journal of Research and Development*, *3*(3), 210-229.

Sardi, S., Vardi, R., Sheinin, A., Goldental, A., & Kanter, I. (2017). New Types of Experiments Reveal that a Neuron Functions as Multiple Independent Threshold Units. *Scientific Reports*, *7*(1), 18036.

Scharnhorst, K., Woods, W., Teuscher, C., Stieg, A., & Gimzewski, J. (2017, July). Non-temporal logic performance of an atomic switch network. In *2017 IEEE/ACM International Symposium on Nanoscale Architectures (NANOARCH)* (pp. 133-138). IEEE.

Schuman, C. D., Potok, T. E., Patton, R. M., Birdwell, J. D., Dean, M. E., Rose, G. S., & Plank, J. S. (2017). A survey of neuromorphic computing and neural networks in hardware. *arXiv:1705.06963*.

Sengupta, A., Ye, Y., Wang, R., Liu, C., & Roy, K. (2018). Going Deeper in Spiking Neural Networks: VGG and Residual Architectures. *arXiv:1802.02627.*

Shu, Y., Hasenstaub, A., & McCormick, D. A. (2003). Turning on and off recurrent balanced cortical activity. *Nature*, *423*(6937), 288.

Silver, D., Hubert, T., Schrittwieser, J., Antonoglou, I., Lai, M., Guez, A., ... & Lillicrap, T. (2017). Mastering Chess and Shogi by Self-Play with a General Reinforcement Learning Algorithm. *arXiv:1712.01815*.

Simon, D. T., Larsson, K. C., Nilsson, D., Burström, G., Galter, D., Berggren, M., & Richter-Dahlfors, A. (2015). An organic electronic biomimetic neuron enables auto-regulated neuromodulation. *Biosensors and Bioelectronics*, *71*, 359-364.

Simonite, T. (2015, April 8). IBM Tests Mobile Computing Pioneer's Controversial Brain Algorithms. *MIT Technology Review*. Retrieved from https://www.technologyreview.com/s/536326/ibm- tests-mobile-computing -pioneers-controversial-brain-algorithms/

Singer, A. (2016, April 6). Mapping the Brain to Build Better Machines. *Quanta Magazine.* Retrieved from https://www.quantamagazine.org/mapping-the-brain -to-build-better-machines-20160406/

Solms, M. (1997). *The neuropsychology of dreams: A clinico-anatomical study*. L. Erlbaum.

Strukov, D. B., Snider, G. S., Stewart, D. R., & Williams, R. S. (2008). The missing memristor found. *Nature*, *453*(7191), 80.

Taherkhani, A., Belatreche, A., Li, Y., & Maguire, L. P. (2014, April). A new biologically plausible supervised learning method for spiking neurons. In *ESANN 2014 proceedings, European Symposium on Artificial Neural Networks, Computational Intelligence and Machine Learning.* Bruges (Belgium): i6doc.com

Tanji, J., & Hoshi, E. (2001). Behavioral planning in the prefrontal cortex. *Current opinion in neurobiology*, *11*(2), 164-170.

TensorFlow. (2015). MNIST for ML Beginners [Digital image]. Retrieved from https://tensorflow.rstudio.com/tensorflow/articles/tutorial_mnist_beginners.html

Tian, Y., Pei, K., Jana, S., & Ray, B. (2017). DeepTest: Automated testing of deep-neural-network-driven autonomous cars. *arXiv:1708.08559*.

Tomita, H., Ohbayashi, M., Nakahara, K., Hasegawa, I., & Miyashita, Y. (1999). Top-down signal from prefrontal cortex in executive control of memory retrieval. *Nature*, *401*(6754), 699.

Vaillant, R., Monrocq, C., & Le Cun, Y. (1994). Original approach for the localisation of objects in images. *IEE Proceedings-Vision, Image and Signal Processing*, *141*(4), 245-250.

Weng, S. F., Reps, J., Kai, J., Garibaldi, J. M., & Qureshi, N. (2017). Can machine-learning improve cardiovascular risk prediction using routine clinical data? *PloS one*, *12*(4), e0174944.

Whittington, J. C., & Bogacz, R. (2017). An approximation of the error backpropagation algorithm in a predictive coding network with local Hebbian synaptic plasticity. *Neural computation*, *29*(5), 1229-1262.

Woodruff, G., & Premack, D. (1981). "Primative mathematical concepts in the chimpanzee: proportionality and numerosity". *Nature*. 293 (5833): 568–570.

Yang, G. R., Murray, J. D., & Wang, X. J. (2016). A dendritic disinhibitory circuit mechanism for pathway-specific gating. *Nature communications*, *7*, 12815.

Yger, P., & Harris, K. D. (2013). The Convallis rule for unsupervised learning in cortical networks. *PLoS Computational Biology*, *9*(10), e1003272.

Yovits, M. C. (1993). *Advances in Computers* (Vol. 37). Academic Press.

Zhang, X., Xu, Z., Henriquez, C., & Ferrari, S. (2013, December). Spike-based indirect training of a spiking neural network-controlled virtual insect. In *Decision and Control (CDC), 2013 IEEE 52nd Annual Conference on* (pp. 6798-6805). IEEE.

Glossary

Action potential An electrical impulse that travels down the axon as neuronal output.

Afferent Channels that conduct substances or information into (vs. out of) a system.

Artificial General Intelligence (AGI) The theorized capacity of a machine to perform all intellectual tasks at or beyond a human level.

Artificial Neural Network (ANN) Computing systems wherein nodes are connected by weighted links, used to perform learning or more generally simulate the structure and function of biological neural networks.

Autoencoder An ANN whose output is a reconstruction of its input.

Backpropagation A training technique for ANNs that proportionally distributes credit for prediction success or error backwards through the network.

Biomimetic The quality of emulating biology.

Bit shift A bitwise operation in which digits are shifted to the left or right, resulting in a new value for the series of bits that have been operated on. Can be used to perform arithmetic computations.

Bottom-up activation Denotes the flow of activation from lower-level representations, generally perceptual input from sensory modalities, to higher-level representations.

Capsule A group of neurons whose outputs represent different instantiation parameters of the same entity.

Categorical cross entropy An algorithm to define a loss function by comparing the probability distribution of the predicted values against the true probability.

Chain rule A formula to calculate the derivative of the composition of two functions.

Composition operator An operator that maps functions to functions.

Connectionist A.I. Large networks of simple units connected by tunable links that have become synonymous with modern machine learning.

Convolution A mathematical operation that produces the integral of the product of two functions after one is translated. In CNNs, used to calculate the correspondence of a feature to a selected part of the image.

Convolutional Neural Network (CNN) An ANN that alternates feature extracting convolutional layers with sub-sampling layers that reduce input dimensions. CNNs dominate the field of image analysis.

Cortical column A vertical module in the cortex proposed by Vernon Mountcastle, suggested to be composed of minicolumns.

Deep learning Learning performed by ANNs with many layers of artificial neurons.

Dynamic routing An algorithm used in capsule networks to determine the most consistent interpretation of information between competing capsules.

Encoder A technique that encodes an input sequence to an internal representation.

Feature maps Matrixes that capture regions with salient extracted features.

Feedforward Denotes that input data moves through the network in only one direction.

Filter A matrix of weights convolved over a larger input matrix in a CNN for learned feature extraction; also known as a "kernel".

Flow modules Individual computational units in MESH that act as groups of nodes.

Fully-connected Refers to a network, or network layer, where all nodes are connected to all other nodes.

Gate A component of an artificial neuron from which output is passed to a link.

Gradient descent An optimization algorithm that seeks the parameters that best minimize an objective function (in backpropagation, generally the loss function).

Grammar Very generally, a set of production rules for symbolic output.

Hebbian learning A type of unsupervised learning where connections between neurons are strengthened based solely on correlated firing between the neurons they connect. Often paraphrased as "neurons wire together if they fire together."

HTM A theory of intelligence developed by Jeff Hawkins that serves as the basis for machine learning technology.

Identity function A non-transformative function, denoted by $x = x$.

Kernel See "Filter".

Link A connection, often weighted, between two network units.

Lobe A specialized region of the cerebrum.

Long Short-Term Memory (LSTM) A type of RNN where state is managed by a series of gates, allowing the network to maintain a representation over time of its past input sequences without interference from new inputs. LSTMs have been field-defining in speech recognition and language processing.

Loss function A function that calculates a penalty to a learning system, generally based on the difference between the expected output and the system's prediction.

Machine learning The field of computer science concerned with giving machines the ability to autonomously draw increasingly accurate predictions from prior experience.

McCulloch-Pitts model A simple linear threshold neuron model developed in 1943 by Warren McCulloch and Walter Pitts based on observations from theoretical neurophysiology.

Membrane potential The voltage difference between a neuron's interior and its exterior environment that gates its inclination to fire.

Memristor A circuit element commonly used in neuromorphic devices that emulates the plasticity and timing dependence of synapses in the brain.

MESH A graphical editor developed by MicroPsi Industries for instantiating MicroPsi agent environments and running neural network agents.

MicroPsi A cognitive architecture based on Dietrich Dörner’s PSI theory designed to provide a framework in which to run and test ANN-based agents.

Minecraft An online video game environment useful for artificial agent exploration.

Minicolumn A vertically interconnected group of ~80-100 neurons thought to act as basic computational units in the cortex.

MNIST A database of handwritten digits developed by Yan LeCun, commonly used to train and test machine learning systems.

Multilayer perceptron (MLP) A fully-connected feedforward ANN with one or more hidden layers, typically trained by backpropagation.

Neocognitron The ANN predecessor to CNNs, inspired by research on receptive fields in the visual cortex.

Neocortex The largest region and outer layer of the human cerebral cortex, associated with higher-order cognitive functions.

Neuro-symbolic Denotes the representation of symbolic knowledge in the brain.

Neuromorphic hardware Biologically inspired devices that can be digital, analog, or hybrid. Famous implementations include the University of Manchester's SpiNNaker and IBM's TrueNorth.

Neuron An electrically excitable cell that serves as the basic unit of the nervous system.

Neuroplasticity The dynamic and continuous adaptability of the brain.

Node A network unit that emulates a biological neuron.

Normalization Restructuring data to make it more internally regular.

Numerosity The perceived magnitude of a stimulus.

Perceptron A linear classifier developed in 1957 by Frank Rosenblatt as an extension of the McCulloch-Pitts neuron model.

Pooling A type of sub-sampling. A favored pooling algorithm in CNNs is max pooling, which selects the largest value within the filter for the corresponding position in the feature map.

Pose The spatial relationship between features, relevant to image analysis.

Receptive field The region of a visual field to which a particular neuron is sensitive.

Recurrency Denotes the presence of directed cycles.

Recurrent Neural Network (RNN) ANNs that incorporate directed cycles.

Reinforcement learning A learning paradigm that focuses on encouraging behaviors that maximize a reward function rather than penalizing suboptimal decisions with a loss function.

Request Confirmation Network (ReCoN) Hierarchical spreading activation networks with constrained top-down/bottom-up recurrency proposed as a possible model for cortical activity during execution of neuro-symbolic sensorimotor scripts.

Request confirmation The protocol governing the spread of ReCoN activation.

Schema A neuro-symbolic representation of objects or events that organizes categories of information and relationships between them.

Script A category of sensorimotor schema that encodes key sequences of events and the relevant actions necessary to fulfil behavioral routines and activities.

Semantic Relating to the meaning or interpretation of logic.

Sensorimotor Involving both the sensory and motor pathways.

Shift-invariance The quality of robustness against (minor) displacements in input, generally used in association with CNNs.

Slot A component of an artificial neuron into which input arrives.

Softmax A function used to force values in an array to sum to zero for probability distribution, with the largest array value indicating the strongest probability.

Sparse Distributed Representation (SDR) Used in HTM, an array of bits where each bit index has a specific semantic meaning encoding a feature of the input data.

Spike-Timing Dependent Plasticity (STDP) A biologically plausible learning paradigm, emulated in SSNs, that locally adjusts link weights based on the relative timing between a neuron receiving activation and firing.

Spike The short voltage increase output by a firing neuron. Depending on the literature, can be considered analogous to "action potential".

Spiking neural network (SSN) An ANN that emulates the spiking nature of biological neurons.

Stack An abstract data type that represents a last in, first out ordered collection of elements.

Stacked calculator A calculator that implements Reverse Polish notation, where operands follow their operators.

Step function A function with a value that is constant during given intervals and changes with each interval.

Sub-sampling A process employed by CNNs to reduce input dimensionality.

Supervised learning A learning paradigm wherein accurate labels are provided alongside the input data.

Symbolic A.I. Networks that focus on developing symbolic representations and associations that can serve as the basis for general knowledge acquisition.

Synapse A communication interface between connected neurons.

Threshold An internal variable in an artificial neuron that mimics membrane potential in a biological neuron, preventing the neuron from firing until the threshold value has been met.

Top-down activation Denotes the flow of activation from higher-level representations to lower-level representations, typically but not necessarily as the result of intentional direction.

Unsupervised learning A learning paradigm wherein a network converges on a function based on statistical regularities in unlabeled input data.

Weight The relative contribution of a particular input. In an ANN, this is typically represented as a coefficient on a link by which passing activation is multiplied.