Neuro: presentation of an Adaptive, Modular, Fault Tolerant Neural Network project

July 19, 2020

Abstract

Description of an ongoing personal project to develop a flexible system integrating different approaches to solve a generic problem. The system is iterative, dynamically partitioning the input space-time, and assigning each partition to a specific module, picked on the basis of its performance. This allows to combine multiple strategies to solve the same problem, providing improved performance, fault tolerance, efficiency, and reducing its overall cost. The modules are completely generic, e.g. from hard coded sets of rules to time-dependent neural networks.

Author's email: pietrom16@gmail.com

Key words: modular, adaptive, neural network, deep-learning, spiking, genetic, mixture of experts, fault tolerant, time-dependent, real-time, GPGPU, OpenCL, SYCL, TensorFlow.

***Note**: this project is entirely personal, self-founded, developed during my free time, with no external support. If interested, I can let you know about my motivation behind it.*

Contents

1 Introduction

Complex problems usually require complex approaches to be solved. There are situations where these are too computationally expensive, or missing altogether because either unknown or non-existent. One technique to face this issue, which dates back to the 1990s and is still progressing today, is the Mixture of Experts model. Basically it is a "divide and conquer" method, where a complex problem is dynamically split into simpler sub-problems.

The purpose of this personal project is to develop a system based on the Mixture of Experts approach, extending it with the addition of modules based, among others, on deep learning architectures and on the Spiking Neuron Model, and setting its configuration at runtime as part of the learning process and offline with genetic algorithms. Each module will specialize on an elementary piece of knowledge, and multiple modules will be combined statically or dynamically to represent higher level concepts. This approach is not affected by the catastrophic interference/forgetting issue, so the whole learning process can be simplified. The capability to automatically extract symbolic knowledge from raw data can be added as a last step.

The long term objective is to build a system which imitates the brain at the functional level, i.e. replicating its inner mechanisms as long as they have an impact on its capabilities, an ignoring all the details that depend on its physical and biochemical properties and constraints.

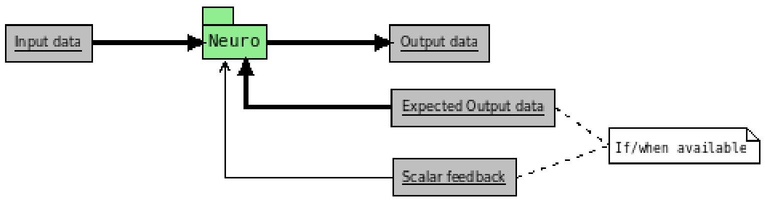
This project is still in an embryonic stage (the use of the present tense does not mean a feature is already available). The target hardware is generic, thanks to the use of Standard C++20 and OpenCL/SYCL, allowing it to run on both CPUs and GPUs. It is being developed in a Linux environment, but it is operating system agnostic at the source code level.

The project name, "Neuro", is intentionally generic, reflecting my personal experience in the field, ranging from neurosurgery, as a patient, to neuroscience.

2 Project objectives

Objectives this projects intends to reach:

- Design and implementation of a general purpose system able to integrate any number of different techniques to perform a generic task.
- The task is a mapping between an input and an output (n- and m-dimensional).



- The mapping can be time-dependent.
- The system has an internal memory, which can be fed back as part of the input.
- The system is recurrent; it can work without input, on the basis of what stored in its internal memory.
- The system aims to be memory efficient, high performance, (soft) real-time.
- The system behaviour can be both hard coded and acquired through learning from an optional training set.
  - Provide a seamless integration between symbolic/theoretical knowledge and acquired/experimental data.
- The software is developed with standard tools: Standard C++20, Boost, OpenCL/SYCL.
- Modular architecture:
  - The output is provided by multiple modules.
  - The modules compete and/or cooperate against/with each other.
  - Allows to train on new data only, without forgetting what learnt before.
- Adaptive system:
  - Explicit knowledge of the problem can be utilised, if available.
  - The system updates itself through interaction with the environment:
    * On-line/real-time: awake status.

        - Batch mode: sleep status/dreaming.
        - Evolution: Genetic Algorithms across generations.

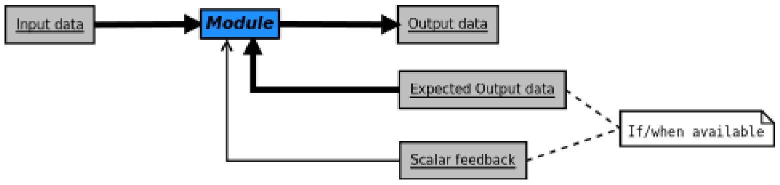
- Fault tolerant:
    - Provide multiple paths for data to follow.
    - Allow to distribute the execution on multiple nodes, either local or remote.
- High performance / real-time:
    - Multiple modules can have the same purpose, with different output quality and computation time requirements.
    - When the output is required, provide the best one obtained so far.
- Energy efficient:
    - A gating network preselects which modules have to process the current input.
    - The other modules are idle, with no energy consumption. For most tasks, most of the network should be off.
        - E.g.: when we see an apple: 1) we do not involve our auditory cortex to predict what sound an apple may produce; 2) we do not involve the part of the visual cortex dedicated to the recognition of completely different objects.
    - Human brain:
        - $8.6 \cdot 10^{10}$ neurons, $2.5 \cdot 10^{-7} J$ of energy needed for a single action potential.
            - Assuming a single action potential per second per neuron, the total power consumption of the brain would be more than $20KW$ (without considering synaptic and dendrite transmission contribution).
            - The effective total power consumption is $12W$. This means that only roughly 0.06% of the brain should be active simultaneously in normal conditions.
            - These data would suggest to use event-based (instead of time-based) algorithms when simulating the nervous system.
- Extensible:
    - Plug-in architecture: modules can be added/removed at compile time and run time.
    - Each module can have its own commercial/closed or open source license. The infrastructure can be either closed or, more likely, open source.

- Scalable:
  - The target hardware can range from mobile phones to workstations, and from peer-to-peer networks to clusters in the future.
- Interface with external applications:
  - Other AI tools:
    - Integrate, e.g., TensorFlow as an internal module.
    - Integrate Neuro within, e.g., TensorFlow.
  - Databases:
    - Source of training data.
    - High level data storage.
    - Database as a server (or client) interacting with an AI/NN client (or server).

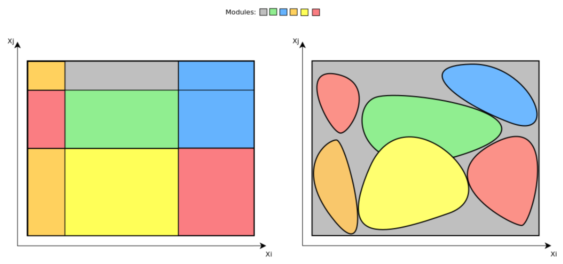
3 Planned capabilities

Development of a system with the following features:

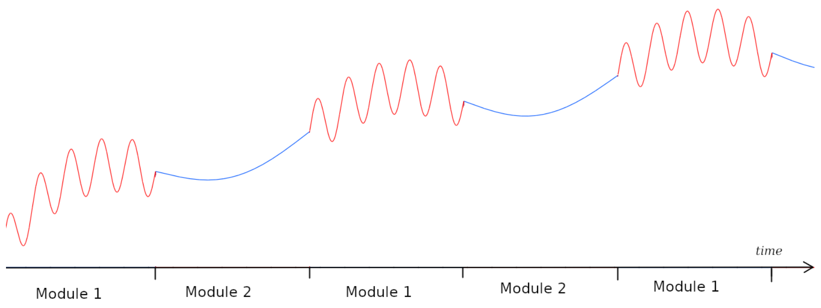
- Modular, hierarchical, multi-approach:
  - A module is a black-box, with: input, output, optional error feedback, timing requirements.



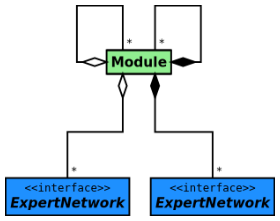
  - The purpose of each module is to process input data coming from (a subset of) the input space.
  - Modularity in space and time.
    - n-dimensional input space is dynamically partitioned depending on each modules' performance, e.g. (for the bidimensional case):



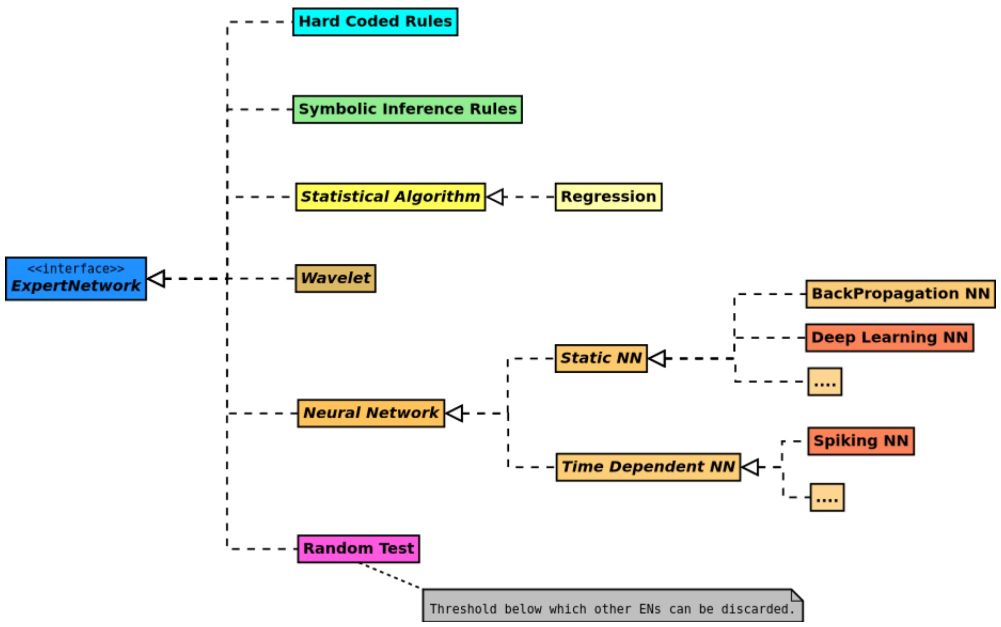
    - Time is partitioned according to emerging patterns, e.g.:
      - Quasi-periodic signal where module 1 is in charge of the first half of a cycle, and module 2 is in charge of the second half.



      - Module 1 keeps processing input data; when it identifies a specific pattern, it passes control to module 2.
  - Each module can work with a different learning strategy, and depending on the context the most appropriate can be selected:

        * Supervised: input and expected output available (cerebellum).
        * Reinforcement: input and scalar error available (basal ganglia).
        * Unsupervised: only input available (cortex).
        * Semisupervised: input and expected output available for a subset of the training data only.
        * Selfsupervised: input available and partial expected output is part of the input (e.g. a different sensor).
    - A module can be composed of a set of sub-modules and/or Expert Networks (EN).
        * 

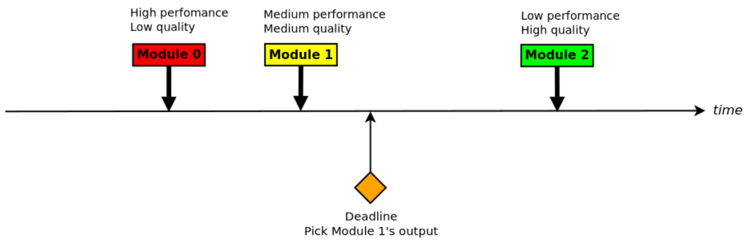
        * Sub-modules and Expert Networks can belong to the containing higher-level module or be shared with other modules.
        * Each sub-module/Expert Network has the same input/output interface as its parent module.
        * Possible implementations:
            · Explicitly encoded rules/equations - to include what already known about the problem, even in an approximate way.
            · Statistical algorithms - in case the data can be easily interpreted in the classical way.
            · Classic neuron models - for the typical pattern recognition tasks.
            · Spiking Neuron models - when time is one of the variables, and when it can provide better performance than classic models.
            · Wavelets - when a simpler, more efficient approach can substitute a neural model (e.g. preprocessing).
            · E.g.: the input-output relation may be approximated with a known symbolic formula, which can be added as an Expert Network with a hard coded rule; then, if/when training data are available, another Expert Network can contribute to improve the output working on the inputs where the hard coded algorithm is less performant. This can provide a seamless integration between symbolic/theoretical and acquired/experimental knowledge.



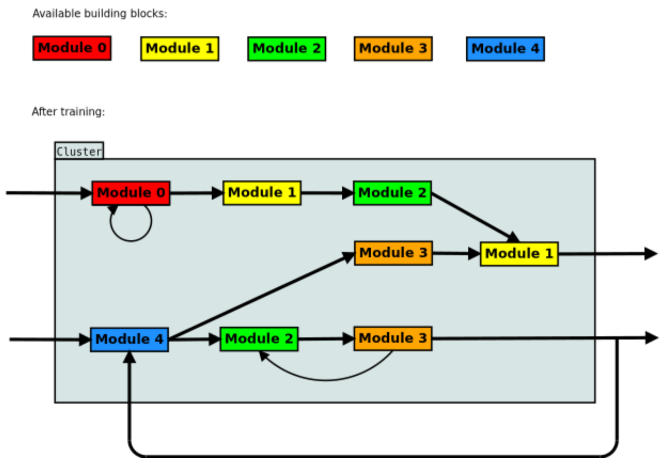
.

    * Specific implementations can be pre-selected on the basis of their past/predicted performance.

- Expert Networks coordination: Gating Networks.
    * Multiple Expert Networks (and multiple sub-modules) inside the same module have to be coordinated to optimise the system.
    * This can be performed by the Gating Networks, which have the following objectives:
        · On the basis of the current input, preselect which sub-modules/ Expert Networks have to be active (performance and energy efficiency).
        · On the basis of the current input, either weight the sub-modules/Expert Networks outputs and combine them together, training all sub-modules/Expert Networks proportionally to their contribution, or pick the best performing sub-module/Expert Network and return its output only, training this best performing block on the current input, together with a predefined number of second-best performing blocks; this allows other candidates to improve and eventually surpass the current best, avoiding being trapped in a local minimum.
        · Allow alternative paths to be followed, which can be picked when they provide better results/performance.
- Real-time behaviour:
    * Multiple modules can have the same purpose, providing different output quality with different time/resource requirements.

        * In a real-time context, when the output is needed the system can provide the best one obtained up to that point.
        * 

- Modules can be combined with each other:
    - Typically, a single module performs a simple task, multiple modules linked together perform complex tasks.
        * Modules can be seen as the nodes of a directed graph, which constitutes a cluster.
        * Each module can be copied and used in multiple clusters, and multiple times in the same cluster.
        * The clusters' topology is adaptive:
            · It can be parallel (e.g. Mixture of Experts), serial (e.g. Deep Learning) or unstructured.
            · It can converge to a stable configuration (during training).
            · It can change dynamically in real time (during a run).
            · It can evolve through a Genetic Algorithm (across generations).
            · The topology adaptation is feasible thanks' to the relatively limited number of modules forming a cluster.
        * A cluster can itself become a module with more complex features and be reused in the loop.
        * This architecture allows to:
            · Acquire capabilities that can be reused in different contexts, without the need of retraining from scratch, forming a flexible knowledge base.
            · Solve the catastrophic forgetting/interference problem.
            · Seamlessly combine the neural and logic approaches on different architectural levels.
            · Organise knowledge in a structured, mostly tree-like way.



- Time dependent processing: process input information with both time dependent and time independent strategies, and pick the best one.

- System "growth": newborn babies have blurry vision, which improves with time; this may help initial development phases in structuring the vision cortex so that it can properly process visual information (e.g. hierarchical architecture to discriminate global features from details); TODO Check how people temporarily blind since birth can see when able to do so (ref.: "Project Prakash, Illuminating Lives, Illuminating Science", "Unravelling the development of the visual cortex: implications for plasticity and repair").

3.1 Evolution / Genetic Algorithm development

Since the constraints Nature has are similar to the ones in technology (e.g. memory storage and processing, energy consumption, timing, ...), it can be a good start to imitate the first. Considering what information can be stored in the DNA in terms of its storage capacity (for humans, $6 \cdot 10^9$ bases, i.e. 1.5 GB, a third of which, 500 MB, focused on the nervous system), we can do the following deductions:

- Information genes/genetic algorithms **cannot** deal with:
  - Single neurons' details (4 bytes/neuron $\cdot$ 100 billion neurons $\Rightarrow$ 500 GB).
  - Synaptic connectivity and weights (16 bytes/synapse $\cdot$ 1 quadrillion synapses $\Rightarrow$ 10 PB).
  - Low level structure details.
- Information genes/genetic algorithms **can** deal with:
  - Common neurons'/synapses' properties and distribution:
    - Numeric coefficients.
    - Growth/connectivity patterns.
    - Internal mechanisms:
      - Evolution of coefficients of, e.g., a Fourier series.
      - Evolution of coefficients of a predefined expression, e.g.: $y = ax^{-1} + b + cx^3(d\log(x) + e\sin(x)) + ...$
      - Evolution of symbolic formulae, e.g. encoding parameters and operators in the DNA.
  - Large/medium scale brain structures.
  - Rules that control how the network structure evolves/grows/develops. These can be encoded with a suitable grammar.

So, I will not use genetic algorithms to store/process synaptic weights, but higher level features, even in terms of statistical properties.

A simple way to decide whether to process a feature through either learning or evolution is to observe if that feature is either typical of a subset of the population or common to all individuals of the same species.

Considering the speed with which genetic evolution takes place: if too high the system becomes unstable (an analogy can be made with organisms living in environments with high radiation levels).

3.2 Cortical Columns

Develop modules with the following neural connectivity pattern, typical of cortical columns:

- Neurons grouped in columns and layers (6 in humans).
- Preferential connectivity along the column (vertically), with some lateral connectivity with other columns, and feedbacks inside the same column.
- TODO - Investigate how different columns are connected to each other through the white matter; check if any patterns are present.

Automatically check through the Gating Network how this approach compares with others, used in other modules.

3.3 Information coding

Store/retrieve/transmit/process information using different coding strategies:

- Spike rates: average number of spikes per second.
- Spike timings: elementary information as an event (spike) at a specific time, using the Spiking Neuron Model.
- High level information: text, bitmaps, video, ...

Different modules can treat information using different approaches, which may be more or less suited to the specific context.

3.4 Postprocessing - Extraction of Symbolic Information

Extract explicit symbolic knowledge from the trained system for the I/O pairs that allow to do so. This would allow to:

- Get high level information about the internal mechanisms built during training, e.g. for debugging purposes.
- Automatically build new modules with explicit rules, which could provide better performance/smaller memory footprint.
- Work as an automated "research" system, able to provide mathematical/high-level descriptions of observed phenomena.

These tasks should be simplified by the fact that the system is already partitioned in modules/Expert Networks after training, and this partitioning reflects the structure of the original problem.

4 Planned implementation

- Combined use of CPU and GPU.
- Minimize data transfer to/from GPU memory (performance).
- Split network elements into low- and high-priority groups, according to specific criteria. Give preferential GPU access to (in this order):
  1. High priority elements, even if used rarely.
  2. Most frequently used elements.
  3. Elements connected to others already processed by the GPU.
  4. Parametrise elements' memory swapping frequency for optimisation.
- Serial processing/CPU: use event lists to schedule operations (e.g. *spike event lists*).
- Use a common language to code for both CPU and GPU: C++ algorithms (with execution policy: `std::parallel::opencl`, `std::execution::unseq`, `std::execution::par_unseq`), SYCL, SyclParallelSTL.
- Consider using existing tools (or parts of) for the low level optimisation (e.g. TensorFlow).

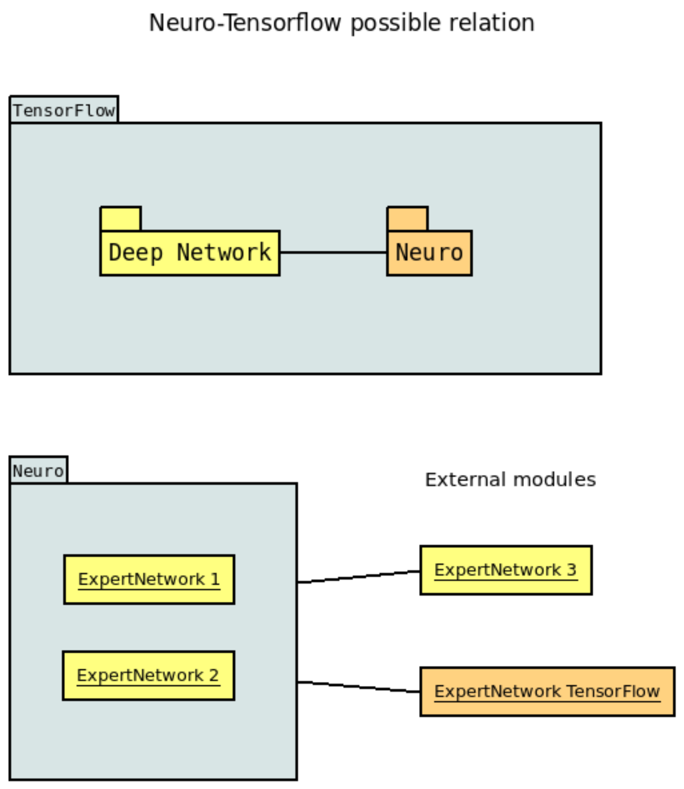
5 Integration with other tools

5.1 Integration with *TensorFlow*

Neuro's modules can be both internal and external. In the latter case, an interface will be developed to provide compatibility with the specific tool.

TensorFlow will be one of the tools which can be integrated. This will allow to incorporate all TensorFlow features, providing alternative paths when needed.

Another possibility is to integrate Neuro as part of TensorFlow.



5.2 Integration with Databases

A database could be used to store data instead of the filesystem; it would offer a higher level interface, the possibility to load a subset of the data through queries and to add/remove data dynamically as an independent process, automatic backups/redundancy/reliability, concurrent access by multiple instances of the Neuro tool (to perform multiple parallel training sessions from multiple executables with a shared memory).

Requirements:

- Multiprocess capability (this excludes, e.g., SQLite).

Candidates:

- SQL databases: general purpose (PostgreSQL).
- NoSQL databases:
  - Key-value store: as a configuration file substitute (Redis, Apache Ignite).
  - Document store: when the result of a neural process can be a pointer to a complex document (MongoDB).
  - Graph: to store the network structure maintaining its topology.
  - Object database: to store the network structure maintaining its object oriented structure.
  - Wide column: TODO (Cassandra, Scylla).

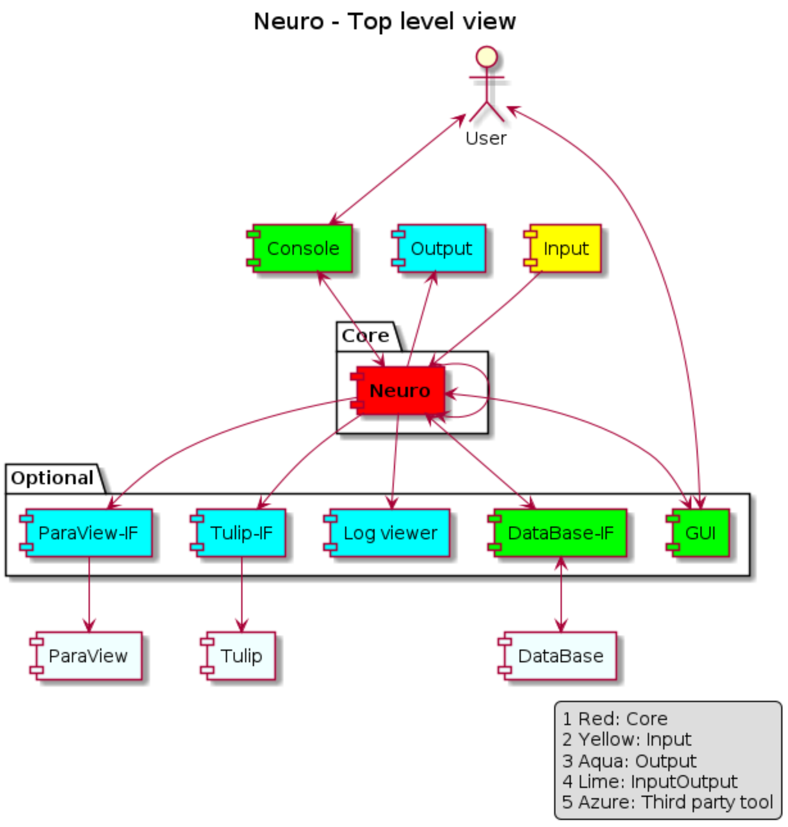
5.3 Integration with graphics tools

5.3.1 *ParaView*

Tool for analysis and visualization of n-dimensional data sets.

5.3.2 *Tulip Software*

Tool for graphs visualization and navigation.


Neuro - Top level view
User
Console
Output
Input
Core
Neuro
Optional
ParaView-IF
Tulip-IF
Log viewer
DataBase-IF
GUI
ParaView
Tulip
DataBase
1 Red: Core
2 Yellow: Input
3 Aqua: Output
4 Lime: InputOutput
5 Azure: Third party tool

6 References

- “Adaptive mixtures of local experts”. R. A. Jacobs, M. I. Jordan, S. Nowlan, G. E. Hinton. Neural Computation, 3, 1-12, 1991.
- “Hierarchical mixtures of experts and the EM algorithm”. M. I. Jordan, R. A. Jacobs. Neural Computation, 6, 181-214, 1994.
- “Spiking Neuron Models: Single Neurons, Populations, Plasticity”. W. M. Kistler, W. Gerstner, Cambridge University Press 2002.
- "Outrageously Large Neural Networks: The Sparsely-Gated Mixture-of-Experts Layer". G. E. Hinton et al. 2017.
- “A Theory of How Columns in the Neocortex Enable Learning the Structure of the World”. Jeff Hawkins et al. 2017.
- “A cortical sparse distributed coding model linking mini- and macrocolumn-scale functionality”. G. J. Rinkus, Frontiers in Neuroanatomy, 2010.

Document written with LYX / LATEX.