Biologically Inspired Cognitive Architectures (2015) 11, 75–105

Available at www.sciencedirect.com

ScienceDirect

journal homepage: www.elsevier.com/locate/bica

NeurOS™ and NeuroBlocks™ a neural/cognitive operating system and building blocks ☆

Lee Scheffler *

Cognitivity™, 77 Fairway Drive, West Newton, MA 02465, United States

Received 7 November 2014; accepted 7 November 2014

KEYWORDS
Cognition;
Perception;
Pattern recognition;
Memory;
Learning;
Behavior

Abstract
NeurOS is an open platform for accelerating research, development and hosting execution of intelligent applications. A NeurOS application is a directed ''neural graph'' of modular components connected by signal paths, similar to biological brain connectivity and functional block diagrams of neural pathways. Built-in reusable modules (NeuroBlocks) provide a wide range of general- and special-purpose capabilities: inputs/senses, outputs/effectors, processing, memory, pattern learning and recognition, visualization/instrumentation, custom module development, integrating external intelligence capabilities, and sub-graph reuse. NeurOS sub-graph assemblies address neural/cognitive functions including perception, pattern learning and recognition, working memory, imagination, prediction, context priming, attention, abstraction, classification, associational thinking and behavior. NeurOS applications are inherently portable, scalable, networkable, extensible and embeddable. NeurOS development tools provide simple intuitive graphical drag and drop application assembly from components without programming, along with testing, debugging, monitoring and visualization. Prototype NeurOS applications have begun to explore a wide range of intelligent functions in diverse areas, including aspects of pattern recognition, vision, music, reading, puzzle solving, reasoning, behavior. Building working intelligent systems using NeurOS and NeuroBlocks lets researchers and developers focus on their core functions and rapidly iterate and instrument working models, fostering both analytical and biological insight as well as usable systems.
© 2014 The Author. Published by Elsevier B.V. This is an open access article under the CC BY-NC-ND license (http://creativecommons.org/licenses/by-nc-nd/3.0/).

☆ Patents pending.

* Tel.: +1 617 331 8317.
E-mail address: scheffler@alum.mit.edu.
URL: http://www.cognitivity.technology.

http://dx.doi.org/10.1016/j.bica.2014.11.011
2212-683X/© 2014 The Author. Published by Elsevier B.V.
This is an open access article under the CC BY-NC-ND license (http://creativecommons.org/licenses/by-nc-nd/3.0/).

Overview

Goals and approach

One major research goal is reverse-engineering the brain: understand how it does what it does, at least well enough to build similar functionality. An emerging point of view is that brain architecture is made up of variations in connectivity among variations of common ''building block'' components (Marcus, 2014). Evolution's adaptations yield both commonality and diversity in how basic genomic/chemical/electrical mechanisms are configured and assembled.

NeurOS is a biologically inspired software embodiment of this viewpoint. NeurOS does not try to be a ''one-size-fits-all'' technology, but rather embraces composition of variations on multiple themes. NeurOS takes a synthetic approach: build working cognitive systems and through that lens, understand how biological brains might work, and thence build better artificial analogues.[1] The faster we can iterate around this loop, the better.

NeurOS enables rapid iterative building of cognitive functions as compositions of configurable reusable biologically plausible components at an easily-grasped useful level of abstraction. Its mission is to hugely accelerate research and development and host the execution of a wide range of practical intelligent systems on a variety of platforms with transparent near-linear performance scalability. NeurOS and NeuroBlocks put the ability to build almost arbitrarily complex cognitive systems into the hands of individual researchers and developers, without requiring any specialized hardware, but enabling scalable deployment without redesign on multiple current and future platforms. NeurOS additionally serves as an open fabric to easily knit together many existing diverse cognitive technologies to produce coherent and highly functional intelligent systems.

NeurOS and NeuroBlocks

NeurOS is a neural operating system for intelligent software applications built as neural graphs composed of modules called NeuroBlocks. It embodies a non-von Neuman non-procedural data flow computing paradigm. Using a simple flow diagram and no programming beyond occasional simple expressions, one can quickly assemble, test and iterate intelligent systems from an extensible toolkit of modular components, including varieties of inputs, outputs, processing, memory, pattern recognition, visualization and instrumentation. Custom components and interfaces to external computation, intelligence, sensor and action functions are easily integrated, and NeurOS applications are easily embedded in other systems.

As in biological brains, intelligent functions emerge from the interconnection of NeurOS modules. NeurOS applications can span a wide range of capabilities, including perception, working and long-term memory, thinking and imagination, behavior, language. Like biological brains, the same core mechanisms, embodied in NeurOS modules and sub-assemblies, can be reapplied in multiple domains and layers through composable flow graph designs.

NeurOS applications are built by linking instances of modular components into a neural graph, which is a directed flow graph with loops allowed. The NeurOS Designer visual interactive design environment (IDE) simplifies this process with a drag-and-drop metaphor, and hosts numerous interactive visual input, output, monitoring, visualization and debugging modules and facilities. Useful subgraph assemblies can be captured and added to the NeurOS toolkit as reusable modules. External programs can be embedded as modules, and NeurOS applications can in turn be embedded in other systems. NeurOS might be thought of as a graphical cognitive programming language.

A notable sub-theme within NeurOS is the reusability of general-purpose long-term memory modules for set, sequential and temporal patterns. These are highly composable building blocks for complex cognitive functionality spanning perception, pattern recognition, learning, classification, prediction, imagination, reasoning and behavior. See Memory Modules.

NeurOS applications can run on supported operating system, hardware and network platforms, with NeurOS taking care of all the fussy details of scheduling, memory management, multi-threading/multi-processing and distributed operation. NeurOS applications are inherently portable to numerous diverse software and hardware environments. Thanks to dataflow principles (Sousa, 2012), NeurOS application performance can be nearly linearly scalable with available resources and distributable over multiple network topologies, using both pipelined and partitioned parallelism, without explicit parallel design and without design change.

By providing many common intelligent system functions and taking care of the mechanics of run-time execution, NeurOS lets intelligent system developers concentrate on their specific added-value components and flows and rapidly evolve and extend their designs. NeurOS applications are easily built up piecemeal and integrated incrementally. Components and whole subsystems can be re-implemented/replaced without affecting other application parts. Similar to much robotics research, making cognitive functions work in NeurOS can yield both biological and analytical insights as well as useful systems.

Illustrative example — What's That Tune?

Fig. 1 shows the NeurOS Designer tool with a simple NeurOS application. At left is a toolbox of built-in NeuroBlock modules (see Module Types). The main panel is a design canvas, showing the neural graph of the application under development, a simple What's-That-Tune? capability. The application was developed by dragging and dropping modules from the toolbox, linking them together, and adjusting parameters via pop-up property sheets (not shown).

In this application, you play a simple melody on a musical keyboard and enter a name for it on a computer keyboard. Later, you play a melody, perhaps in a different key and tempo and with mistakes or extra/missing notes, and the application incrementally recognizes similar known melodies with relative confidence scores.

[1] This perhaps echoes Braitenberg's ''downhill invention'' approach to understanding complex systems (Braitenberg, 1986, p. 21).

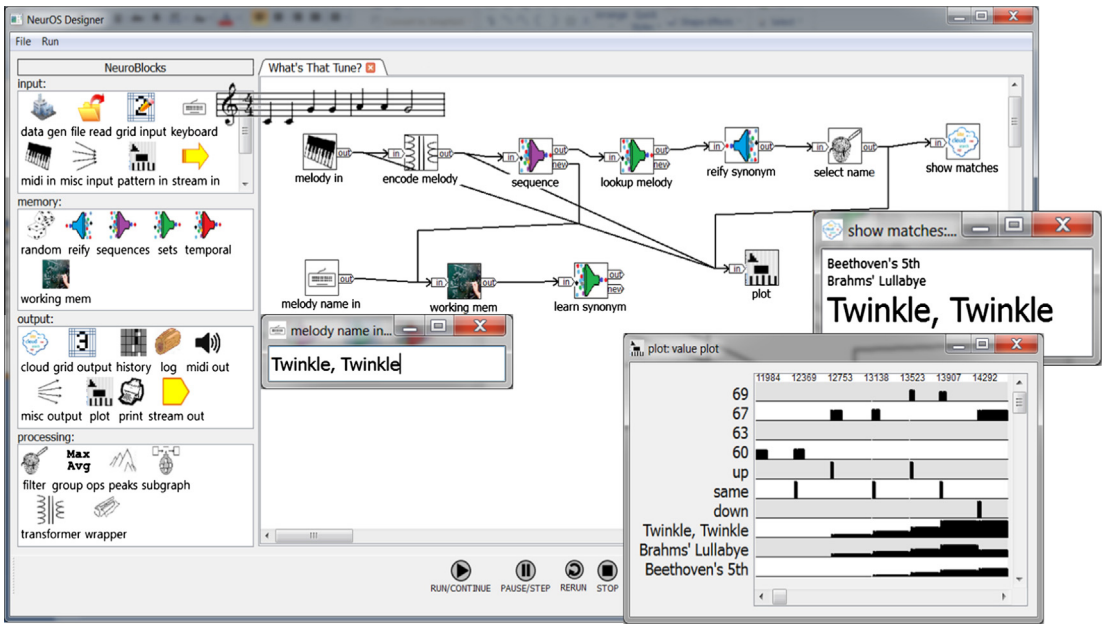


Fig. 1 What's-That-Tune? application in NeurOS.

In more detail, a simple melody is entered via the ''melody in'' MIDI[2] input module (e.g., attached to a MIDI-capable piano-style keyboard). This generates NeurOS signal events for MIDI's note-on and note-off messages (e.g., 60 is middle-C). These events are next encoded by the ''encode melody'' Transformer module into an abstract invariant representation (Hawkins & George, 2006) of the melody (relative pitch intervals of ''same'', ''up'' and ''down''[3]) as a sequence of NeurOS events. This sequence is then learned as a native NeurOS ''sequence'' pattern with a modest mistake tolerance. A name for the melody is separately entered via a computer keyboard input module. The ''working mem'' Working Memory module persists both the name and (an identifier for) the newly learned melody pattern; when they concur in time the two are associated as synonyms via a NeurOS ''set'' pattern in the ''learn synonym'' Set module.

Subsequent entry of a similar melody, possibly in a different musical key and tempo and with missing/extra/wrong notes, is recognized by the same ''sequence'' module, with increasing levels of confidence as additional notes are played. The recognized melody pattern stimulates the synonym pattern, which is then reified into its components (the name and the melody pattern identifier). The previously associated name is selected and displayed in a the ''show matches'' tag-cloud display, where a larger font indicates higher pattern-matching confidence. The EKG-like plot display over time shows the raw and encoded inputs and the progressive normalized matching scores against known melody sequence patterns as input notes are played (each bar height ranges from 0 to 1). Other previously learned melodies match the input sequence to a lesser extent. Subsequent entry of a sufficiently different melody (poorly matching already-known melody patterns) leads to creating and remembering a new melody pattern; this is how the different melodies are originally learned.

This application is not particularly impressive. Similar functionality is available in special-purpose applications like SoundHound® (SoundHound). The computations are simple and could be programmed in a few pages of specialized code. What is notable, however, is that this application can be assembled in a few minutes entirely from general-purpose NeurOS modules. These same modules and sub-assembly techniques are reusable in a wide variety of cognitive domains and applications. Replacing the MIDI input with other sequential feature inputs such as letters, words, visual feature motion or tactile sensation progressions can achieve a similar function in other domains. The application can be easily enhanced in numerous ways, for example by replacing the MIDI-input module with a more serious audio-input pre-processing sub-graph, or by adding a melody temporal sequence exemplar record and replay capability (see What's That Tune? — the enhanced version).

Organization of this paper

Section ''Perspectives and core ideas'' (Perspectives and Core Ideas) surveys the broad range of concepts and perspectives that underlie NeurOS and NeuroBlocks.

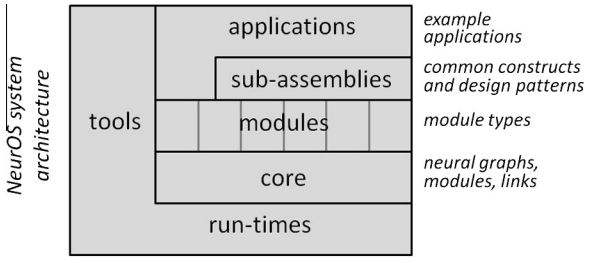


Fig. 2 Organization of this paper (and NeurOS architecture overview).

[2] Musical Instrument Digital Interface (MIDI).
[3] This is a variant of Parson's code (Parsons, 1975).

Fig. 2 shows an overview of the NeurOS system architecture, which serves to organize of the rest of this paper.

Section ''NeurOS elements: neural graphs, modules and links'' describes the core NeurOS machinery of signaling, virtual time, neural graph structure, modules and parameters. Section ''Module types'' describes a number of built-in module types. Together, these are the basic elements of NeurOS and how they fit together, metaphorically what you find when you first open the LEGO® bricks box.

Section ''NeurOS system architecture'' discusses the overall NeurOS system architecture, including development tools, run-time environments, external application representation, and extension/enhancement points. This is the fabric for putting the pieces together.

Sections ''NeurOS usage: common constructs and design patterns'' and ''Example NeurOS applications'' start building things with NeurOS. Section ''NeurOS usage: common constructs and design patterns'' surveys frequently repeated design constructs and techniques for some common neural and cognitive functions that have begun to emerge from using NeurOS. These are perhaps some generally useful ''molecules'' of cognitive function. Think of these usages like ''application notes'' for electronic components.

Section ''Example NeurOS applications'' shows several simple prototype applications that have been developed using NeurOS, especially to illustrate the breadth of application and component reusability. These are sub-systems and whole systems assembled from the pieces.

The usages and examples in Sections ''NeurOS usage: common constructs and design patterns'' and ''Example NeurOS applications'' are in no way proposed as definitive solutions to particular cognitive challenges. Their only purpose is to demonstrate the broad potential of the NeurOS approach and hopefully motivate considerably more work.

Sections ''Discussion and Status and directions'' conclude with perspectives and future directions.

Perspectives and core ideas

NeurOS is not a low-level neural simulation. In most cases it does not try to closely duplicate the operation of biological neurons and their connections. Rather, it emulates plausible abstractions of neural and cognitive functions using software rather than biological technology. Nevertheless, the structure and organization of NeurOS processing reflects the general architecture, connectivity character and dynamic behavior of biological brains. NeurOS applications perform functions in broadly similar ways to biological brains. Suggestions of plausible biological analogs of NeurOS functions and structures are interspersed throughout this paper.

NeurOS combines a number of core ideas and perspectives from different fields:

From neuroscience:

- Basic neuron and synapse functions are similar most everywhere in biological brains. Therefore, multiple cognitive/intelligent functions must emerge substantially from (a) differences among neuron types and their biochemical neighborhoods, and especially (b) how neurons are interconnected.
- Just a few neuron layers in from senses and muscles, neurons cannot distinguish between ''real'' external world inputs and inputs from other neural processes.
- Long-term memory seems rooted in creating synapses and adjusting their conductances.
- Brain connectivity is locally dense and globally sparse.
- The brain has both feed-forward and feed-back connections.
- Neuron signaling is highly bursty. Most neurons are relatively quiet most of the time.
- The brain pursues many activities in parallel; the strongest tend to have the greatest effects on subsequent processing.
- The brain continually learns and adjusts with experience.

Evolution tends to favor incremental adaptations and reuse and ''good enough'' satisficing rather than optimal solutions, but occasionally resorts to new specialized mechanisms. NeurOS application designers fill a similar role.[4] in choosing how to address cognitive challenges by configuring/tuning reusable mechanisms or introducing new mechanisms. Evolution (and for NeurOS, iterated design) provides the structure; experience programs it.

Artificial intelligence and other research domains have yielded a wealth of solutions to sub-problems in areas like vision, speech, language, pattern recognition, planning, and robotics. These beg for integration.

From engineering:

- View brains (and parts of brains, recursively down to the level of neurons, dendrites and synapses) as signal processors, transforming input signals and state into state changes and output signals.
- Multiplex handfuls of 10^9 Hz CPUs to do the work of billions of slow $\sim 10^2$ Hz neurons.
- Dataflow technology matches the locally-dense/globally sparse brain connectivity architecture well and enables scalability and parallel distributed processing. Follow dataflow principles and avoid global synchronization.
- Favor local rather than distributed memory representations.[5]
- Modularity and composability are critical to flexibility, functional and performance scalability, and good system engineering.
- Respect the core modularity architecture. All LEGO bricks, no matter what else they do, need compatible bumps and sockets to work together.
- Do not bother re-computing 0 s for brain regions with no or insignificant activity.
- Memory is cheap, fast and plentiful. Creating new memory objects dynamically as needed is fast and trivial.
- Memory addressing and indexing are very fast.

[4] I am reminded of Braitenberg's ''Vehicle 6'' processes by which developers repeatedly make small changes to surviving vehicles, and of course establish the ground rules for survival (Braitenberg, 1986, pp. 26–28).

[5] A single ''memory'' may of course derive from recombinations of many other memories. However, NeurOS generally avoids the classical neural network formulation where multiple distinct memories commingle their effects on most connection weights among most units.

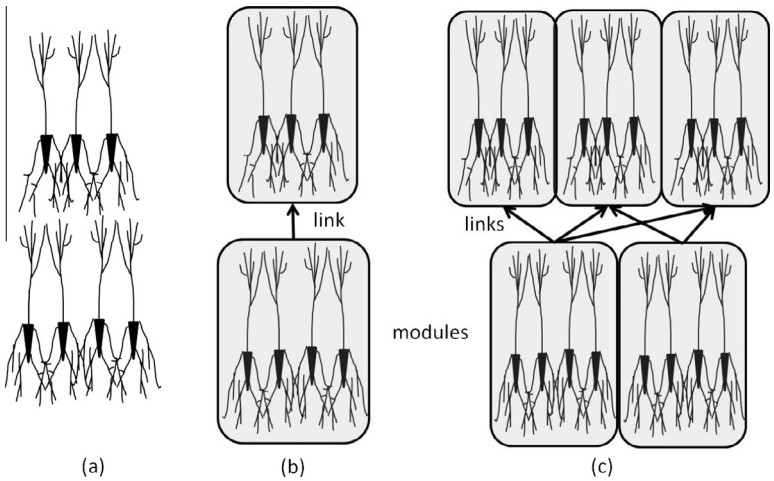


Fig. 3 Rough biological analogy for NeurOS modules and links.

- Make time virtual and explicit to avoid real-time synchronization requirements.
- Favor practicality over purity. Even if some function can be built directly as a NeurOS graph of reusable modules, it may be more efficient to use a custom implementation. Integrate/embed existing technologies as useful.
- Embrace parameterization as a mechanism to enable variations on common themes.
- Stick closely to biological brain architectures, but not too closely. Biological brains work well at what they do, so shamelessly copy architectural features where reasonable. Generally confine non-biological-analog representations and algorithms (e.g., recursion, pointers/references/variables, indices, grammars, procedures, logic) to black-box component internals.

NeurOS elements: neural graphs, modules and links

NeurOS applications are built as directed graphs of computational modules (NeuroBlocks) and directed links, with loops allowed. A module can be thought of as ''managing'' or doing the work of a group of similar neurons or neural sub-assemblies. A link carries neural signal events from one module to another.

Fig. 3 suggests a rough biological analogy. (a) depicts two stylized biological neuron collections or layers, with axons facing upward. In (b) a module performs the work of each neuron layer and a link conveys the aggregate signals between modules. NeurOS neural graphs often use multiple links for cross-connectivity among modules and the neuron functions they manage as in (c).

A module is thus one locus of CPU multiplexing over multiple emulated neuron activities. A link is thus similar to an electronic bus carrying multiplexed signal traffic from one group of neurons to another group, effectively defining the aggregate potential input field of the group of neurons modeled by a target module. One virtue of this module concept is that (virtual) neurons can be created as needed (e.g., to represent newly learned patterns) without changing the structure of a neural graph.

Neural graphs tend to resemble typical functional block diagrams, at multiple levels of detail, which have been derived from biological brain studies. Modules tend to capture distinctive functional layers, clusters and regions of neurons performing similar functions.[6] Links tend to capture distinct neural ''pathway'' segments. Neural graphs address both local and non-local brain connectivity. Neural graphs are composable and nestable, so one can ''zoom'' in and out to different levels of detail.

The core compositional interface in NeurOS is signaling via events. A NeurOS event encodes the concept that ''this neuron is spiking at this rate starting at this time''. This encoding is particularly efficient considering the bursty nature of neuron spiking and the consideration that most needed computation results from changes. This formulation enables computation based on common spike rate, spike-timing and population neural information encoding models.

Each event is a tuple (timestamp, ID, value). timestamp is a virtual time moment, used to sequence and synchronize multiple events. ID is a locally-unique signal source/path identifier. It can be thought of as a (local) neuron or axon identifier, or as a single (concrete or abstract) feature. ID allows easy commingling of multiple signals on links.

In its simplest form value is a scalar proportional to a neural spike rate, normalized to the range [0, 1] for interoperability, with 1 representing a maximum spiking rate (see below). There is, however, no requirement that event values be so normalized, as long as recipient modules can properly interpret the range/encoding. Event values can also be vectors or matrices with numeric or symbolic indices, and even arbitrary objects. These are especially useful in high-dimensional regions like low-level sensory processing.

NeurOS events, like neuron spikes, digital circuit signals and spreadsheet cells, are type-less. That is, an event can represent an arbitrary semantic, whether a perception, behavior, abstraction, deduction, recognition or any other cognitive element, determined entirely by the connections

[6] The partitioning of neurons into modules is functional rather than geometric. It seems credible that functionally distinct neuron types with overlapping input fields may be physically commingled in biological brains. See Classification and Clustering, Stereotypes and Exemplars.

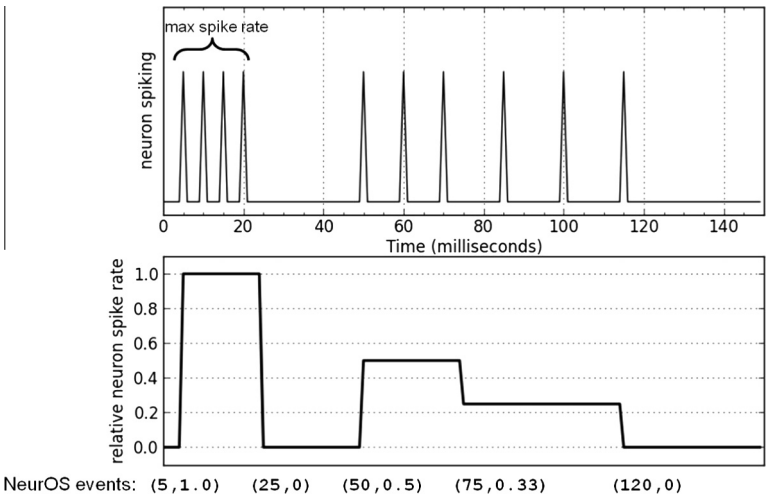


Fig. 4 NeurOS event relationships to neuron spiking.

over which they flow. This is crucial to composability, reusability and generality.

A biological neuron can be thought of as a little signal processor: its output signal (instantaneous spike rate) and internal state (activation level) is a function of the spike rates of its input signals over time. A NeurOS event value generally represents a new spike rate changed from a previous spike rate. An accurate-enough spike-rate signal could be reconstructed if needed from these edge events. Events reporting a repeat value (especially 0) are generally avoided/suppressed, since nothing has changed. Fig. 4 shows an example of an idealized neuron output spike train, the corresponding spike rates, and the corresponding NeurOS events. The 5-msec spike repeat shown is a typical maximum possible spike rate[7] and so is encoded as a NeurOS event value of 1.

A module is effectively a digital signal processor operating on multiple encoded signals in parallel. Fig. 5 shows the NeurOS module formalism.

Modules may have input ports and/or output ports (terminology as seen from inside the module). Pure system input modules (i.e., senses) have no input ports, while pure system output modules (e.g., low-level actions, displays) have no output ports. Most processing modules have one input and one or more output ports. Multiple module output ports reflect functionally distinct processing results, such as filtered feature streams derived from common inputs, and distinguishing new long-term memory pattern creation vs. known pattern matching. The choice of module input and output ports depends on the engineering intuition of the module designer, anticipating component usage and reusability. Module ports may be left unconnected if not needed in a neural graph.

This module formalism is composable and recursive. Linking an output port of one module to an input port of another module yields a composite module. The internals of a module can be implemented by composition of other

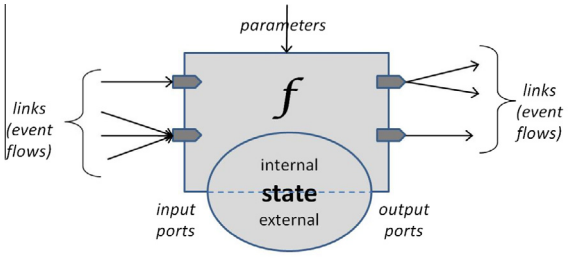


Fig. 5 NeurOS module formalism.

modules. Thus a NeurOS application is effectively a recursive composition of modules.

A link carries a stream of events in virtual time order from an output port of one module to an input port of a(nother) module. Multiple links to a single input port have their events commingled in virtual timestamp order. Multiple links from a single output port broadcast their events to multiple destinations. An important property of links is that their ''address space'' of event IDs is unconstrained. New IDs, for example, representing newly learned patterns, can flow over existing links. The ''address space'' of event IDs on a link is also local and need not relate to any other link.

Time inside NeurOS is virtual and monotonically nondecreasing. Virtual time effectively ''flows'' along the links of a neural graph courtesy of event timestamps, and is used to serialize and synchronize events only when needed. NeurOS assures that events arrive at modules in virtual time order. Virtual time can be slowed down or stopped/stepped so developers can monitor complex dynamics or to cope with system resource limitations, and can be sped up as resources (e.g., processing power) are available. Virtual time can be synchronized with real time at external input and output interfaces, especially useful in real-time processing and robotic control. Incorporating varying speed components and algorithms has no functional effect on overall graph operation. Processing modules usually add a

[7] This number is loosely based on the minimum typical cortical neuron post-spike recovery period and is not fundamental, just convenient.

Table 1 NeurOS elements and biological analogs.

NeurOS Element	Rough biological analog
Event (timestamp, ID, value)	New spiking rate of a specific neuron at a particular time
Module	Group/cluster/layer of neurons performing a similar function
Link (multiplexed time-ordered stream of events)	Synaptic connections from one group/cluster/layer of neurons to another
Neural graph (directed graph of modules interconnected by links)	Collection of neuron groups/clusters/layers and connections among them
Parameter	Neuron function/characteristic/type
Sharable parameter	Neuro-chemical concentrations (see State Modulation)
Memory pattern	Long-term potentiation: synapse connection geometry and strengths of one neuron or an assembly of neurons (see Memory Modules)
Memory space (collection of related memory patterns)	Region of highly interconnected neurons (see Memory Modules)

small ''processing delay'' time to output events generated from input events, simulating the biological reality that neurons are not infinitely fast. This also enables graph loops without infinite recursion.

A module implements three core life-cycle functions: **`start()`** and **`finish()`** (both optional) are called to initialize and reset module-specific resources, such as files/channels and internal state. The module **`run()`** function is called to stimulate processing as of a specific virtual time, only when new events arrive on the module's input port(s), or based on a timer for polling-based modules. As in digital signal processing, the module's **`run()`** function computes new outputs and new state from previous inputs, state and parameters. Internally, a module may use most programming language and function library features with local scope as well as multi-threading or multi-processing. To preserve dataflow parallelism and multi-processing/distributed operation, modules may share only NeurOS-managed resources with other modules: long-term memory spaces and shared volatile parameters.

Many modules include parameters to fine-tune their function. As in recent views of brain architecture (Marcus, 2014), parameters, together with neural graph design, enable building cognitive functions as diverse configurations of common building blocks. The NeurOS Designer provides pop-up property sheets to adjust parameters, at design time as well as during execution. NeurOS also manages sharable volatile parameters, useful for emulating the broad effects of ''mental states'' (see State Modulation). Parameters can be set to constants or expressions involving sharable parameters. Most parameter values can be changed dynamically during application execution.

Neural graphs can and often do have both feed-forward links and feed-back links forming loops. Feed-back links form a vital part of various ''thinking'' and ''imagination'' processes. NeurOS graphs can also be changed dynamically while they are running, especially useful for development and debugging.

NeurOS applications can be open-loop (with perhaps external loop paths from actions back to sensors), or closed-loop (with internal feedback). In some cases, the effects of likely local biological feedback looping are emulated inside a module implementation, such as in the Working Memory module.

Table 1 summarizes the core NeurOS elements and rough biological analogs.

Module types

NeurOS includes a library of built-in reusable module types (NeuroBlocks). Many intelligent applications can be built directly by composing just these module types without any new components. Table 2 briefly describes many modules used in examples in this paper. Additional modules are expected as NeurOS is further developed.

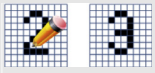
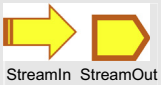
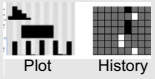
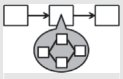
Input modules are driven by either external events or poll timers. They convert external events into the NeurOS event stream representation, with scalar, vector or matrix valued events. The Keyboard Input and Pattern Input modules are especially useful to inject specific spatial/temporal patterns during development. The Data Generator and Random Pattern modules emulate spontaneous generation of events. Future interfaces to sensor systems like ROS (Robotic Operating System) and Microsoft® Kinect®, as well as other resources (e.g., network sockets, web clients, search results, speech recognition, database access, web crawling) are expected to provide rich sources of inputs.

Output modules serve two roles: application outputs like actions (e.g., robotic control, audio output), analysis results, pattern recognition and processing results, network actions, etc.; and monitoring/visualization/instrumentation. Developers are free to litter their neural graphs with such modules (even dynamically while an application is running) to watch and capture what is happening, similar to using fancy oscilloscopes in electronic design. The Tag Cloud and Plot modules have proven very useful to visualize ''what is going on'', and the Print module provides event details during development and debugging.

The Transformer is a work-horse general-purpose processing module. It includes a lightweight expression language and function library, including scalar, vector, matrix and structure operations, for computing output events, internal module state and volatile parameter changes from input events. The Transformer also allows inclusion of externally-defined functions (see the Motion Tracking example application).

With the Transformer, one can quickly build a wide range of processing functions from simple expressions without

Table 2 NeurOS built-in module types (NeuroBlocks).

Module type	Description	Module type	Description
 Keyboard Input	Read and parse events from a keyboard in several formats	 Transformer	Transform input events to output events, internal state and volatile shared parameter changes. Includes a simple expression language and access to external functions
 Pattern Input	Graphically input spatial-sequential-temporal (SST) pattern events	 Filter Group	Pass/exclude events based on matching/range criteria; operate on groups of events
 File Input Log	Read and parse events from a file; log events to a file	 Working Memory	Repeat and decay input events
 MIDI In MIDI Out	Receive MIDI events from a source; send MIDI events to a target	 Set Memory	Learn and recognize patterns of roughly concurrent events
 GridDraw GridOut	Draw 2D input; display 2D output	 Sequence Memory	Learn and recognize patterns of sequential events
 StreamIn StreamOut	Read and parse/send events on sockets or other streams	 Temporal Sequence Memory	Learn and recognize patterns of temporally related events
 Data Generator	Create input events according to a probability/time/value distribution	 Reify	Expand Set, Sequence, Temporal Sequence patterns into their component features
 Print	Print events to a window	 Generate Patterns	Generate random pattern-recognition events from a pattern space
 Plot History	Plot events over time in several formats	 Subgraph	Reuse sub-graph as a ''macro'' module
 Tag Cloud	Show recent events with font size proportional to value/confidence	 Wrap	Wrap external program, command or service as a module

being forced into a full programming language. Following the ''be practical'' dictum, the Transformer is often more efficient for fixed (non-learning) functions than equivalent functionality built with layers of more general patterns and learning.

The Filter module is particularly useful for thresholding, to de-multiplex event streams including use of regular expressions, and to capture fleeting events for monitoring. Other modules for standard signal processing functions and for domain-specific functions like phoneme extraction, etc. are anticipated over time. The Group module performs operations on groups of events like soft-max and averaging.

The Subgraph module encapsulates a NeurOS sub-graph as a reusable module, as a convenient way to structure complex systems and hide levels of detail, and to promote reuse of especially useful cognitive sub-assemblies. The Wrapper module integrates external programs as callable library functions or via pipes/sockets/web service interfaces, with a variety of data formats.

Memory modules

Built-in memory modules address both short-term and long-term memory. Custom/wrapper modules can be used to embed arbitrary other pattern/memory technologies like neural networks, classifiers, and HMMs, as well as external database and search access. Batch and on-line learning are both enabled by NeurOS module assemblies.

Sensory memory (e.g., retinal after-image persistence, sensor fatigue) is handled by respective input modules (and of course, by any physical systems they encapsulate) emitting appropriate event streams.

The internal state of Transformer modules can be used as a form of very short-term memory, for example, computing cumulative functions over multiple input events.

The Working Memory module provides classical ''blackboard'' style short-term memory. It maintains an internal state of its recent input events, updated as new events arrive, and generates periodic repeated output events according to a parameterized decay profile, reflecting items ''currently in mind''.

A working memory module is often used to persist momentary or short-lived events so they can participate in pattern recognitions over a longer time aperture. A typical assembly might start with a high-volatility momentary input source (e.g., a retina or cochlea analog), some filtering and pre-processing, feature recognition, and then a Working Memory module to allow detected features to accumulate (e.g., from eye saccades or image scanning) and persist long enough to be matched to long-term memory patterns. The Working Memory module is also useful when multiple separate signal paths (e.g., different sensory domains) need to rendezvous with some temporal uncertainty, and in support of creative ''thinking'' processes that commingle new combinations of events.

Long-term memory modules and pattern memory spaces

Long-term memory modules and patterns model abstraction and reification processes that are key to learning and cognition.

NeurOS long-term memory modules are the closest analogs to biology, performing operations similar to neurons, dendrites and small neuron assemblies. In theory, almost all the functionality available in NeurOS could be built with compositions of these modules. Practically, fixed-function performance-sensitive operations are often better performed by other processing modules, while functions involving learning over time are best handled with long-term memory modules.

NeurOS built-in long-term memory modules follow a local rather than distributed memory representation. New pattern instances are created as needed for unique combinations of local inputs (event IDs), and adjust to repetitions of similar input combinations.[8] Indexing is used heavily to limit matching computation to just patterns that include specific input event IDs that have arrived. Non-repeated (accidental) patterns are eventually discarded.

Set, sequence and temporal patterns are core primitive abstractions that are both biologically plausible and universally powerful. These abstractions are building blocks of a great many semantics. NeurOS built-in Spatial Sequential Temporal (SST) memory modules learn and auto-associatively recognize set, sequence and temporal patterns of inputs.

- Set patterns respond to co-occurrences of input event values. Set patterns are widely reused for feature recognition, synonyms/naming, many-to-one relationships and abstraction. A set pattern is, effectively, a weight vector with weights corresponding to the ''importance'' of input features. Pattern match scoring multiplies concurrent input feature values and corresponding weights, scaled by a normalizing function. A response curve parameter enables a wide range of pattern matching semantics so that fewer or more input values are needed for significant matching confidence, spanning [any/OR/synonym, a few, some, many, most, all/AND]. Typical usage first feeds momentary events (e.g., sensory inputs) through a working memory module to persist (repeat) them, effectively providing a ''concurrency time aperture'' to a Set module. The Set module manages a collection of Set patterns, evaluating a matching score on each relevant pattern as new events arrive. A Set pattern is somewhat analogous to a single biological neuron, which, depending input geometry and cell type, can serve a similar range of semantics.
- Sequence patterns learn and match event sequences independent of time. Although biological constructs for recording and recognizing such sequences are not yet well understood, it seems clear that brains widely employ such a core capability. NeurOS includes several alternative built-in sequence pattern representation and matching styles. The primary one uses a 2D weight matrix of event ID rows and sequential step columns. The weight of an ID element (row) is highest at the sequential step(s) (columns) of an original pattern instance. A tolerance parameter spreads weights to neighboring sequence columns to allow for missing/extra/misordered sequence events, such as needed for

[8] This is one notable departure from biology: in NeurOS unfamiliar patterns lead to creating the logical equivalent of new neurons or neuron assemblies, rather than enhancing input connections to existing neurons.

recognizing misspelled words or similar melodies. Non-zero weights for multiple ID rows at the same step column represent multiple concurrent/alternative feature IDs at a sequence step. This is useful to accommodate, for example, multiple similar letter shapes like ''a'' and ''o'' arising from poor lighting or ambiguous penmanship, or multiple notes in a musical chord. Fixed-length diagonalizable sequence patterns with 0 positional tolerance function as classical N-gram recognizers. Other built-in sequence representation and matching styles are based on regular expressions, string edit distances (Wikipedia, 2014), Dehaene's positional open bigrams (Dehaene, 2009) and other forms of cross-correlation. Biologically, sequence patterns plausibly relate to neuron chains where one neuron firing for one stimulus enables one or more subsequent neurons to fire in the presence of subsequent sequential stimuli (Seung, 2012).

- Temporal patterns further impose relative timing constraints and tolerances on sequences, and allow for matching at a range of speeds/tempos. A temporal pattern keeps a sequence of proportional time-relative peak weights for each input ID. A temporal tolerance parameter governs the spread and fall-off of weights around each such peak, producing an interpolated weight curve over proportional relative time. Matching aggregates cross-correlation computations on multiple component signals (input IDs). As new events arrive they are matched to the corresponding interpolated curves to allow for matching event occurrences that are near-enough in (virtual) time. Biological brains seem to have some mechanisms for recording, recognizing and replaying time-based and time-relative sequences like music and muscle coordination, although specific biological structures for this capability are not yet clear.

These patterns can be composed via NeurOS sub-graphs as needed to represent nearly arbitrary abstractions and semantics. Feedback connections among SST modules can be used to iteratively ''chain'' through meshes of patterns (see Thinking). Multiple input features match a pattern (an ''abstraction'' of those features). This, together with other features or matched patterns, contributes to matching additional patterns, etc. (See Words, Phrases, Concepts) Often the reification of a pattern into its constituent features (see Reify module) is included in the feedback loop.

Feature weights in patterns are generally normalized to a range $(-1, 1)$ with 0 representing ''irrelevant'', positive (excitatory) weights representing feature importance to the pattern, and negative (inhibitory) weights representing the importance that the particular feature NOT be present. An alternate form of patterns using input differences from optimal feature values instead of or in addition to weights is under development.

Patterns are matched incrementally as input events arrive, and events for matched pattern IDs are emitted with values proportional to matching scores and the strengths (values) of input features. It is typical to see matching confidence scores (SST module output event values) for multiple pattern candidates grow and shrink as new input events arrive that confirm or disconfirm each pattern.

Patterns exist in memory spaces, which can be shared (usually locally) among multiple SST and related modules. Pattern modules map collections of features into collections of patterns. Matching an input event set/sequence with an existing pattern adapts (adjusts feature weights of) the highest matching pattern(s) within a memory space to the new events, controlled by a learning rate parameter and the history of previous matches.

Learning new patterns starts out as ''one-shot'' learning. A new-pattern threshold determines how strong a match is required within a memory space before a new input set/sequence spawns a new pattern. This threshold effectively controls how specific a pattern is to a particular feature set: higher thresholds create multiple distinct exemplars, while lower thresholds continually adjust stereotypical ''average'' patterns. Non-repeated patterns are typically garbage-collected. Learning an existing pattern merges current input with the pattern based on learning rate and pattern matching history. Learning in patterns can add or remove elements based on experience. Learning in sequential and temporal patterns can shrink, lengthen, insert or remove sequential/temporal elements based on experience, and adjust relative timings and time-aperture tolerances. SST modules additionally offer a ''learning profile'' to favor learning with repetition at different time intervals, and a ''forgetting profile'', to emulate varying degrees of medium-term (hours-to-days) memory.

NeurOS long-term memory modules implement flexible classification/clustering. (See Classification and Clustering, Stereotypes and Exemplars) Unlike many classification schemes, there is no built-in splitting or combining of SST patterns. Rather, patterns learn progressively and in parallel (several patterns may match concurrently or at different times). The most frequent strongest matches tend to dominate future matching to continuing experience.

Memory modules can be pre-loaded (via their `start()` method) with patterns, as well as learn incrementally. Batch learning can be accomplished with a FileInput module feeding the SST module's input (along with other more dynamic input links). Initial running of the graph then ''plays'' the file's contents through the memory module as events. Neural graphs that have run for a while can be saved together with their learned patterns, and restored and restarted at a later time with their prior learning preserved. Populated pattern spaces can be reused and even shared live with other applications.

Of course set, sequence and temporal sequence patterns are not the only long-term memory patterns possible; these just seem to have good general-purpose utility for human-scale usages. NeurOS facilities for external interfacing and custom module development can be used to incorporate other classification and pattern recognition technologies, including those with batch-oriented learning and on-line continuous learning.

Future pattern/matching possibilities include formulations like Kurzweil's value-distribution and time-distribution concepts (Kurzweil, 2012) and Hawkins/George Hierarchical Temporal Memories (Hawkins & George, 2006).

Reify module

Reify modules reverse-transform SST patterns back to their component features, emitting sets/sequences of events for the constituent elements of a pattern. Reifying a set pattern emits all the set's components concurrently, with

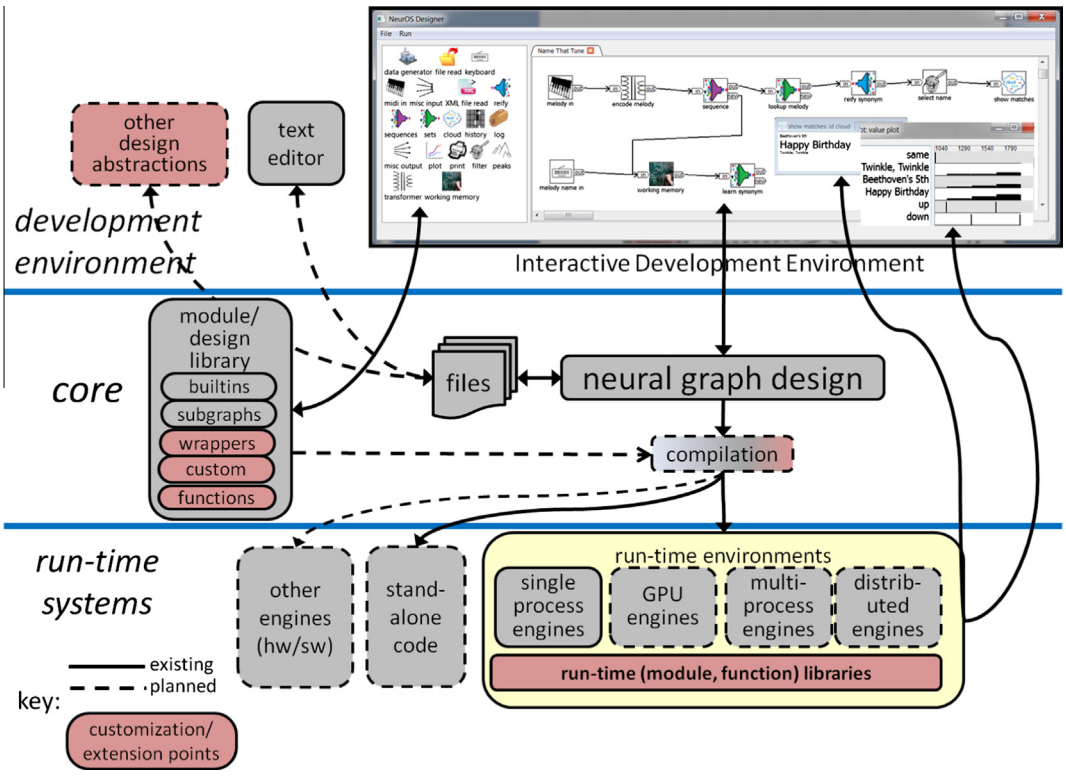


Fig. 6 NeurOS system architecture.

strength values proportional to pattern weights. Reifying a sequence or temporal pattern emits events for its elements with a parameterized tempo. Feature strengths emitted for any specific pattern follow the distribution of feature weights learned over experience matching that pattern.

The Reify module possibly models some of the extensive ''downward'' feedback connections found in brains. Layered or recursive reification expands highly abstract patterns down to successively lower levels of detail. This cascading reification is key to behavior, particularly actions affecting the external environment through layers and sub-layers of what we might think of as ''learned action macros''. Reify modules effectively implement ''generative'' processes over learned patterns.

SST and Reify modules usually share a functionally-related long-term memory space. Such a memory space is explicitly shared locally and not usually shared globally. Despite the computer science temptation to synchronize access to shared resources like this, the only synchronization is to ensure structural integrity, not synchrony. This echoes our own experience: we can easily ''miss something'' and then ''see it'' a moment later.

A frequent sub-assembly is an SST-Reify pair that implements a form of imagination or pattern completion or prediction. An SST module watches an input feature stream and generates events for likely candidate patterns in an auto-associative way. These feed a Reify module which generates output event sets/sequences including all the previously learned pattern component features, not just those that have actually been seen or heard. See Prediction.

Looping back a Reify module's output to the corresponding SST module input port commingles perceived and imagined features to ''firm up'' recognition of some object, for better or worse: ''we see what we expect to see''. See Imagination.

NeurOS system architecture

The NeurOS architecture enables rapid iterative and collaborative design, functional and performance scalability, modularity of tools and run-time systems, portability, embedding and extension.

Fig. 6 shows the NeurOS system architecture. At the core is the neural graph design, independent of both development tools and run-time environments. NeurOS run-times execute neural graphs either hosted or as standalone applications on a variety of software and hardware platforms, including uni-processors, multi-processors, distributed systems and specialized hardware like GPUs and perhaps emerging neural integrated circuits (Abate, 2014). Some run-times require a compilation step (e.g., into native and/or specialized processor code) while others can directly execute a neural graph design. NeurOS tools provide neural graph editing, manage module libraries, support custom module creation and sub-graph reuse, and provide run-time monitoring and visualization. Module libraries include platform-independent module definitions, along with corresponding tool user interfaces and run-time-environment-specific implementations.

NeurOS run-times follow typical industry models for dataflow software systems (Sousa, 2012). Thanks to dataflow rules, NeurOS application graphs can run without explicit design for parallelism and without design change using both pipelined and partitioned parallelism. Single-threaded, multi-threaded and multi-processor run-times schedule module run function executions on available threads/

processes as input events become available, with minimal mutual exclusion only for data structure integrity on link data structures and shared pattern memories. Individual module implementations can take advantage of multi-threaded parallelism internally. NeurOS run-times for non-shared-memory clusters and distributed systems will allocate module instances and related structures (e.g., modules sharing pattern memories and external resources) to nodes as appropriate, partition and pipeline event flows, pattern memories and processing as needed, and multiplex link event and parameter updates onto sockets managed by NeurOS. Activities on different nodes are only synchronized when neural graph links converge, to ensure that events are delivered in virtual time order, supporting near-linear scaling with parallelism.

NeurOS applications can be run, halted, and saved, including internal state. Thus, any amount of previous learning from experience can be captured, restarted, and copied/reused. Most NeurOS application parameters and graph structure can be modified dynamically.

The NeurOS system architecture is intended to support a broad multi-dimensional open ecosystem: integration of existing cognitive technologies and external systems, new module types, custom functions, run-times for multiple platforms including operating systems, general-purpose and specialized hardware, multi-processing, distributed and mobile environments; better and alternate tooling; open exchange of sub-assemblies; and of course, intelligent applications.

NeurOS usage: common constructs and design patterns

We have barely scratched the surface of the potential of intelligent functions and applications that can be built with NeurOS modules and neural graph composition. Nevertheless, even with limited exploration, several frequently used sub-assembly constructs, techniques and design patterns have emerged. As in biological brains, cognitive functions emerge from how neural sub-assemblies are interconnected. These constructs perhaps illuminate some plausible biological mechanisms for aspects of cognitive function. Think of the following sub-sections as analogous to electronic component ''application notes'' — typical usage patterns combining available components.

The purpose here is not to assert any definitive or best solutions to any cognitive challenge area, merely to show the breadth of application of NeurOS constructs and the possibilities of reusable sub-assemblies.

A couple of points about the presentation of these usages and the following Example NeurOS Applications:

- The scenarios are purposely simplified down to only a few patterns for explanatory ease. More realistic applications will have thousands of patterns and events.
- Many normally internal entities like features and recognized patterns are shown with meaningful symbolic IDs for explanatory purposes only: ''vanilla ice cream'' or ''Fido'' rather than ''set_243546''. In any substantial application, almost no internal events will have meaningful symbolic labels. As in biological brains, a neuron recognizing an ''ice cream'' concept has no symbolic label and indeed is ''identified'' only by its connections.
- Much activity in NeurOS applications (and presumably brains) happens very dynamically. In a running NeurOS application, dynamic graphical activity displays like active-link-coloring and the Tag Cloud module show these rapid changes visually. This is difficult to capture in static figures. Segments of Plot module displays are used frequently to show how activity develops over time.

Application structure

NeurOS is a decidedly non-von Neuman non-procedural computing architecture. Designing an application in NeurOS is somewhat different from writing a conventional program. Here are some design profiles that have emerged:

- **Multiple concurrent ''solutions'':** Like biological brains, NeurOS tends to continually compute multiple results in parallel and over time, with many concurrent activities in unplanned and possibly non-deterministic intertwining flows. Multiple patterns may match stimuli to different extents, and these recognition strengths change over time as more stimuli arrive. There is rarely any ''single answer'' anywhere, but rather an ebb and flow and swirl of recombinant collections of possibilities with different degrees of confidence over time, feeding additional processing, fading out (e.g., failing to sufficiently stimulate other pattern recognitions), or being overshadowed by other activities.
- **Multiple paths/multi-channel processing/domain fusion:**Unlike procedural programming, NeurOS applications tend to do lots of work in parallel, much of it speculative. This is not for performance reasons (NeurOS handles parallel performance scalability transparently), but to perform different kinds of processing. For example, reading might involve multiple feature dimensions of letter shape, letter and N-gram sequences, familiar phrases, phonemes, etc. (Dehaene, 2009). Depending on ambient lighting, noise, speaker accents, handwriting/fonts, context, vocabulary, familiarity, etc., feature recognitions and strengths may flow in flexible feed-forward and feed-back paths. Two (or more) concurrent paths may address different working regions of pattern matching, for example different sequence matching algorithms.
- **Fixed functions at the edges, memory/learning in the middle:**Perhaps reflecting biological brains, NeurOS application graphs tend to have more specialized and fixed (non-learning) modules at the sensory and output edges of graphs, and SST pattern memory modules and processing toward the middle. This implicitly also serves a performance function: sensory preprocessing tends to be dense and high-dimensional and highly compute-intensive and so is better implemented in fixed functions, while much compute-intensive pattern matching is done on dimensionally-reduced abstracted features.
- **Wide input fields:** Discrimination among patterns is often enhanced by including more features computed

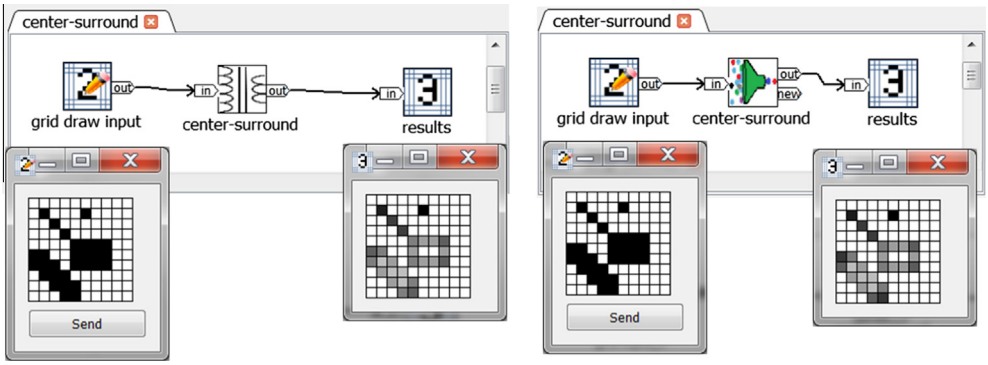


Fig. 7 Center-surround processing using a Transformer or Set patterns.

along multiple parallel paths, perhaps from multiple domains. Pattern learning mechanisms winnow down the important features with experience.

- **Layers of recombination:** The nature of many cognitive problems is recombination of recurring patterns at multiple levels: shapes/letters/N-grams/words/morphemes/phrases, edges/corners/objects, frequencies/phonemes/words, etc. Each layer learns the frequently recurring combinations (sets, sequences) of elements from a lower layer, a much smaller subset than the full combinatoric space.
- **Variables and references:** IDs (analogs to individual neurons/axons) are variables whose values change with activity/time, similar to spreadsheet cells; an event reports a value change to an ID. A link serves as a re-bindable reference or pointer mechanism, effectively a list of non-0-valued IDs as of a particular time. In this way, a link effectively represents the most recent activities/results from its source scope.
- **ID filtering:** NeurOS graph links are multiplexed over a possibly large space of neural signals, often from multiple domains. An especially good example is the output of a Reify module, which may produce component features of a pattern from multiple domains, not all of which are interesting to all targets. The primary de-multiplexing mechanism in NeurOS is to filter event streams by ID. It is often useful to add a prefix to event IDs signifying a domain or other grouping, to allow downstream modules to select relevant events (e.g., with a regular expression on event IDs). This is used for example in the What's-That-Tune? application where the synonym relationship between an abstract invariant melody code and its name is reified, and then the name selected for display.

Perception

Much low-level biological perceptual processing is fixed and uniform across a broad input field. Such pre-processing can be addressed with NeurOS sub-graphs using general pattern mechanisms, but is often far more efficient with specialized processing. Fig. 7 shows two equivalent examples of well-known ''center-surround'' lateral inhibition visual processing (Marr, 1982), used by biological visual cortexes to highlight edges.

The Grid Draw input module emits an event with a matrix of pixel intensity values of 0 or 1. In the version on the left, the Transformer module uses a center-surround computation function to compute the output matrix, using a center pixel weight of 1 and −0.125 weights for the 8 surrounding pixels, and limiting the results to the 0–1 range. The Grid Output ''results'' module image shows the expected contrast-enhanced results in shades of gray.

This same processing is done with a NeurOS Set module as shown on the right using a separate Set pattern for each output pixel. Slower and uses more memory, but yields the same results.

This kind of processing structure can be cascaded to model various sensory pipelines/meshes.

Working memory

NeurOS includes a dedicated Working Memory module. However, a simple form of working memory uses a feedback loop, as in Fig. 8:

The ''inputs'' (Keyboard In) module emits four event IDs: ''big'' and ''black'', followed by ''barks'' at 500 ms and ''fur'' at 1000 ms, simulating staggered arrival of sensory inputs. The Filter module is used only to apply a gain factor (0.9 here) and a delay (100 ms here) to its input events. Input events then cycle around with exponentially diminishing strength, with parameters for gain and delay. The Plot modules show the original and resulting exponentially declining values.

This sub-assembly is perhaps analogous to biological ''autapses'', where an axon branch connects back to an input dendrite of the same neuron (Seung, Lee, Reis, & Tank, 2000).

Words and other sequences

One possible representation of words in NeurOS is as simple sequences of letters. The matching tolerance parameter of **Sequence** patterns allows such sequences to be recognized even when misspelled in a variety of ways. For example, the word ''badge'' (Dehaene, 2009) is represented by the following Sequence pattern[9] with a tolerance of ±1 positions.

```
{'b': [1,   0.5, 0,   0,   0],
 'a': [0.5, 1,   0.5, 0,   0],
 'd': [0,   0.5, 1,   0.5, 0],
 'g': [0,   0,   0.5, 1, 0.5],
 'e': [0,   0,   0,   0.5, 1]}
```

Fig. 9 shows a simple NeurOS sub-graph.

[9] This is shown in the format of a Python dictionary.

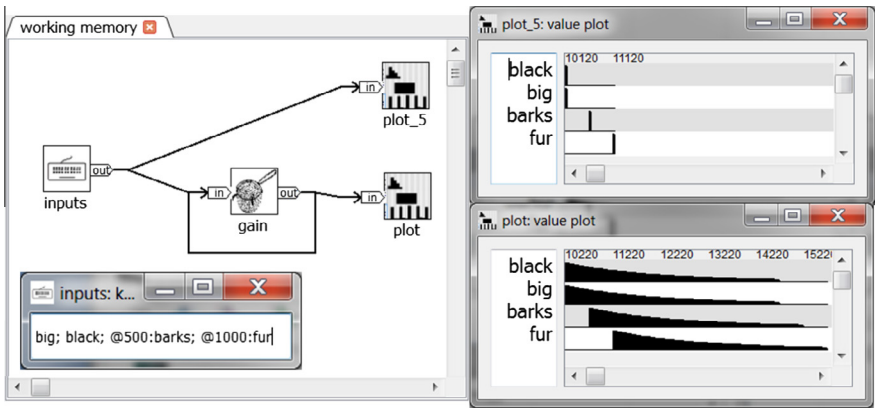


Fig. 8 Working memory using a Filter module feedback loop.

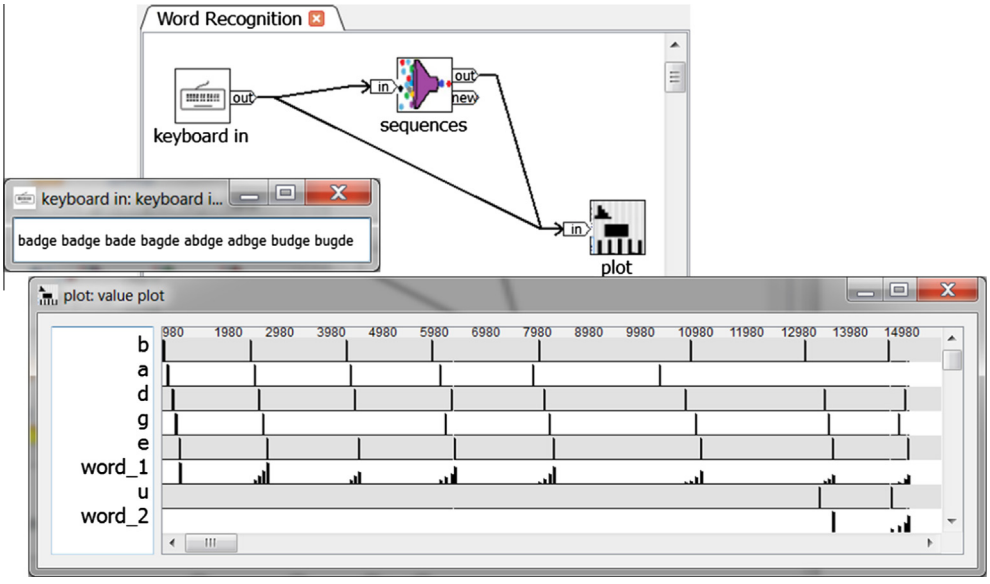


Fig. 9 Word recognition with positional tolerance.

Letters are entered individually via the ''keyboard in'' module at the left and propagated as individual events. Initially the ''sequences'' module learns the word ''badge'' (assigned the internal ID ''word_1'' at pattern creation time). Subsequently the ''word_1'' sequence pattern incrementally matches the correct spelling most strongly, and several different similar misspellings less strongly. The word ''budge'' is sufficiently dissimilar to ''badge'' that the Sequence module creates a new distinct pattern (''word_2''). Subsequent entry of the misspelling ''bugde'' matches word_1 (''badge'') weakly, but word_2 (''budge'') strongly.

This design pattern applies not just to written words, but to time-independent sequences in many domains, such as phonemes, simple music encodings (as in the What's-That-Tune? example), behavior and more.

Words, phrases, concepts

''Word'', ''phrase'' and ''concept'' apply not just to language, but also to many other domains: music, shapes, image segmentation, and across domains. The natural representation of words and phrases in NeurOS is the Sequence pattern, an ordered series of elements (whatever they represent) together with auto-associative pattern matching.

For example, the familiar phrase ''four score and seven years ago'' is a sequence of words, each word a sequence of letters. In the simple NeurOS sub-graph in Fig. 10, as letters are entered individually via keyboard, the first Sequence module ''letter seqs'', holding (pre-loaded) word patterns, progressively emits matching word candidates. The ''best matches'' Filter module passes only high-confidence word matches. These in turn feed the second ''word seqs'' Sequence module which progressively recognizes phrases (also pre-loaded). The plot display shows the recognition strengths of individual words and the progressively more strongly recognized phrase.

Fig. 11 shows a subsequent trial, where the neural graph starts to recognize the familiar phrase, but as the new input departs from what is expected, the recognition strength remains low.

More generally, compositions of NeurOS Set and Sequence patterns naturally represent a huge range of concepts. Using conventional list $[a, b, \ldots]$ and set $\{a, b, \ldots\}$ notation, the English sentence ''The big brown dog chased the cat.'' might look something like [[The, big, brown,

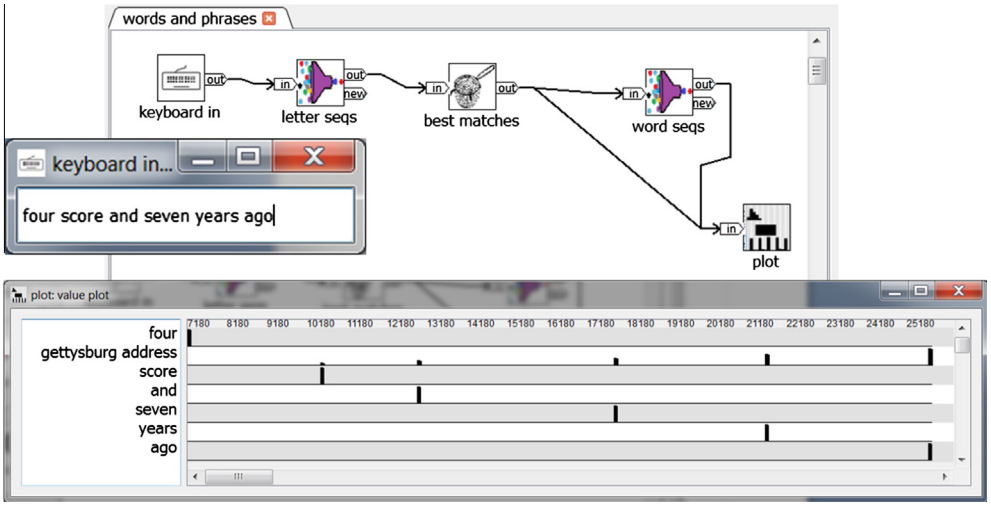


Fig. 10 Words and phrases in NeurOS.

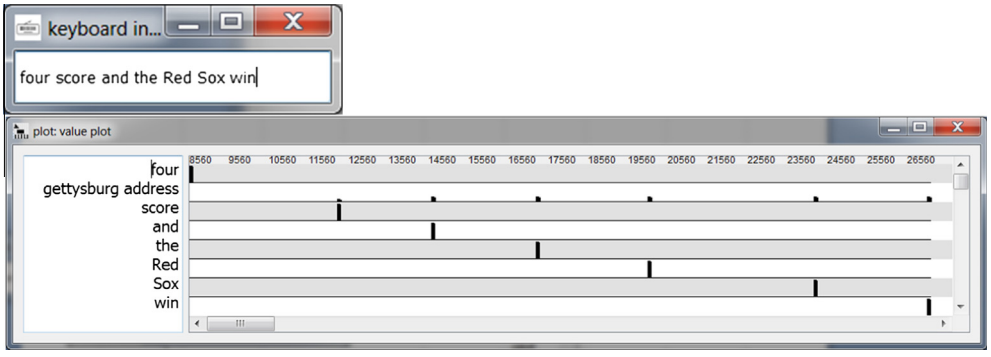


Fig. 11 Partial recognition.

{furry, barks, tail}], chased, [the, {furry, meows, tail, claws}]].[10] Indeed, each word is itself a sequence of letters, each letter a set of visual properties like edges and curves, etc. A musical phrase likewise is a sequence of sequences (e.g., intervals, arpeggios, runs) and sets (e.g., chords). Each note played by each instrument is itself an audio spectrum (set of audio frequencies with energy/sound levels) temporal sequence. A person can similarly be represented as a set of features, some of which are sets (visual attributes, voice frequency spectrum) and others sequences like gait and movement. In NeurOS these are naturally represented as chains of Set, Sequence and Temporal sequence modules representing layers of abstraction/processing.

Prediction

By adding a Reify module to the graph in Fig. 10 we can begin to see how prediction can work in NeurOS, as shown in Fig. 12. As the words are entered in the keyboard, the ''gettysburg address'' pattern is progressively more and more strongly recognized. It is immediately reified (by the ''reify seqs'' module) into its component elements. (A ''predicted_'' prefix has been added here to distinguish these imagined features from ''sensory'' inputs. In a realistic system, these predicted elements would have the same identifiers as the sensory inputs.) Downstream processing can immediately begin to operate early as if the predicted items had already been perceived.

Entering an initially familiar phrase leads to predictions which fail to grow as the similarity fades, as shown in Fig. 13.

Synonyms and naming

Names and other aspects of a concept (e.g., feature patterns, alternative shapes for letters, stereotypes, exemplars, pronunciation, synonym words, images of the same object) are akin to synonyms. In NeurOS, such synonyms are conveniently represented as disjunctive Set patterns: multiple inputs with a matching semantic parameter curve set to an ANY/OR value. Arrival of a significant signal for any member of the set stimulates the overall concept pattern. A subsequent Reify of the concept pattern activates

[10] Reading and language representation and algorithms are considerably more complex than this simple grammatical nesting, involving additional domains (e.g., phonetics) and feedback connections (Dehaene, 2009).

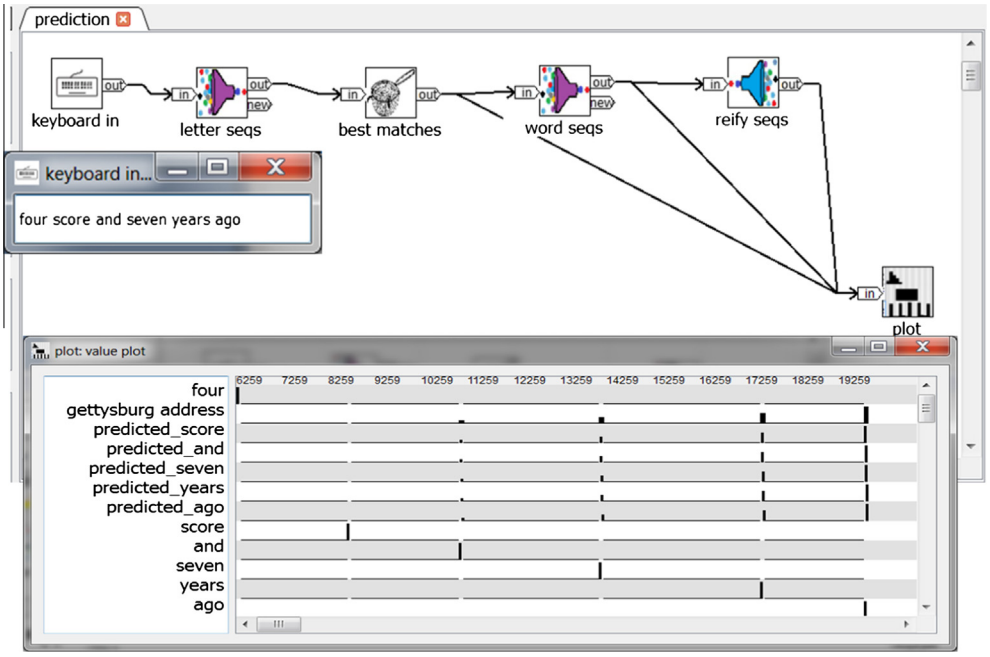


Fig. 12 Prediction.

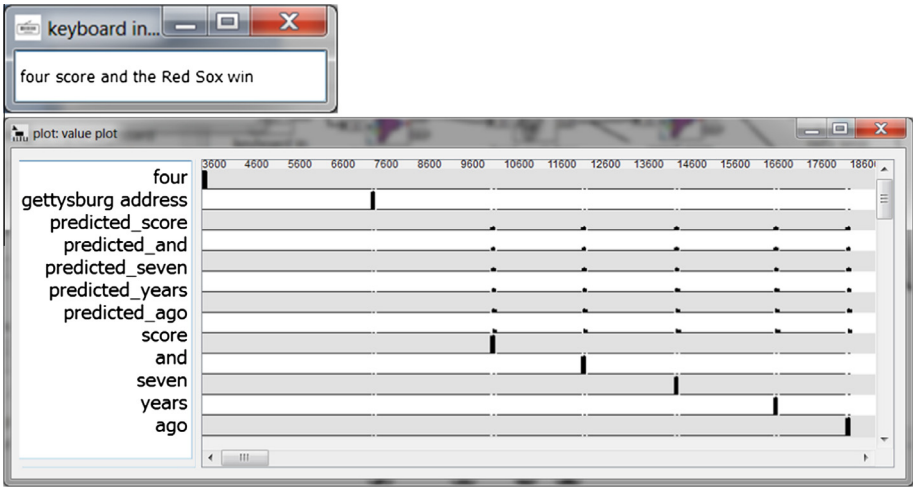


Fig. 13 Curtailed prediction.

all the component synonyms, typically propagating to further processing, imagination feedback and/or output.

In Fig. 14(a), the words ''pier'', ''dock'', ''wharf'' and ''levee'' were previously entered concurrently enough to be captured as synonyms in a new Set pattern by the ''set patterns'' module. Subsequent entry of the word ''pier'' (shown) stimulates this synonym pattern, which is then reified into its component event IDs, which are displayed in the tag cloud output.

We will see this Synonym-Reify construct repeatedly, for example in Prediction, Imagination, Thinking, and What's That Tune? — the enhanced version.

This is a good candidate for creating a reusable composite sub-graph module, as shown in Fig. 14(b). The ''in'' port to the ''set patterns'' module becomes the ''in'' port to the composite module. The composite module offers two output ports: ''abs'' emits events for matched set patterns (whole abstract synonym concepts), and ''out'' delivers reified synonyms for matched concepts. ''abs'' typically feeds further abstraction processes such as semantic composition, while the new ''out'' port typically feeds display and output processes (e.g., ''putting a concept into words'').

Conjunctive-disjunctive assemblies

A frequent cognitive construct is the combination of multiple aspects of a thing: multiple shapes for a letter; sound, shape and letter sequences for a word; multiple synonym words for a meaning; and as in the — **What's That Tune? — the enhanced version** example application, the name, coded signature sequence and original melody for a tune.

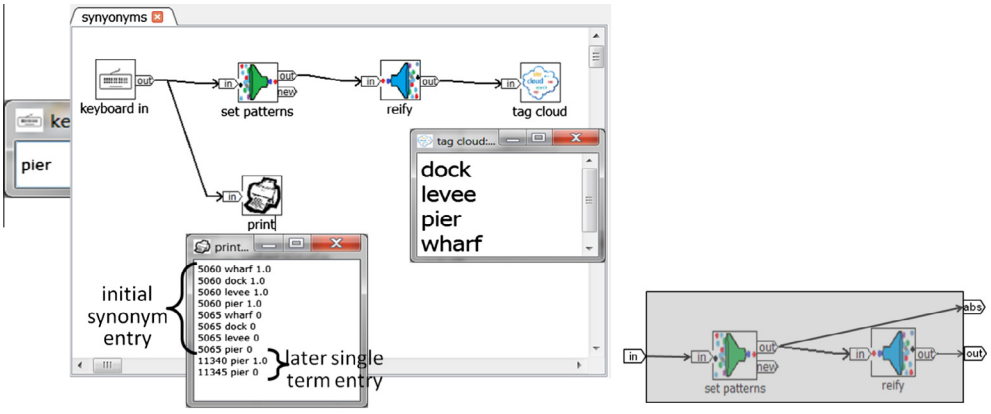


Fig. 14 Synonyms.

In NeurOS, a sub-graph like that in Fig. 15 following accomplishes this:

The ''features 1'' and ''features 2'' transformers represent sources of features, perhaps from different sensory domains (e.g., vision and hearing) or perhaps from different preprocessing channels within a domain. The ''sets *n*'' and ''sequences *n*'' modules learn and recognize distinctive set and sequence patterns from these features. These alternative aspects of what is currently being experienced are collected in the ''disjunction'' module which learns/recognizes sets with an ''any'', ''a few'' or ''some'' semantic.

Used in layers and/or loops this construct can support complexities like sound and vision feeding phoneme and letter shape recognition feeding word recognition feeding meaning and so on.

Classification and clustering, stereotypes and exemplars

NeurOS enables building a range of classification schemes. There are no inherent requirements about hierarchy or exclusivity; these emerge from how modules are connected and parameterized. SST modules offer a new-pattern threshold parameter, which specifies a minimum pattern matching confidence before a current event collection is recorded as a new pattern. Matches that exceed this threshold participate in adjusting feature weights of the strongest currently matching pattern(s). If no match exceeds this threshold, a new pattern is minted. Thus, a parallel suite of SST modules all fed from the same feature space can yield a flexible fuzzy multi-hierarchy of stereotypes and exemplars. One SST module is tuned to create new exemplar patterns for any mildly distinct new input pattern; non-repeated (accidental) exemplar patterns can be later discarded for efficiency. Other SST modules are tuned for increasingly broader clusters, adjusting the equivalent of cluster centroids and breadths with experience. Accidental feature weights eventually fade as part of continuous learning.

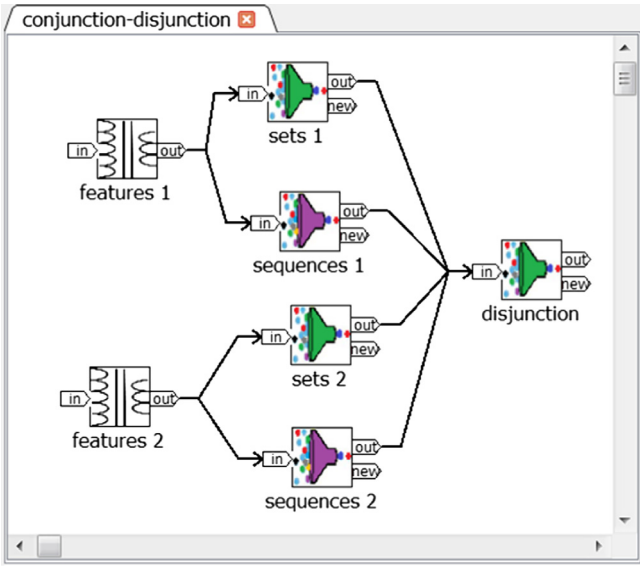


Fig. 15 Conjunction-disjunction sub-graph.

Fig. 16 shows a simple example of this usage. Concurrently observed features are entered as shown in the temporal ''pattern in'' module at the left. The ''exemplars'' Set module has a high new-pattern threshold, meaning that it will create a new exemplar pattern for any input feature set that is not a good match to an existing exemplar. The ''stereotypes'' Set module has a lower new-pattern threshold so that similar feature set patterns will progressively train a moderately-matching pattern rather than creating a new stereotype.

The ''plot_7'' window shows the sequence of input features commingled with new pattern creations at different virtual times. After the first feature set (around virtual time 10140), both a new exemplar ''exemplar_1'' and a new stereotype ''stereotype_1'' are created, highlighted with red dashes. The second input feature set (around time 10,240) is different enough to create a second exemplar ''exemplar_2'', but not different enough for a new stereotype. The same is true for the third input set. The ''recognitions'' tag cloud output shows the relative recognition strengths of the four patterns after the third input feature set. Not surprisingly, the new input matches its own new exemplar pattern the most. It strongly matches the stereotype, but only weakly matches the other somewhat different exemplars.

The resulting internal pattern representations after the inputs are shown here.

```
Set pattern ''exemplar_1'' (count / 1):{'barks':1.0, '4 feet':1.0, 'big':1.0, 'friendly':1.0, 'fur':1.0,
'brown':1.0, 'tail':1.0}
Set pattern ''exemplar_2'' (count / 1):{'yips':1.0, 'nasty':1.0, '4 feet':1.0, 'small':1.0, 'white':1.0,
'fur':1.0, 'tail':1.0}
Set pattern ''exemplar_3'' (count / 1):{'medium':1.0, 'black':1.0, '4 feet':1.0, 'friendly':1.0,
'slobbers':1.0, 'whines':1.0, 'fur':1.0}
Set pattern ''stereotype_1'' (count / 5):{'medium':0.101, 'yips':0.196, 'slobbers':0.101, 'tail':0.898,
'black':0.101, 'small':0.196, 'white':0.196, 'brown':0.702, 'whines':0.101, 'barks':0.702, '4 feet':1.0,
'big':0.702, 'friendly':0.803, 'nasty':0.196, 'fur':1.0}
```

Note that the individual exemplars only have a single experience input (count = 1) each so all their feature weights are 1.0 as in the original events that created the exemplars. Their feature weights will only train if they are among the highest matching patterns for any subsequent new feature input set (e.g., another sighting or view of the same or a nearly identical dog). The stereotype pattern, however, shows repeated training and incorporates both the common features of all the inputs (weights = 1.0) and the occasional features with lower weights. Note also that the stereotype_1 pattern's feature weights are affected by history order: the missing 'tail' in exemplar_3 reduces the 'tail' weight in the stereotype only a little. However, the fact that 'black' only shows up for the first time in the third example gives it a low weight in the stereotype. This, arguably, resembles human experience: we are most strongly affected by our earliest experience; first impressions count for a lot.

Thus, seeing your pet Fido might stimulate a specific Fido exemplar, a dog breed stereotype, a broader generic dog stereotype, a broad animal stereotype and a (non-dog-specific) pet stereotype, all in parallel. Seeing a furry animal initially might stimulate just the broad animal stereotype. Upon further observation, the dog stereotype and dog breed stereotypes might light up, and once you recognize Fido, his exemplar might light up most strongly. Note also that the patterns are not necessarily exclusive nor hierarchical. The same features might also stimulate a ''pet'' stereotype, a ''friend'' stereotype, and even a ''danger'' stereotype.

NeurOS patterns learn from experience. Just like two people can never have identical experience histories and so build up different associational and classification structures in their minds, there are no guarantees that a NeurOS network will converge to the same classification structure given similar but non-identical experiences.

Current NeurOS memory modules do neither cluster splitting nor merging. Instead, multiple patterns grow and change with experience. Those that do not match much experience are eventually garbage collected, while those that match frequently and strongly become dominant and permanent. There are no formal hierarchical relationships

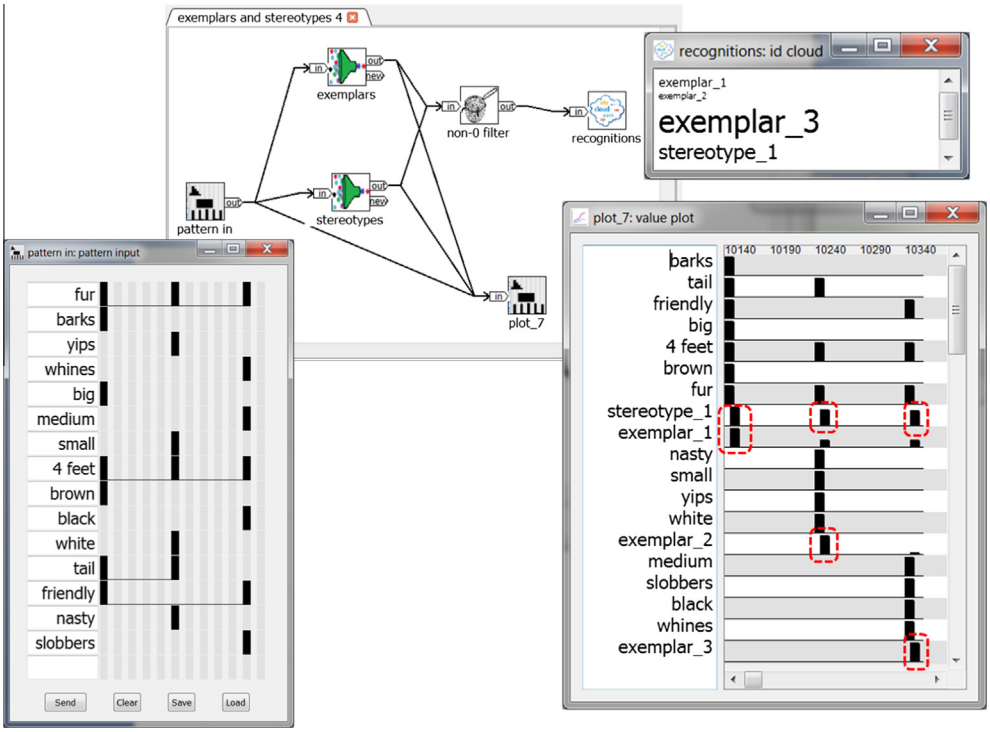


Fig. 16 Stereotypes and exemplars.

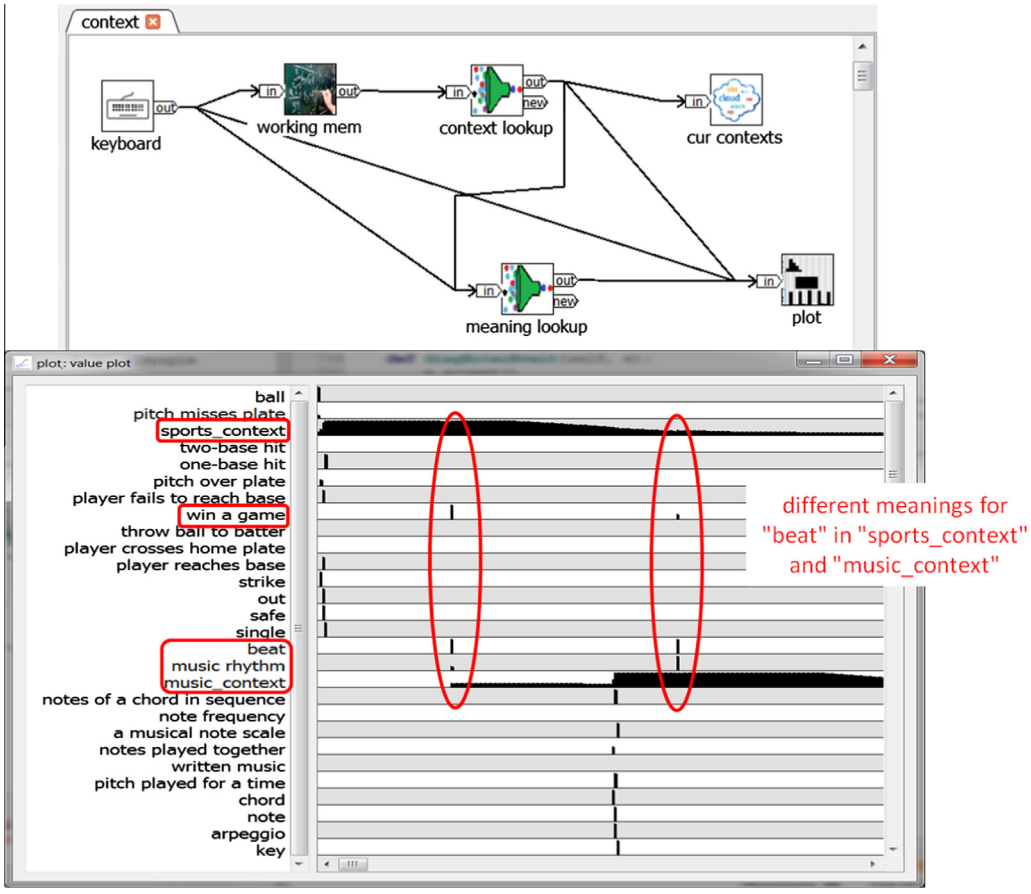


Fig. 17 Context priming.

among the different patterns, so any feature combination can stimulate multiple patterns. This also arguably mimics human experience where clear-cut disjoint and strict hierarchical clustering are the exception rather than the rule.

Context priming

Our interpretations of ambiguous words like ''beat'', ''pitch'', and ''score'' depend strongly on the context of our recent conversation, for example: music or baseball. Conversely, the context of a conversation derives from the words recently used. If we hear ''ball'', ''strike'', ''safe'', ''out'' and ''single'' we are likely in a ''sports'' or ''baseball'' context. If we recently heard ''chord'', ''arpeggio'', ''note'' and ''key'' we are more likely in a ''music'' context. Fig. 17 illustrates this dynamic.

Initially, the words ''ball'', ''strike'', ''out'', ''safe'' and ''single'' are input via the ''keyboard'' module. These features are persisted for a few seconds in the ''working mem'' module, and are combined in the ''context lookup'' Set module to establish a strong ''sports_context'' context, which in turn persists for a few seconds. Subsequently entering the ambiguous word ''beat'', together with the current context, are looked up in the ''meaning lookup'' Set module, yielding a strong interpretation as ''win a game'' and a very weak alternative of ''music rhythm'' because of the lack of a music context. Without continued refresh (as in a continued sports-related conversation), the working memory module decays the sports_context. Subsequently entering the words ''chord'', ''arpeggio'', ''note'' and ''key'' firmly establishes a ''music_context'' context. Reentering the word ''beat'' now yields a strong interpretation as a ''music rhythm'', and a very weak one for ''win a game''.

State modulation

Biologically, different mental and emotional ''states'' appear to be modulated by neurochemical ''messengers'' in certain brain regions, for example the ''fight or flight'' stress response to danger, with broad effects on perception, thinking, memory and action.

In NeurOS, these broad effects can be emulated using sharable volatile parameters. Many NeurOS modules have local parameters affecting things like learning rates, sensitivity, scaling, delays, and significance thresholds. Rather than static constants, these parameters can be set to live expressions referring to and combining the values of sharable volatile parameters. The sharable parameter values can be adjusted dynamically by a Transformer module in response to ''current events'' to affect for example, attention, concentration, focus, sensitivity, balance of external perception vs. internal imagination, openness to learning, etc.

Fig. 18 models this process for a simple case of danger response. Along the top (green highlighted) path, raw perceptual inputs directly feed a ''danger patterns'' Set mod-

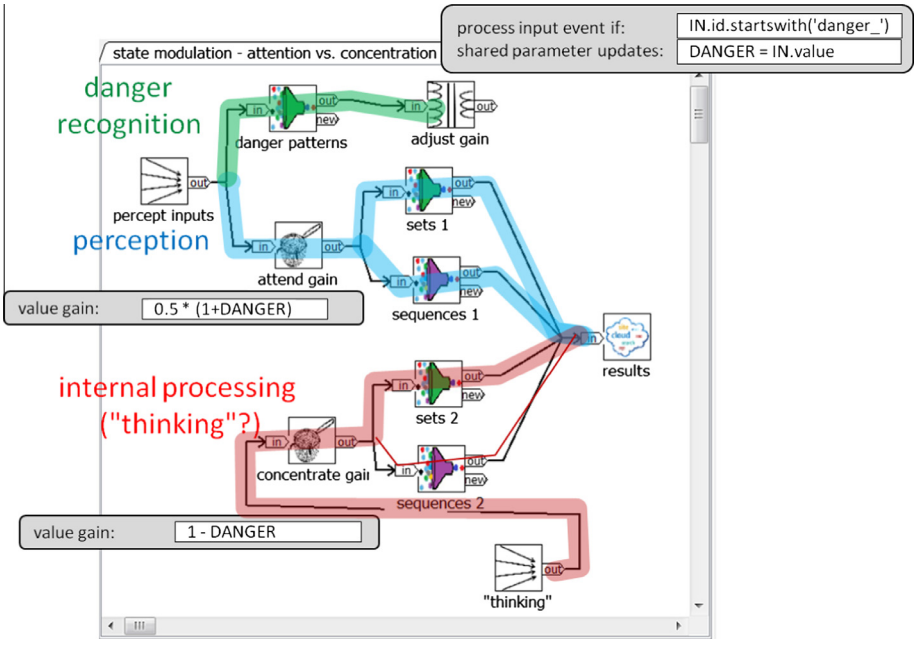


Fig. 18 State modulation — attention vs. concentration.

ule, analogous perhaps to some ''lower/primal'' brain functions. Any (pre-loaded or learned) danger patterns matched feed the ''adjust gain'' Transformer module. (Only relevant parts of property sheets for Transformer and Filter modules are shown.) If any danger event is received (the IN.id.startswith('danger_') expression is true), the DANGER shared parameter is updated with that pattern's value (DANGER = IN.value).

Two separate paths shown represent perceptual (blue) vs. internal (pink) processing. Each path includes a Filter module whose ''value gain'' parameter is set to an expression. In the absence of any danger, the normal gain factor on the perceptual path is 0.5 (e.g., ''background attention''); the normal gain is 1 on the internal processing path (e.g., ''concentrating''). The DANGER shared parameter is normally 0. If a danger pattern is detected, the ''adjust gain'' Transformer boosts DANGER to a corresponding high value. This boosts the ''attend gain'' module's gain via the 0.5 * (1 + DANGER) expression, and drops the ''concentrate gain'' module's gain to much closer to 0 via the 1-DANGER expression. As the danger passes, the 'danger_' events detected diminish in strength, and the respective gains correspondingly settle to their normal values.

The DANGER parameter here is sharable via multiple modules on multiple paths in larger neural graphs, emulating the broad effects of biological ''fight or flight'' chemical changes. This same technique can be used to implement effects of other mental and emotional states like excitement, depression, sleep, and wariness.

In a robotic context, such state modulation through shared parameters can address prioritizations such as low-power conservation modes and ''find power'' behaviors; adjusting sensor sensitivities to environmental changes; boosting ''where am I?'' behaviors when lost and re-planning behaviors when needed.

Teaching

Learning generally involves both experience (unsupervised) and teaching (supervised). In NeurOS, experience and teaching can be intermixed as shown in Fig. 19.

The ''main line'' (upper) path feeds on-going experience through an SST set/sequence/temporal sequence memory module for pattern matching and subsequent processing. Learning parameters along this path are set to minimal values, for on-going fine tuning and adjustment of already learned patterns. Teaching input feeds a separate SST module sharing the same memory pattern space, with a learning rate parameter set high. Via this path, a teacher (or other source) supplies ''especially good examples'' to add new patterns or substantially adjust existing patterns. These take effect immediately because both set modules share the same pattern space, and so apply to subsequent experience input. The additional teaching path can even be configured on the fly in a running system. A similar effect can be achieved over time with a single SST module, by dynam-

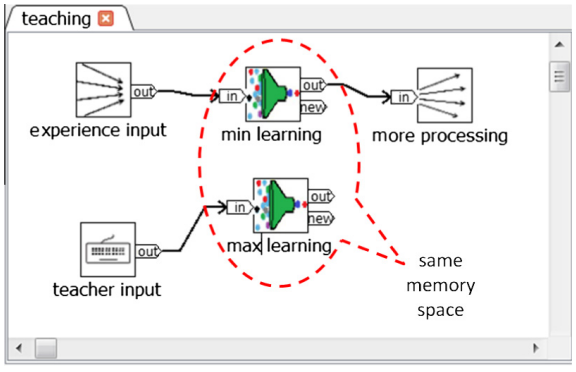


Fig. 19 Teaching in NeurOS.

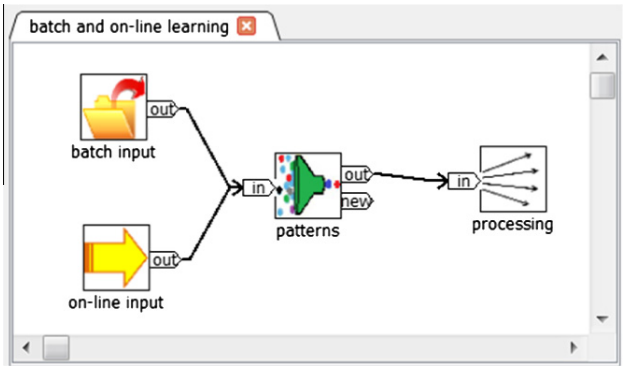


Fig. 20 Batch and on-line learning.

ically raising its learning rate parameter during training periods (see State Modulation).

Batch and on-line learning

Batch and on-line learning can be commingled, as shown in Fig. 20. At the start of running of an application, the File In module labeled ''batch input'' reads feature data samples from a file and feeds it continuously to the ''patterns'' Set module to learn and adjust patterns. Subsequently (or concurrently) on-line input feeds the same module. Other input modules (like database query) can also be used as sources of batch learning. The on-line input can also derive from external sources like sockets/streams/image/audio sources, etc.

Interruption and concentration loss

Concentrating means ''keeping several things in mind at once'', like a programming task, writing or playing music. In NeurOS, this is naturally modeled as multiple concurrent input features feeding one or more pattern-recognition modules. If source feature events occur or are recognized within a reasonable time range with high confidence, the signals overlap in time and can be recognized strongly by a known pattern. Interruptions, distractions and brain resource conflicts (i.e., multi-tasking) may delay or diminish individual feature signals, leading to weaker or no pattern recognition.

The example in Fig. 21 shows a simple neural graph. Input time distributions of ''fur'' and ''barks'' feature signal strengths are entered via the graphical ''pattern in'' module. Each signal trace shows a normalized strength in a [0, 1] range. The input feature pattern and resulting pattern matching strengths are shown in the Plot module display.

The ''sets'' Set module has a (pre-loaded) pattern for ''dog'' which includes the features ''fur'' and ''barks'' and a matching semantic parameter of most (most of the aggregate strengths of concurrent input features are needed to stimulate a significant output response). If ''fur'' and ''barks'' roughly co-occur as in the first time segment shown, they strongly stimulate the ''dog'' pattern. If ''barks'' is delayed by even as little as 30 msec, the overlap of signal strengths is diminished and ''dog'' is recognized weakly and a few time steps later, as in the second segment. Delay the recognition of ''barks'' even longer (third segment), or reduce its input confidence (e.g., due to noise) (fourth segment), and the recognition of ''dog'' diminishes further. (Speculatively, this dynamic response of a single pattern can all be done biologically by a single neuron!).

More generally, the input features come from sensory pre-processing paths, working memory, and imagination feedback, and the Set module holds thousands of previously learned patterns. Disruption of any of these many paths can delay or diminish feature signals needed to keep ''many plates spinning'' or ''many balls in the air'', that is, to sustain the continuing recognition of patterns that constitute our collective current working memory ''state''.

Imagination

Imagination involves the ability to ''see'' a collection of features without actually experiencing them. Often we visualize a familiar pattern such as our dog Fido or a sunset. The NeurOS Reify module is the primary vehicle for this: start with an event for a previously learned pattern, and activate all of its component features. Via feedback connections, these features can be ''experienced'' as if we are currently perceiving them.

If, for whatever reason, we happen to ''think of'' several familiar concept patterns, like ''flying'' and ''horse'', they may simultaneously reify into surprising new combinations of features which are not constrained to match reality. If we are in an ''open-minded'' state (i.e., learning rate parameters set high), this serendipitous concurrence may mint a new pattern. If we repeatedly ''think of'' these same abstractions, the concurrence (or sequence) of their

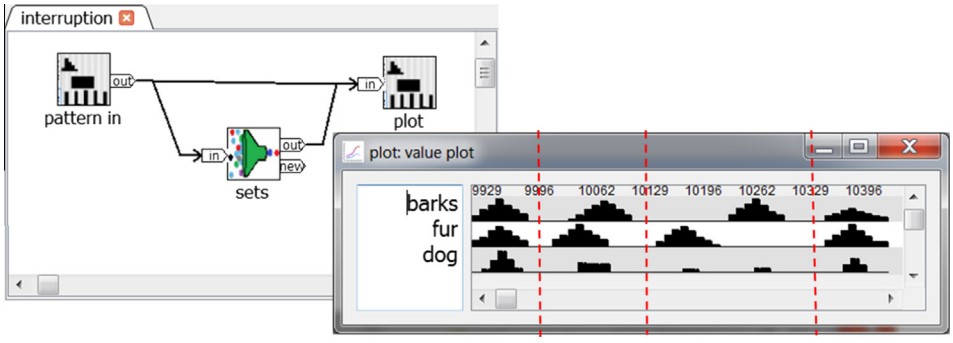


Fig. 21 NeurOS graph showing dynamics of interruption.

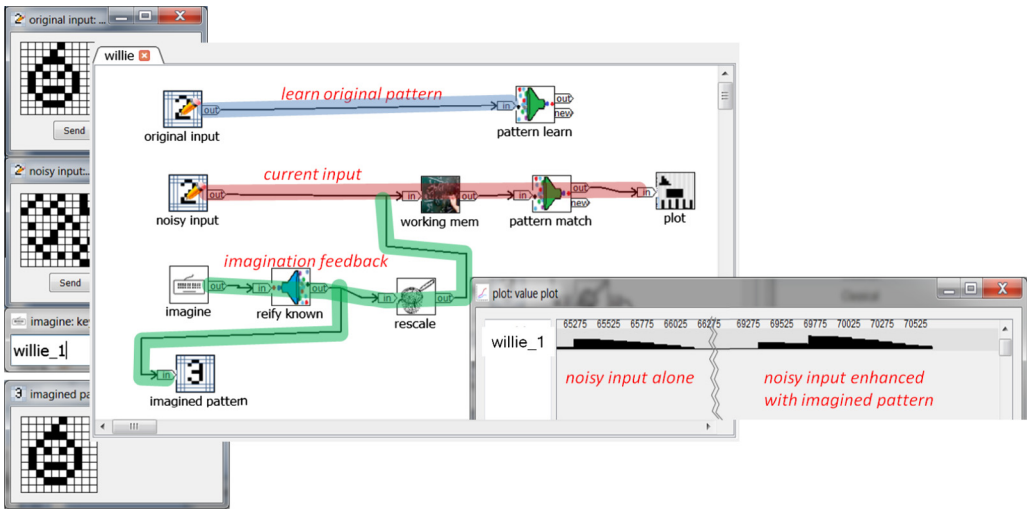


Fig. 22 Finding known patterns.

component features may become familiar enough to strengthen the new pattern and prevent it being garbage-collected. We can thus ''remember'' things that we never experienced. The Data Generator and Random Pattern modules can contribute random generation of input and pattern-recognition events to such imagination processes.

A particular role for imagination is perception enhancement for known patterns, as shown in the next section.

Where's Willie?

Finding a known pattern in a busy, noisy input field is a frequent challenge in multiple domains: perceptual domains like vision and hearing as well as recognizing abstractions. Having a clear image in mind of what we are looking for can boost recognition. The NeurOS sub-graph in Fig. 22 suggests one plausible approach.

Three distinct paths are shown. In the top (blue highlighted) path, an original ''Willie'' image is entered and learned as a new Set pattern. The middle (pink) path feeds the current (noisy) input image through a Working Memory module to feed pattern recognition. The bottom (green) path simulates imagination feedback of the pattern sought. In the first pass shown, the noisy input is entered alone and shows a modest recognition in the Plot display. In the second pass, the looked-for pattern's identity ''willie_1'' is imagined (simulated by keyboard entry into the ''imagine'' module), reified into its component features, rescaled (so as not to overshadow input evidence) and additively combined in the working memory, yielding an enhanced recognition score greater than perception or imagination alone. This additive combination of perceived and imagined features models the convergence of feed-forward and feed-back synapses on individual neurons.

This works as well with input of multiple known pattern IDs, searching for multiple known patterns or the same object in multiple poses. In a realistic application, the known pattern IDs would instead be supplied as the results of other processing, for example speculative thinking about what one might find or reified memories of images of a favorite person or pet. Also current pattern input would likely be a sequence of images resulting from a scanning process such as visual saccades or other abstract feature sets.

Thinking

Any universal code of ''thinking'' is yet to be broken. Intuitively, thinking involves chasing associations up, down and sideways through meshes of associations. We recognize some known pattern, perhaps from a few features. This makes us think of other things, which lead to other things. We travel ''up'' through successively higher levels of abstraction, back ''down'' to component features of abstractions, and ''across'' to related concepts.

In NeurOS, these concepts are modeled as follows:

- Set, Sequence and Temporal Sequence (SST) modules provide auto-associative matching of input features to more abstract learned patterns. Layers of SST modules are the ''up'' direction.
- The Reify module decomposes abstractions into their component features. Layers of Reify modules are the ''down'' direction.
- The ''across'' connections are ''up'' and ''down'' links among the SST modules, including feedback links.

Fig. 23 is a tiny suggestive example of this sort of associational thinking chaining.

This takes a little explaining. (Unfortunately, the current visual link routing algorithm in the NeurOS Designer needs more work to avoid visual confusion!) Keyboard input feeds both Set and Reify modules, as well as the Plot display. A collection of set patterns has been pre-loaded into the Set module's memory space, as indicated in the ''sets_2 memory patterns'' display at left. Both the Set module and the Reify module feed back to each other through a Filter module, which damps the feedback with a gain factor of 0.9. Entering the initial stimulus of ''vanilla ice cream'' starts the association chaining looping through the memory space

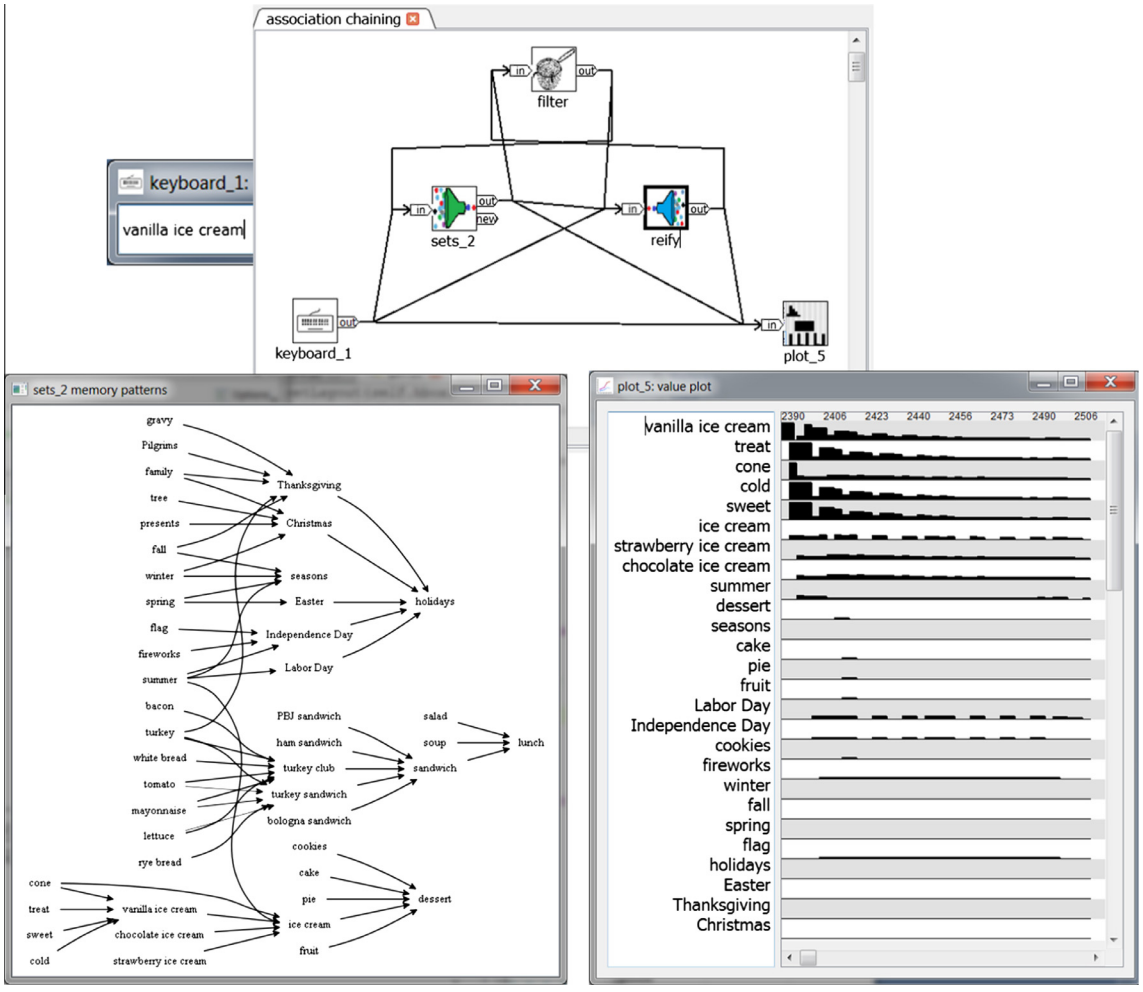


Fig. 23 Thinking — association chaining.

patterns. The Plot module output at the lower right shows the activation spread and decay over time from earliest (top) to latest (bottom).

Without any additional stimulation, the activations diminish over time and with ''semantic distance'' (number of links) from the original stimulus. However, if something else comes along, a new perceptual input, from some other thought path, or from random neural firing (e.g., using the Random Pattern module), some concurrence may lead to other cascades and recombinations.

This is just a speculative toy at this point. It is, however, suggestive of a research direction. Intuitively, a connectivity balance needs to be struck between segregation and commingling. We can only ''get ideas'' by fostering recombination of previously unconnected elements, but without being swamped with overwhelming meaningless combinations.

Example NeurOS applications

Biological brains achieve many different cognitive functions through connections among essentially the same building blocks and learning from experience. The same is true of NeurOS. This section briefly surveys a few of many prototype NeurOS application snippets that have been developed to date. These show how NeurOS modules and usage constructs can be combined for increasing levels and broad ranges of functionality.

These examples are not (yet) complete serious applications or new/better/best solutions to cognitive challenges. All could of course be built with custom coding. The point is that practical, usable solutions to these kinds of problems can be created and adapted quickly with reusable components using NeurOS. There is plenty of room to embellish and enhance these examples, all within NeurOS.

For ease of explanation, these example applications are small in the number of modules and links and in the volume of memory patterns. However, size and performance are limited only by available processing and memory resources courtesy of NeurOS' parallel dataflow architecture.

Motion tracking: follow the bouncing ball

Fig. 24 shows a crude motion tracking capability within a small fixed image field.

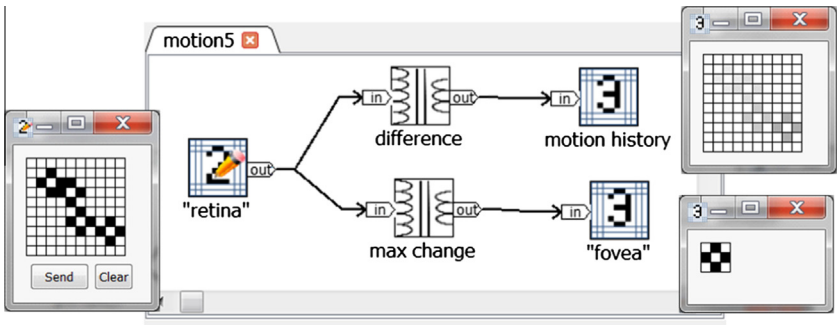


Fig. 24 Simple motion tracking.

The ''retina'' GridDraw module simulates successive input frames of a simple moving pattern. (Each little pattern was drawn and sent in sequence from the upper left.) The ''difference'' Transformer module computes the differences between successive input frames using simple matrix arithmetic, producing the time-decaying ''motion history'' output. The ''max change'' Transformer selects the 3x3 region of maximum change between successive input frames, shown in the ''fovea'' Grid Output display, which remains centered on the region of maximum motion within the larger scene. This simplistic approach might be suitable for simple robotic motion tracking within a fixed wide-angle field of view.

Biological vision systems are of course considerably more complex. The human eye has a lower-resolution motion-sensitive wide-angle retina surrounding a higher-resolution narrow central fovea sharing a common axis. Motion tracking there requires stimulating eye muscles as well as factoring out common-mode motion in the surrounding retina as the eye moves. This more complex processing can be built with NeurOS, but has not been done yet.

Note that this simple toy application does not use any memory patterns. As in the Where's Willie scenario, this is an example where fixed non-learning functionality is more efficiently implemented in custom algorithms rather than more general-purpose memory-based approaches.

Behavior

Behavior is a complex cognitive task and we can only scratch its surface here.

One formulation of behavior is as a collection of pre-condition/action pairs loosely connected in a hybrid open/closed loop way, as in Fig. 25:

Preconditions consist of external and internal observations along with an impetus to perform a particular behavior. The preconditions can include both excitatory and inhibitory elements. Together these stimulate a Set or Sequence pattern with a MOST or ALL pattern matching semantic. The SYN element finds the collection of actions that perform this behavior, and the REIFY element expands them into sets/sequences/temporal sequences of sub-behaviors and other action events. The effects feed back through open internal and external environment paths as well as more explicit closed sub-behavior event paths, to possibly stimulate other action steps. The combination of intentional sub-behavior events and observed precondition events yields a flexible multi-path behavioral mesh.

A NeurOS sub-graph implementing a form of this is shown in Fig. 26:

The behavioral task is for a (virtual) robotic claw to grasp an object. Two collections of set patterns record the behavior preconditions and actions. `grasp_behavior` is enabled by both `grasp_intent` and `object_in_range`. `close_thumb_behavior` is enabled by `close_thumb_intent` and inhibited by `thumb_pressure` (−1 weight). `grasp_action` is stimulated by `grasp_behavior` (synonym ''any'' set matching semantic) and reifies to `close_thumb_intent` and `close_finger_intent`. And so on.

The graph looks up potentially enabled behavior patterns. Those found then feed action patterns which reify into additional behaviors and actions, which feed back until ultimately leaf actions (`close_thumb_action`, `close_finger_action`) are emitted.

The two plots on the right show the iterated cascade of set matching and reification events leading to low-level actions. In case 1, both initial preconditions are present

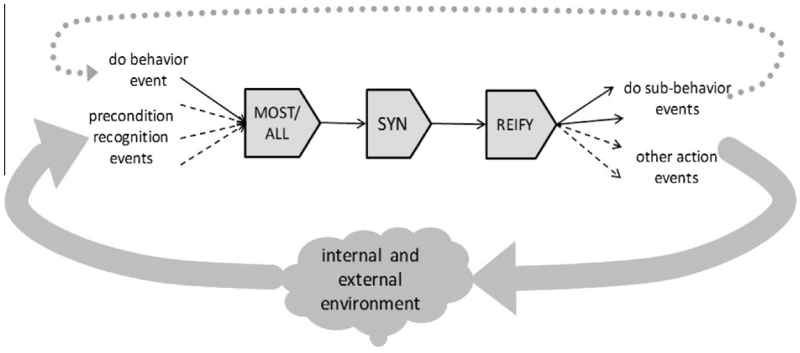
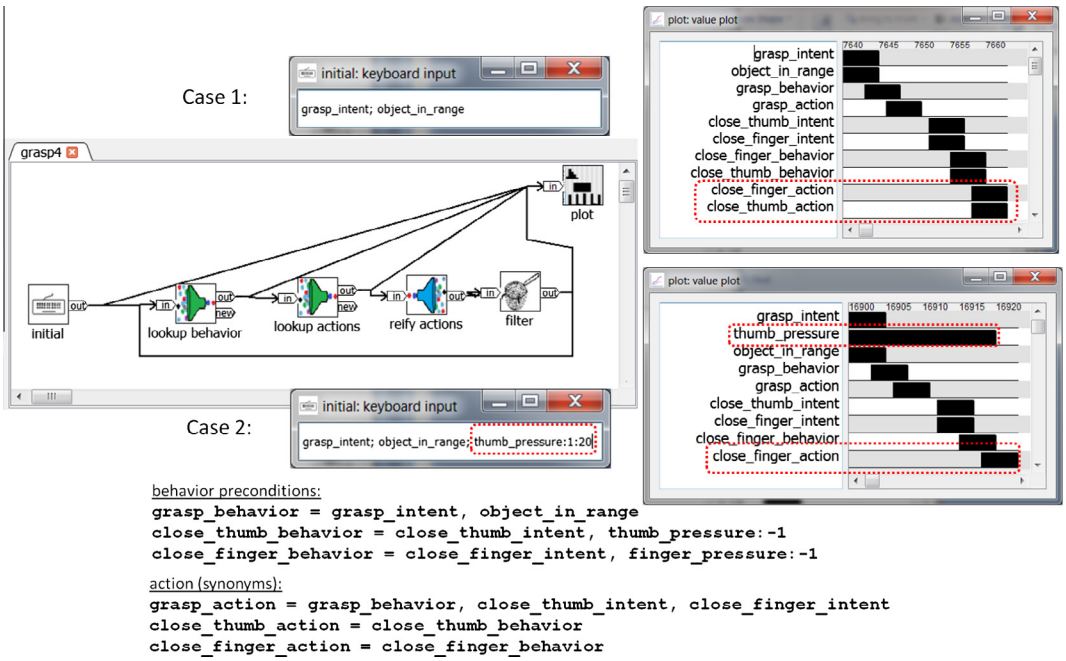


Fig. 25 Basic composable behavior assembly model.



behavior preconditions:

```
grasp_behavior = grasp_intent, object_in_range
close_thumb_behavior = close_thumb_intent, thumb_pressure:-1
close_finger_behavior = close_finger_intent, finger_pressure:-1
```

action (synonyms):

```
grasp_action = grasp_behavior, close_thumb_intent, close_finger_intent
close_thumb_action = close_thumb_behavior
close_finger_action = close_finger_behavior
```

Fig. 26 Simple behavior chaining.

and the chaining proceeds down to the level of generating the expected low-level actions of `close_thumb_action` and `close_finger_action`. In case 2, an additional external observation `thumb_pressure` is added (with a long-enough time duration to impact the behavior chain). The first iteration of processing is the same, but the presence of the `thumb_pressure` condition inhibits the recognition of `close_thumb_behavior` and hence no `close_thumb_action` is generated. (We can argue about whether this is the desired behavior. Should the whole grasp be inhibited if either thumb or finger feels pressure? Or just the individual thumb/finger movements as shown. This kind of mechanism can be adjusted to represent either policy.)

Stepping back a bit, this behavior architecture suggests that ''chaining'' of behavior steps within the brain seems to be rather loose: perform a step, and likely next steps are enabled by both the previous step and the internal or external changes, rather than explicit sequencing. Unlike procedural programming, a behavior pattern need not anticipate all the things that might go wrong or even all the consequences, expected or not. Other behavior patterns in the brain may detect and handle those situations. (When things go wrong, young children just stand there, lacking any learned behaviors to cope with the unexpected situation!) Notions of hierarchy and sequence and procedure and conditionals seem to be overlays projected from our procedural

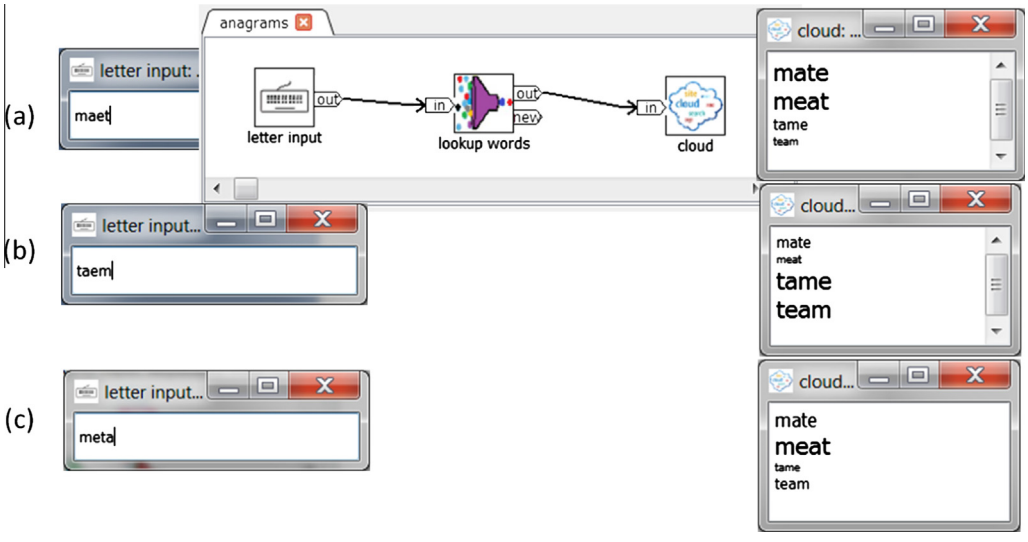


Fig. 27 Anagrams.

programming heritage. This loose open chaining also enables rapid adjustment to the unexpected and to discovering novel alternative behavior steps.

Learning behavioral elements seems similar to any other learning. Concurrence or sequential occurrence of inputs, actions taken and recognized patterns create and anneal behavioral set/sequence/temporal patterns just like any other patterns.

Anagrams

A first attempt at solving anagrams in NeurOS turned out to be surprisingly straightforward. Words are naturally represented as letter sequences. One way to think of anagrams is as misspelled words! Setting a NeurOS Sequence module's sequence match positional tolerance parameter to a high value allows it to discover all the permutations (known words) of the input letters (and perhaps other similar words). In Fig. 27, (a) shows a first run with the input letters ''maet''.

A more sophisticated version (not shown) might use a behavioral sequence to mimic human anagram solving strategies even more. Rewrite the letters (either on paper or in our imagination) in different sequences, thereby ''resubmitting'' them to our perceptual and pattern recognition machinery, until additional valid words ''pop'' more strongly, as shown in (b) and (c). This is an example of cooperation between Kahneman's ''system 1'' (fast, reflexive parallel pattern recognition) and ''system 2'' (slow, serial, deliberate) brain parts (Kahneman, 2013).

Crossword puzzles

At its simplest, a crossword puzzle clue is a synonym for a desired word. The corresponding puzzle grid entry shows the number of letters with possibly several letters filled in courtesy of previous efforts. The snippet of a simple crossword puzzle solving NeurOS application in Fig. 28 takes a typed input pair of a clue word and a letter pattern to be solved. The word to be solved is entered with explicit letters and a dot (·) for each unknown letter. Previously a set of synonym patterns (disjunctive Set patterns) has been learned, in this case {wharf, pier, levee, dock} among others. The clue word directly activates all its synonyms (via the disjunctive set synonym pattern feeding the reify module). The partial answer letter sequence stimulates possible matching known words. The concurrence of a synonym for the clue (''pier'') and a word spelling matching the partial pattern (''w. . .f'') creates the strongest match as shown in the tag-cloud module.

The above is like Kahneman's ''system 1'' fast, parallel, reflexive perceptual-oriented processing (Kahneman, 2013).

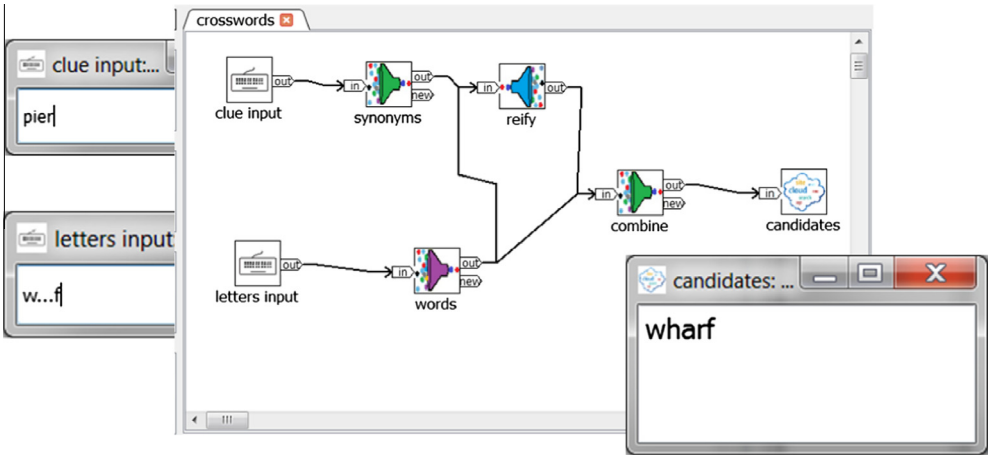


Fig. 28 Simple crossword puzzle solving.

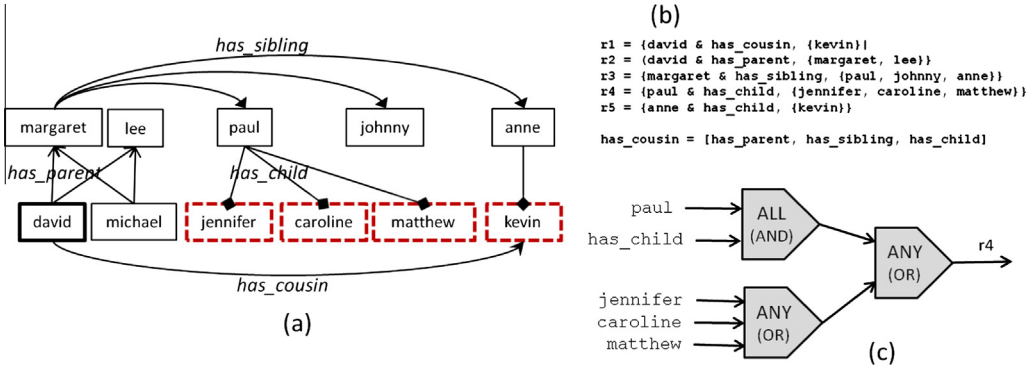


Fig. 29 Simple family tree and a priori known relationships.

An additional aspect to crossword puzzle solving is the dynamic relationship between Kahneman's ''system 1'' and ''system 2''. When we are ''stuck'' (that is when the above neural graph does not yield any decent word candidates), higher-level cognitive patterns may suggest a missing letter or two, merge those imagined letters with the perceived letters, and re-submit the combined letter sequence via downward feedback paths to the ''system 1'' machinery to generate some other candidates. A future NeurOS application will attempt to model this dynamic.

Cousins

How do we figure out, learn and remember our cousins? We learn some cousins from direct experience: ''This is your cousin Kevin''. We can also ''figure out'' who our cousins are starting from known fragmentary relationships. This is a kind of behavioral ''procedure'' sequence: find your parents, find their siblings, find their children. After travelling this path a time or two, we may then ''remember'' cousins we derived, and can avoid the longer inference path the next time.

Many representations and algorithms are possible to attack this cognitive task. How do human brains do it? Biological representations of 1:many binary relationships like has_cousin(A,B) are not yet understood. In particular, biological brains seem to lack any obvious analog to re-bindable programming variables or pointers, which eliminates many traditional programming approaches. How can NeurOS facilities, modeled on biological brains, do it? Here is one biologically plausible model in NeurOS.

Fig. 29(a) shows some fragmentary a priori known family relationships.

Using a set/logic notation in (b), each set r1—r5 models a 1-many synonym (disjunctive set) relationship between a concept of a person+relation and a set of people. Thus, in r4, ''paul & has_child'' is a synonym for the set of people ''jennifer'', ''caroline'', and ''matthew''. (c) suggests a plausible corresponding biological 2-layer construct: the first layer has two types of neurons: one fires with the conjunction of a particular person concept and a particular relationship concept, and the other collects the elements of the ''many'' side of the relationship. These both then feed a synonym (ANY/OR) neuron.

In addition, in (b), a ''procedure'' for deriving ''has_cousin'' is modeled as a sequence of other relationships to traverse: [has_parent, has_sibling, has_child].

''david'' and ''has_cousin'' are the input query; correct answers are shown in (a) above with dashed red boxes. Fig. 30 shows a plausible NeurOS application to answer this kind of query.

The input query initially follows the pink (upper) path, first triggering the conjunction of ''david'' and ''has_cousin'', which subsequently triggers the synonym set with the collection of people, in this case the singleton set containing just ''kevin'', the a priori known cousin. Two layers of reification then yield the expected cousin's name. Meanwhile, along the green (lower) path, the ''reify_sequence'' module emits events ''has_parent'', ''has_sibling'' and ''has_child'' in sequence. The blue (middle) feedback path recycles results of each relationship, each combining with a next relationship sequence step, to yield subsequent intermediate results (in parallel in virtual time). Ultimately, the deliberate procedural process yields the complete cousins result set, including the expected reactivation of ''kevin''. The ''plot'' module illuminates the progressive activities (note: each module adds a virtual 3-msec delay).

We should stop here and notice the importance of relative timing among events/signals. As in the Interruption and Concentration Loss usage, tinkering with this example has made it clear that delays and temporal extensions of various signals can yield different results. This is certainly a future avenue of exploration. Also worth noting is that the bounded-time appearance of recognized patterns serves

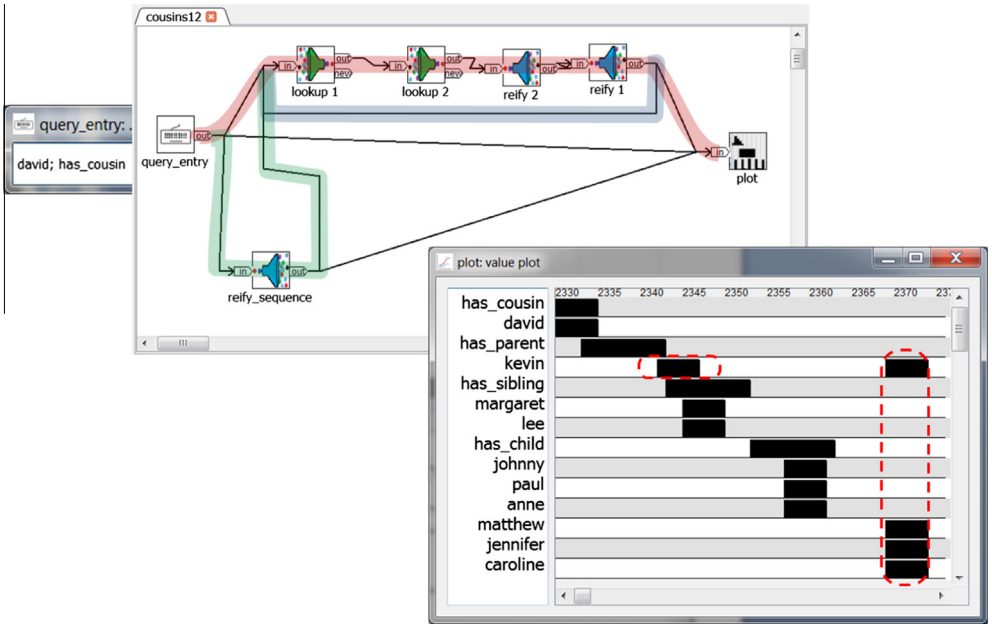


Fig. 30 Cousins.

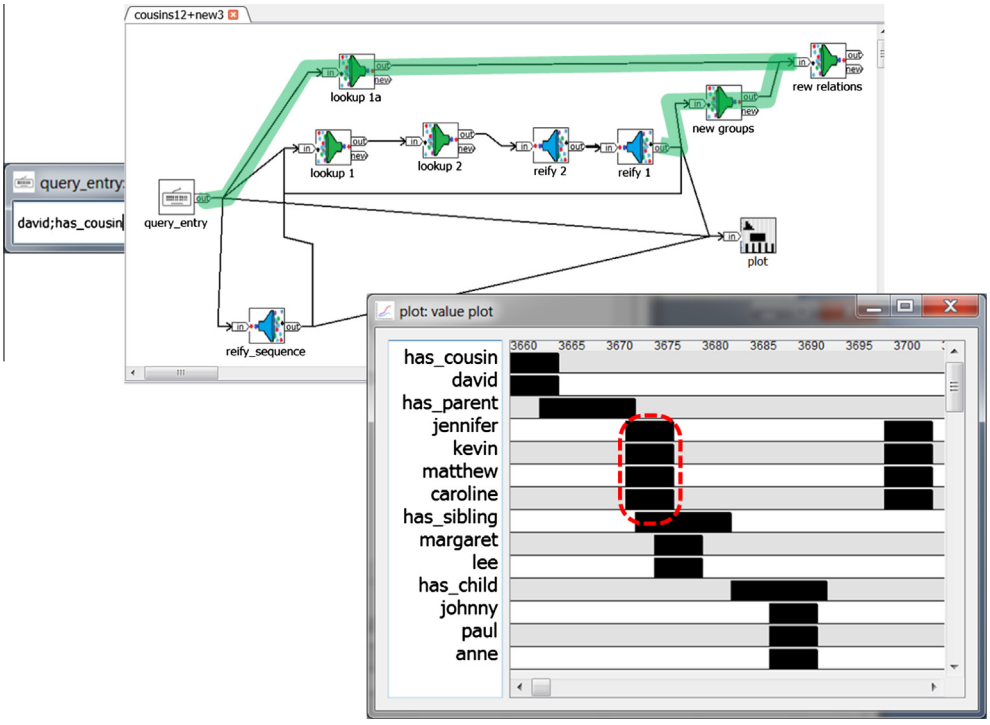


Fig. 31 Cousins with learning.

as the equivalent of ''temporary variables'' or ''intermediate results''. It is important the ''margaret'' stops being stimulated when the ''has_child'' sequence element appears: we do not want to include her children in the subsequent processing.

The next step is to learn the newly derived relationship instances, as shown in Fig. 31 below.

The new (green highlighted) path does several things. ''lookup 1a'' repeats the conjunction of ''david'' and ''has_cousin'' without the feedback loop of ''lookup 1''. ''new groups'' watches the concurrency of people names emerging from ''reify 1'', and when it finds a new active concurrent combination (of people, in this case, the desired set ''{kevin,jennifer,caroline,matthew}'') not previously seen before, creates a new distinct group. The ''new_relations'' module then creates the desired synonym relation between {david, has_cousin} and the new group. In a repeated trial of the original query, the new relationship formed after the original query is found directly and more quickly (Kahneman's ''system 1'') as shown in the red dashed outline in the plot display.

What's That Tune? — the enhanced version

A simpler version of this application was previewed above in Illustrative Example. This version in Fig. 32 adds the ability to replay a known melody (in its first-heard form) after entering either its name or a few recognizable notes (in any key, tempo, etc.).

Colors highlight the distinct paths for explanatory purposes. The ''melody in'' (MIDI Input) module generates NeurOS event messages for MIDI note-on and note-off events, with a timestamp derived from the MIDI timestamp, using

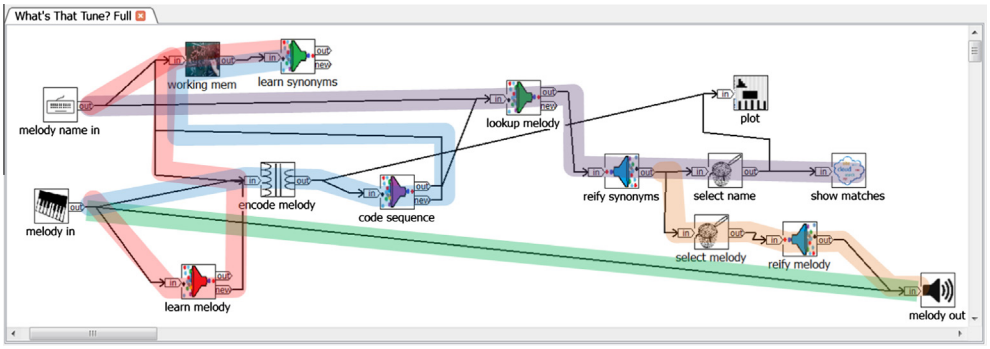
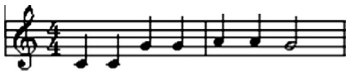


Fig. 32 What's That Tune? — the enhanced version.

the MIDI note number as the event ID, and the MIDI velocity (volume) as the event value, normalized to [0,1]. This raw melody input feeds several paths. The green path to the ''melody out'' (MIDI Output) module plays the melody notes as entered. The pink path remembers this signal in some detail in the ''learn melody'' (Temporal Sequence) module for later replay. The blue path uses a Transformer stage (''encode melody'') to quantize just the note-on events into an abstract invariant code event stream with IDs from {down, same, up}.[11] (This coding could be achieved with SST patterns mimicking biological neuron meshes, but is much more efficient in a couple of simple expressions.) So the opening phrase of ''Twinkle, Twinkle''.



is encoded as the sequence [same, up, same, up, same, down].

The ''code sequence'' (Sequence) module remembers this sequence as the abstract invariant signature for this melody. The ''working mem'' (Working Memory) module persists all three of a typed name, the melody code pattern ID, and the original melody temporal pattern ID, which are then associated as synonyms in the ''learn synonym'' module. Later MIDI input of any significant part of the melody, even with extra/missing/wrong notes and in a different tempo, yields a significant matching confidence with the code sequence pattern in ''lookup melody''. Later input of either a similar melody or the known melody name (purple path) stimulates the disjunctive (synonym) Set representing that melody, yielding the previously remembered synonym ID. Reification then generates the name, the ID of the invariant melody sequence, and the ID of the recorded Temporal Sequence pattern for the original melody. Further reification (tan path) of the original melody pattern into its component MIDI note messages feeds the ''melody out'' (MIDI Output) module which then plays the original melody.

Discussion

> ''All models are wrong but some are useful'' — *George E.P. Box*.

NeurOS and NeuroBlocks are infants. The usages and applications described are largely illustrative toys, suggestive but far short of definitive or production quality. As just the first few attempts to apply NeurOS to serious cognitive challenges, they necessarily suffer from all sorts of biases and limitations. Sufficiency, completeness, accuracy, functional and performance scaling remain to be stressed and measured.

Nevertheless, working with NeurOS and NeuroBlocks so far is encouraging. Some positive signs include:

- The core elements of event signaling, virtual time, neural directed graphs, module life cycle and built-in modules seem to be holding up to repeated usage in multiple domains.
- Little or no new invention has been needed to address the diverse range of cognitive tasks that have so far been addressed using composition of existing modules.
- Just as biology sometimes resorts to specialized structures, external interfacing/embedding/extension/customization points simplify integrating new sensory, motor and processing elements to address additional domains.
- Particularly encouraging is the extensive reuse of the Set and Sequence long-term memory patterns in a wide variety of contexts. These are emerging as key building blocks of cognitive processing and learning.
- Synergy seems to be emerging from combining multiple common usages and sub-assemblies, as in the example applications shown.
- The drag and drop visual editing and incremental iterative development processes have greatly accelerated the work reported here. Changes in parts of a neural graph have no effects on other parts except for explicit connections.
- Work so far has not required compromising the dataflow engineering rules that are key to parallel performance scalability and distributed processing.

Perhaps the real prize is accelerating insight, especially from staying ''close enough'' to biology without getting lost in details. Getting something to work in NeurOS suggests how it might work in biology. Speculation about biological constructs suggests how to attack a problem in NeurOS.

This frequent experience loop has so far yielded several ''Aha!'' moments, suggesting and supporting insights like these:

- Being able to build diverse intelligent functions from the same toolbox of reusable components strengthens the notion that brain components and structure are ''similar everywhere''.
- Cognitive abilities emerge from interconnection of components and assemblies.
- Structured limited connectivity in the forms of layers, feedback and distinct functional regions is crucial. Too much connectivity leads to confusion.
- Imagination commingles with sensory input via feedback connections and powers prediction.
- Different matching and learning parameter settings for long-term memory patterns serve quite different functions, and might provide clues to different roles for different neuron types and assemblies.
- Set, sequence and temporal sequence patterns appear to be solid primitive building blocks for intelligent capabilities. This strongly suggests that we look for biological analogies. Single neurons appear to match Set patterns well. Both dendritic computation and connected neuron chains might match sequence patterns.
- ''Timing is everything'', or at least is quite important. Several usages and examples work differently when relative timings among events change. In retrospect this seems obvious: biological brain mechanisms depend on time-dependent physical/chemical/electrical processes, and indeed work differently when conditions (e.g., neurochemical concentrations) change.

[11] This is a variant of a Parson's Code (Parsons, 1975).

- Signals representing intermediate results often need to fade quickly so as not to confuse subsequent processing. In the Cousins example application, each step activates possibly several people concepts (e.g., parents, siblings) that participate in the next inference step but should not participate in later steps.
- First impressions (i.e., ''good'' first examples) count for a lot. An initial feature set/sequence that forms a new pattern tends to dominate future learning.
- Cooperative mechanisms between Kahneman's reflexive ''system 1'' and deliberative ''system 2'' are becoming clearer: (a) system 2 processes produce imagined features to be commingled with sensory input to exploit system 1 pattern recognition; and (b) concurrent stimulation between inputs and system 2 results create Hebbian learning opportunities to remember new patterns for future system 1 direct use.
- Physical separation among segments of brain processing pathways may be necessary to segregate the broad-brush regional effects of neurochemicals.
- Some common words and phrases for mental phenomena take on deeper meanings: ''re-view'', ''re-member'', ''first impression'', ''see what we expect to see''.

Net, working with NeurOS seems to be a promising direction in computational neuroscience research and development.

Related work

Several other systems offer composition of modular components into executable graphs to perform general-purpose and neural/cognitive processing, offer libraries of reusable components, and have implemented some cognitive functions. IBM®'s CoreLet® (Amir, 2013) language is closely bound to the TrueNorth® (Cassidy, 2013) neural processing chips, where simple spiking neuron circuits are interconnected in cross-bar fashion and such assemblies are linked together to perform cognitive functions (Esser, 2013). The Neural Engineering Framework (Stewart, 2012) offers component creation based on parameterized distributed neural networks of spiking neurons. The related Nengo (The Nengo Neural Simulator) development environment offers graphical composition, execution and visualization. Several Nengo models address specific cognitive functions, and the large-scale Spaun (Bekolay et al., 2013) model emulates human brain region connectivity for several cognitive tasks. The MATLAB® Simulink® (Simulink, 2014) system provides extensive support for general-purpose system simulation, with extensive mathematical and signal processing functional blocks, plus more specialized neural-network components.

Status and directions

Work to date has concentrated mostly on the cognitive function and neural graph dimension, leaving much straightforward engineering work for the future. An initial implementation[12] of NeurOS architecture elements was used to develop and run the sub-assembly and prototype applications reported here.

The broad intent is to encourage a rapidly growing open broad and synergistic ecosystem. NeurOS Development continues along multiple dimensions:

- Documentation, distributable packages, exchanges for add-on modules, sub-assemblies and applications.
- Improvements and extensions to existing tools, run-time and built-in modules, and potential new tools.
- More module types, including interfaces to high-utility external facilities like the Microsoft Kinect, the Robotic Operating System, LEGO Mindstorms®, digital signal processing, classification and neural network technologies, access to web and data resources.
- Additional NeurOS run-time ports and implementations, for better multi-threading/multi-processing, distributed networking, custom hardware (e.g., GPU clusters and emerging ''neural chips'' (Abate, 2014)), mobile platforms and hybrid execution.
- And of course, lots and lots of applications and reusable sub-assemblies.

Perhaps most exciting are many potential research directions suggested and enabled: delving into complex layering/looping of learning (features/alphabets/vocabularies) and thinking, further developing the behavior dimension, Dehaene's reading paths (Dehaene, 2009), attacking additional hard cognition problem areas. Intriguing might be building a variety of ''virtual beings'' or ''multi-functional agents'' complete with multiple sensors and effectors learning from experience in a variety of real and virtual environments. I cannot help but be reminded of Braitenberg's seminal ''Vehicles'' book (Braitenberg, 1986), taken to the next level or two, and of Minsky's ''Society of Mind'' (Minsky, 1987).

[12] Python 3.3 on Windows 7/8, PyQt-based GUI and run-time system.

References

Abate, T. (2014). Scientists create circuit board modeled on the human brain. In *Stanford news service* (p. 30). <http://news.stanford.edu/pr/2014/pr-neurogrid-boahen-engineering-042814.html>.

Amir, A. e. (2013). Cognitive computing programming paradigm: A Corelet language for composing networks of neurosynaptic cores. <http://www.research.ibm.com/software/IBMResearch/multimedia/IJCNN2013.corelet-language.pdf>.

Bekolay, T., Stewart, T. C., Choo, X., DeWolf, T., Tang, Y., Rasmussen, D., et al. (2013). Spaun: A large-scale model of the functioning brain. In *Cheriton Symposium*. David R. Cheriton School of Computer Science.

Braitenberg, V. (1986). *Vehicles*. Bradford.

Cassidy, A. S. (2013). *Cognitive computing building block: A versatile and efficient digital neuron model for neurosynaptic cores*. <http://www.research.ibm.com/software/IBMResearch/multimedia/IJCNN2013.neuron-model.pdf>.

Dehaene, S. (2009). *Reading in the brain: The new science of how we read*. Viking.

Esser, S. K. (2013). *Cognitive computing systems: Algorithms and applications for networks of neurosynaptic cores*. <http://www.research.ibm.com/software/IBMResearch/multimedia/IJCNN2013.algorithms-applications.pdf>.

Hawkins, J., & George, D. (2006). <http://web.archive.org/web/20090206115723/http://numenta.com/Numenta_HTM_Concepts.pdf>.

Kahneman, D. (2013). *Thinking, fast and slow*. Farrar, Strauss and Giroux.

Kurzweil, R. (2012). *How to create a mind: The secret of human thought revealed*. Viking Penguin.

Marcus, G. (2014). *Computational diversity and the mesoscale organization of the neocortex*. Retrieved from MIT COMPUTER SCIENCE AND ARTIFICIAL INTELLIGENCE LABORATORY — Brains, Minds and Machines Seminar Series: <https://calendar.csail.mit.edu/events/126591>.

Marr, D. (1982). *Vision: A computational investigation into the human representation and processing of visual information*. W.H. Freeman.

MIDI. (n.d.). <www.midi.org>.

Minsky, M. (1987). *Society of Mind*. Simon and Schuster.

Parsons, D. (1975). *The directory of tunes and musical themes*. S. Brown.

Seung, H., Lee, D., Reis, B., & Tank, D. (2000). The autapse: A simple illustration of short-term analog memory storage by tuned synaptic feedback. *Journal of Computational Neuroscience, 9*(2), :171—85.

Seung, S. (2012). Connectome: How the brain's wiring makes us who we are. In *Mariner Books*. Houghton Mifflin Harcourt.

Simulink. (2014). <http://www.mathworks.com/products/simulink/>.

SoundHound. (n.d.). Retrieved from SoundHound: <www.soundhound.com>.

Sousa, T. B. (2012). *Dataflow programming concepts, languages and applications*. <http://paginas.fe.up.pt/~prodei/dsie12/papers/paper_17.pdf>.

Stewart, T. C. (2012). *A technical overview of the neural engineering framework*. University of Waterloo, Centre for Theoretical Neuroscience.

The Nengo Neural Simulator. (n.d.). Retrieved from Nengo: <http://www.nengo.ca>.

Wikipedia. (2014). *Edit distance*, 606682413. Retrieved from Wikipedia, The Free Encyclopedia: <http://en.wikipedia.org/w/index.php?title=Edit_distance&oldid=606682413>.