CNS: Cortical Network Simulator

A general, GPU-based framework for the fast simulation of cortically-organized networks.

Jim Mutch
Center for Biological and Computational Learning
MIT Dept. of Brain & Cognitive Sciences
Some Current Deep Architectures

**Convolutional Networks (Lecun)**

Lecun et al. (1995)

**HTM (Hawkins)**

Dileep George (2008)

**DBNs (Hinton)**

Hinton et al. (2006)

**HMAX (Poggio)**
Some Challenges with Deep Models

1. While single cells typically do simple things, keeping track of connectivity can get complicated.

2. They are computationally demanding.

*GPU programming helps address the second problem, but exacerbates the first.*

*One option: do your experiments in a slow but flexible language and then port your best models to GPUs.*

*Better option: use a framework that is both flexible and fast.*
“Cortical” Models and CNS

• “Cortical” model class:
  • Any number of layers.
  • Layers are N-dimensional arrays of cells. (N may differ from layer to layer.)
  • All cells in a layer are of the same type (perform same operations, maintain same variables).
  • Many different connectivity patterns.
  • Feedforward or recurrent, static or dynamic.

• Goals of CNS:
  • Flexibly cover this entire class.
  • Minimize complexity of code without loss of generality.
  • Automatically run on GPUs.
Working with CNS

- Network structure defined in a MATLAB struct:
  - Number and type of layers.
  - Dimensionality and size of layers.
  - Connectivity.
  - Initial values of variables.

- The only procedural code you write (in C++) is that executed by a single cell.

- Cell code calls macros to read/write its variables, find other cells, read their variables.
  - Makes it possible to compile for CPU or GPU.

- Details of who is connected to who, how memory is organized, etc. all handled by the framework.

```c++
m = struct;
m.layers{1}.type = 'ndp';
m.layers{1}.size = {100 100 50};
...
m.layers{2}.type = 'max';
m.layers{2}.size = {30 30 50};
...
```

// Retrieve the filter size.
int ySize = WEIGHT_Y_SIZE(ZW);
int xSize = WEIGHT_X_SIZE(ZW);

// Find RF in the previous layer.
FIND_LAYER_Y_NEAREREST(ZP, ySize, y1, y2);
FIND_LAYER_X_NEAREREST(ZP, xSize, x1, x2);

// Compute response to the filter.
float v = 0.0f;
for (int j = xSize - 1, x = x1; j >= 0
  for (int i = ySize - 1, y = y1; i >= 0
    float p = READ_LAYER_VAL(ZP, y, x);
    float w = READ_WEIGHT_VAL(ZW, i, j,
    ...
```
Challenge #1: Connectivity

- CNS addresses two significant challenges in working with cortical models.
- Challenge #1: while single cells typically do simple things, network connectivity can get complicated – mainly due to convolution, i.e. limited receptive field size along at least one dimension.
- Occurs for stimulus dimensions (often spatial ones) where there is topography (proximity in layer = proximity in feature space).

modified from Blasdel & Salama (1986)
Topographic Dimensions

• A dimension over which the receptive fields of downstream cells are limited.
A dimension over which the receptive fields of downstream cells are limited.
• A dimension over which the receptive fields of downstream cells are limited.
What Connectivity Problems?

- Annoying problem:
  - Repeated convolution, with edge loss & subsampling → lose track of where higher level units are centered in the original input space.

- Hard problem: **convergence**.
• General problem: an average cortical area gets input from around 10 others… how to define local convolution-type operations over multiple layers having different resolutions & edge loss?

• One solution: forget convolution and just enumerate synapses.
  • Consumes memory and time.

• Another (everyone’s favourite): only study the subset of models for which the problem doesn’t arise.

Felleman & Van Essen (1991)
Connectivity Under CNS

- Integer indices of cells within layers are not meaningful across layers.
- Under CNS, for topographic dimensions, each cell knows its position in a real-valued coordinate space that is meaningful (e.g. retinal position).
- When a cell executes, it can call macros to find its input cells:
  - e.g. “find the 4x4 cells nearest me in layer 1”
  - e.g. “find all cells within 0.03 units of me”
Challenge #2: Speed

- Any cortically-inspired model is going to be susceptible to parallelization.
  - Neurons are slow: the brain’s computational power derives from parallelism.
- Modern GPUs can support millions of threads and execute any 480 of them simultaneously. So: just map 1 neuron/unit to 1 thread.

- But it’s not quite that simple....

Why aren’t there PCs with 512 CPUs?
  - Acting independently they would overwhelm the memory system.

GPU processors need to work in coordinated groups of 32.

GPU programs only run fast if:
  - All 32 threads in a group are executing the same instruction.
  - If they’re accessing main memory, locations must be contiguous.

In a sense, instead of 512 scalar processors, you actually have 16 vector processors that operate on 32-element vectors.

But you can program them as if they’re scalar processors.

If above constraints are not observed, code still runs, but slower.
  - Better than having to write vectorized code.
  - But still not easy.
So: GPU programming shortens run time but can lengthen development time.

But under CNS, models can run automatically on NVIDIA GPUs without modification. How does this work?

- Everything in CNS is defined parametrically except the code a single kernel thread executes.
- Even that code only communicates with its environment via macros.
- When compiling for a CPU, kernel macros expand into code that accesses data structures in host memory.
- When compiling for a GPU, those same macros expand into code that accesses GPU memory.

GPU details that CNS hides from you:

- Memory management:
  - Class of memory (global, texture, constant, shared, ….)
  - Host-GPU transfers.
  - Alignment and addressing.
  - Dimension mapping (N-D to 2-D, texture packing).
- Thread management.
- The GPU programming API (CUDA).
Example Package: 3-D Convolutional Networks

- Developed with Srini Turaga et al. from the Seung lab.
- Learns to segment cell bodies in 3-D scanning electron microscope stacks.
- Implements the backpropagation algorithm: forward pass, backward pass, weight update.
- Speedup of about 100x over a single CPU.
- 300 lines of code, about 2d development time.
Example Package: 3-D Convolutional Networks

Turaga et al. (2009)
• The Poggio lab’s object recognition model based on the ventral visual pathway.
• Similar to some of Lecun’s convolutional nets for digit/object recognition, but:
  • hardwired layer 1 features
  • MAX pooling
  • thousands of learned features
  • forward pass only
  • multiresolution
• Most recent Caltech 101 results are around 71%.
• CNS version is about 100x faster than older implementation on a single CPU.
• Dynamic simulations using biophysically realistic neurons. Models are iterated through many time steps.

• Each neuron (or compartment) maintains a full set of Hodgkin-Huxley state variables.

• Connectivity is irregular: each cell has an explicit list of synapses.
  • Each synapse also maintains its own state variables (channel states, conductances).

• For a model with 10,000 neurons and 330,000 synapses, CNS was able to process 5,000 time steps per second on a GTX 285 GPU.

• For dynamic models, CNS can pull out time series of variables as they change.
Another Application: Action Recognition in Video

- Basically HMAX with a time dimension added.
- Spatiotemporal filters.
- Initial port to CNS: about a day’s work.
- Speed increased from 0.55 frames/sec to 32 frames/sec (real time).

Jhuang et al. (2007)
Temporal Shifting

- Special handling for a “time” dimension.
- Enables streaming video with operations that span multiple frames.
- Special handling for a “time” dimension.
- Enables streaming video with operations that span multiple frames.
Temporal Shifting

- Special handling for a “time” dimension.
- Enables streaming video with operations that span multiple frames.
Temporal Shifting

- Special handling for a “time” dimension.
- Enables streaming video with operations that span multiple frames.
Internals: Mapping N-D to 2-D Textures

\[ \text{dnames} = \{ 'f', 't', 'y', 'x' \}; \]
\[ \text{dims} = \{ 1, 2, 1, 2 \}; \]
\[ \text{dparts} = \{ 2, 2, 1, 1 \}; \]

Also: “dimension splitting”

\[ \text{dnames} = \{ 'f', 'y', 'x' \}; \]
\[ \text{dims} = \{ [1, 2], 1, 2 \}; \]
\[ \text{dparts} = \{ [2, 2], 1, 1 \}; \]
• One texture per cell field.
  • If caching on.

• Each block = one layer
  (already flattened to 2-D)

(Packing algorithm could be improved)
To Do List

• Open up data structures to allow manipulation via:
  • MATLAB’s new gpuArray type
  • Arbitrary (non-CNS) CUDA code
Resources

• Has links to:
  • Tech report (high level)
  • Programmer’s manual (full details)
  • Code download & installation instructions
  • Example packages
  • Forum
Collaborators

Ulf Knoblich (Ph.D. student, Poggio lab, MIT)
  • Spiking models

Tomaso Poggio (MIT)
  • My supervisor

Other early adopters:
Hueihan Jhuang, Sharat Chikkerur (Poggio lab)
Srini Turaga, Kannan Venkataraju, Matt Greene, Viren Jain (Seung lab)