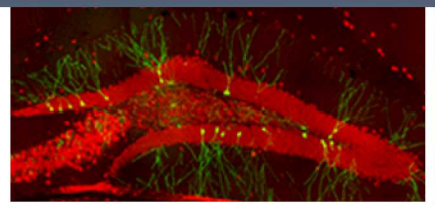
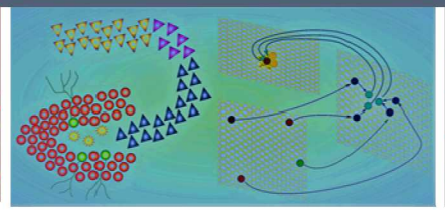
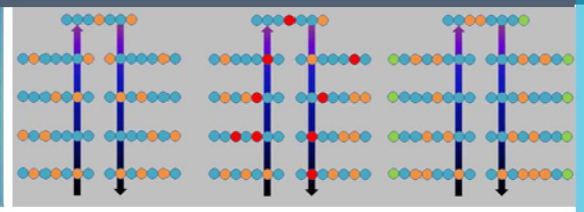
SAND2019-12044PE

# Fugu: Algorithm Development for Neuromorphic Hardware







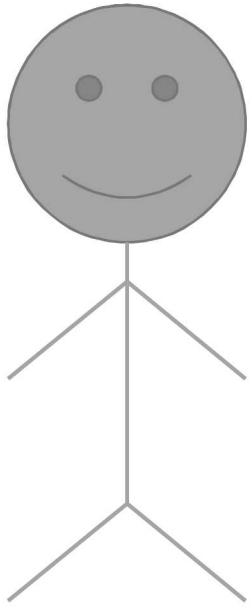
Presented by: Srideep Musuvathy

Team: Brad Aimone, Suma Cardwell, Frances Chance, Ryan Dellana, Yang Ho, Leah Reeder, William Severa, Craig Vineyard, Felix Wang

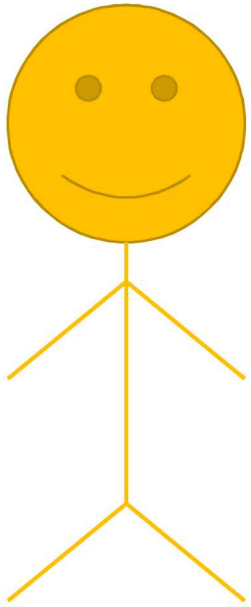
Sandia National Laboratories is a multimission laboratory managed and operated by National Technology and Engineering Solutions of Sandia LLC, a wholly owned subsidiary of Honeywell International Inc. for the U.S. Department of Energy's National Nuclear Security Administration under contract DE-NA0003525.

*Spiking neuromorphic hardware will have significant, widespread impact if we can both demonstrate compelling utility (algorithms), and facilitate usability (software)*

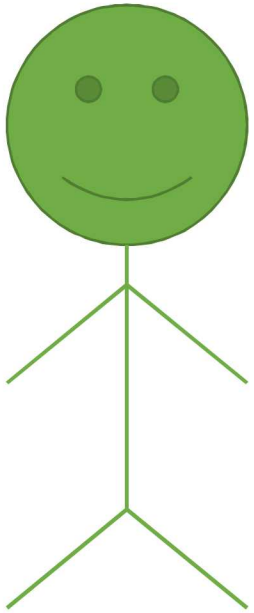
# 3 types of neuromorphic users



Typical Computer Scientist

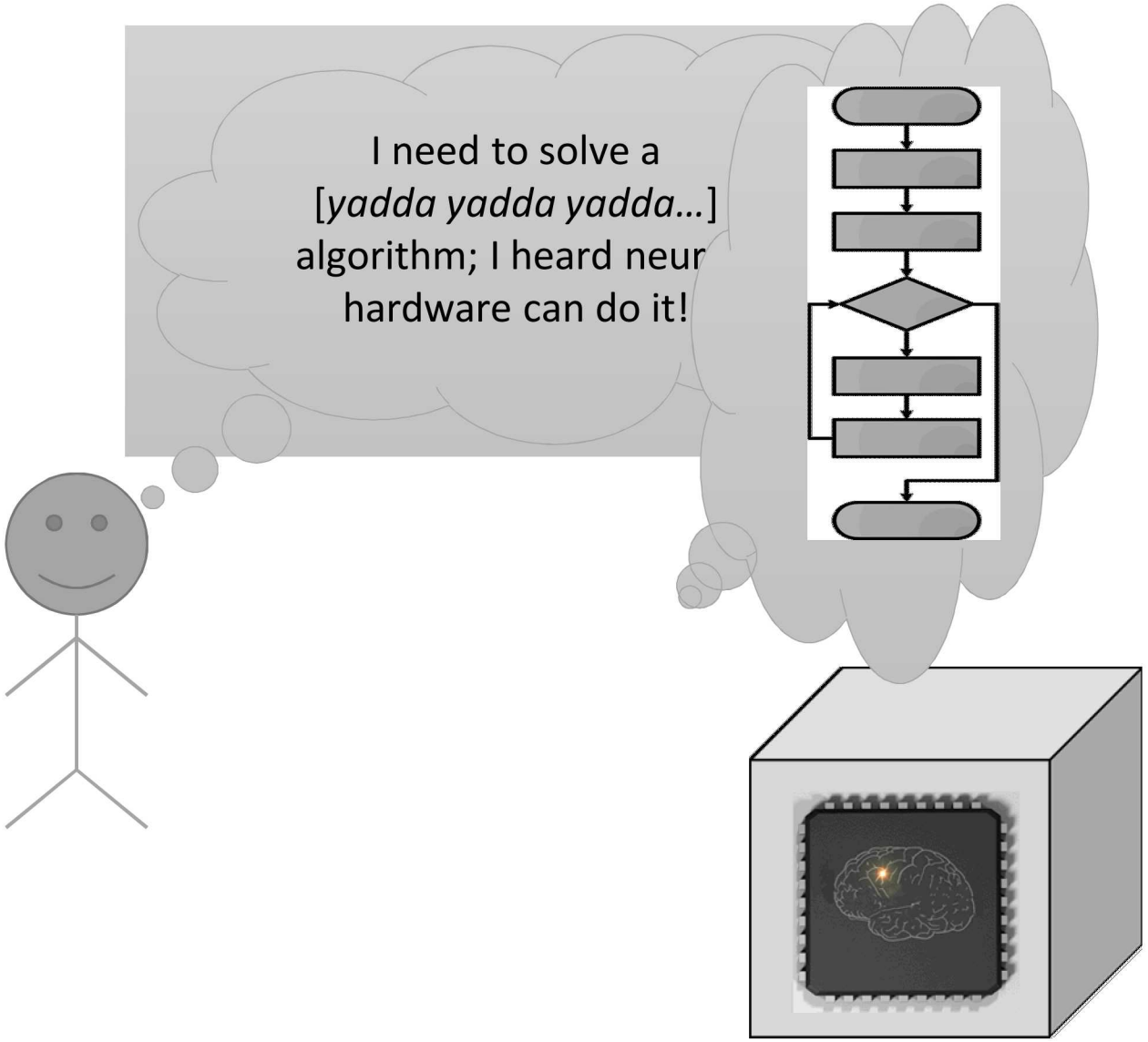


Neural Algorithms Researcher



Neural Architecture Developer

# 3 types of neuromorphic users…



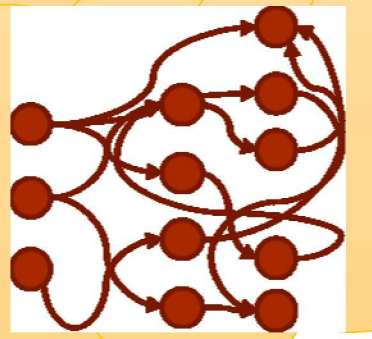

```
Algorithm yadda yadda yadda
  Initialize with data
  a=f1(start)
  b=f2(a)
  while b != 0
        c=f3(b)
        b=f4(c)
  end
```

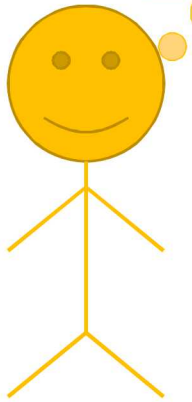
Goals for this user:

- ❑ Never have to learn anything about neurons!
- ❑ Program like any other machine
- ❑ Take advantage of libraries and great performance!

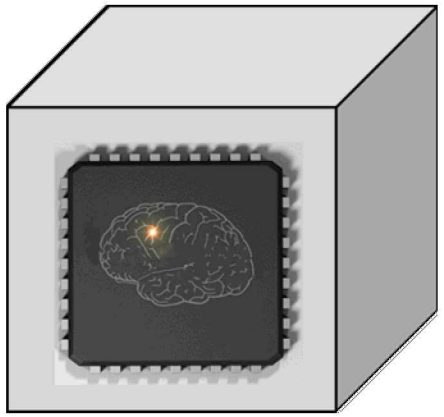
# 3 types of neuromorphic users…

I have an idea for programming a neural algorithm to solve the first *yadda* in the [*yadda yadda yadda…*] algorithm!





```
Neural algorithm yadda
  Initialize neurons
  pop1 = neurons(3)
  pop2 = neurons(4)
  pop3 = neurons(3)
  pop4 = neurons(2)
  conn1_2 = synapses(5)
  conn2_3 = synapses(2)
```

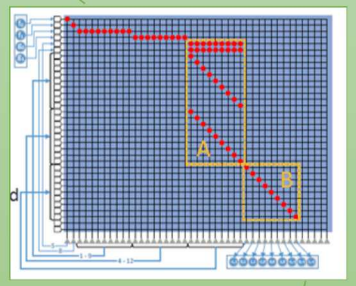


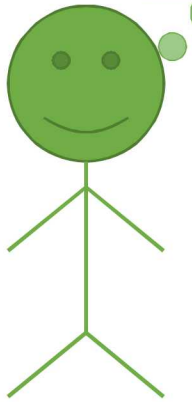
Goals for this user:

- ❑ Can create spiking neural algorithms!
- ❑ Program independently of underlying hardware
- ❑ Create libraries for first set of users.

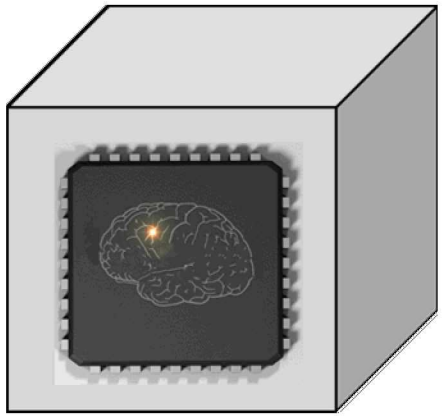
# 3 types of neuromorphic users…

I know how to tailor the hardware platform to enable sparse synapses to be more efficiently accessed for *yadda*





```
Module neuralCore(pop, syn)
for n in core
  neuron(n).v=pop(n).v_ini
  neuron(n).v_t=pop(n).v_t
for s in core
  synapse(s).w=syn(s).w
  synapse(s).source=syn(s)
  synapse(s).targ=syn(s).t
```



Goals for this user:

- ❑ Can compile algorithms onto specific hardware platforms
- ❑ Interested in optimizing algorithms for hardware constraints
- ❑ Create libraries for first set of users.

# Fugu aims to bring neuromorphic solutions to general computing world

Typical Computer Scientists

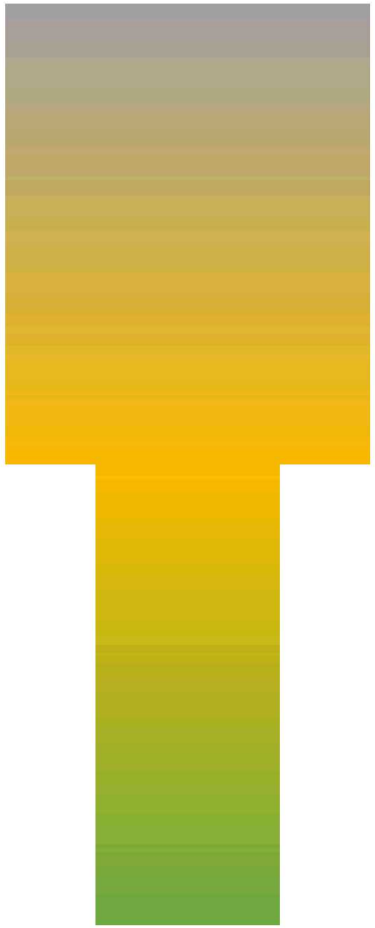
Wants to program with libraries

Neural Algorithm Researcher

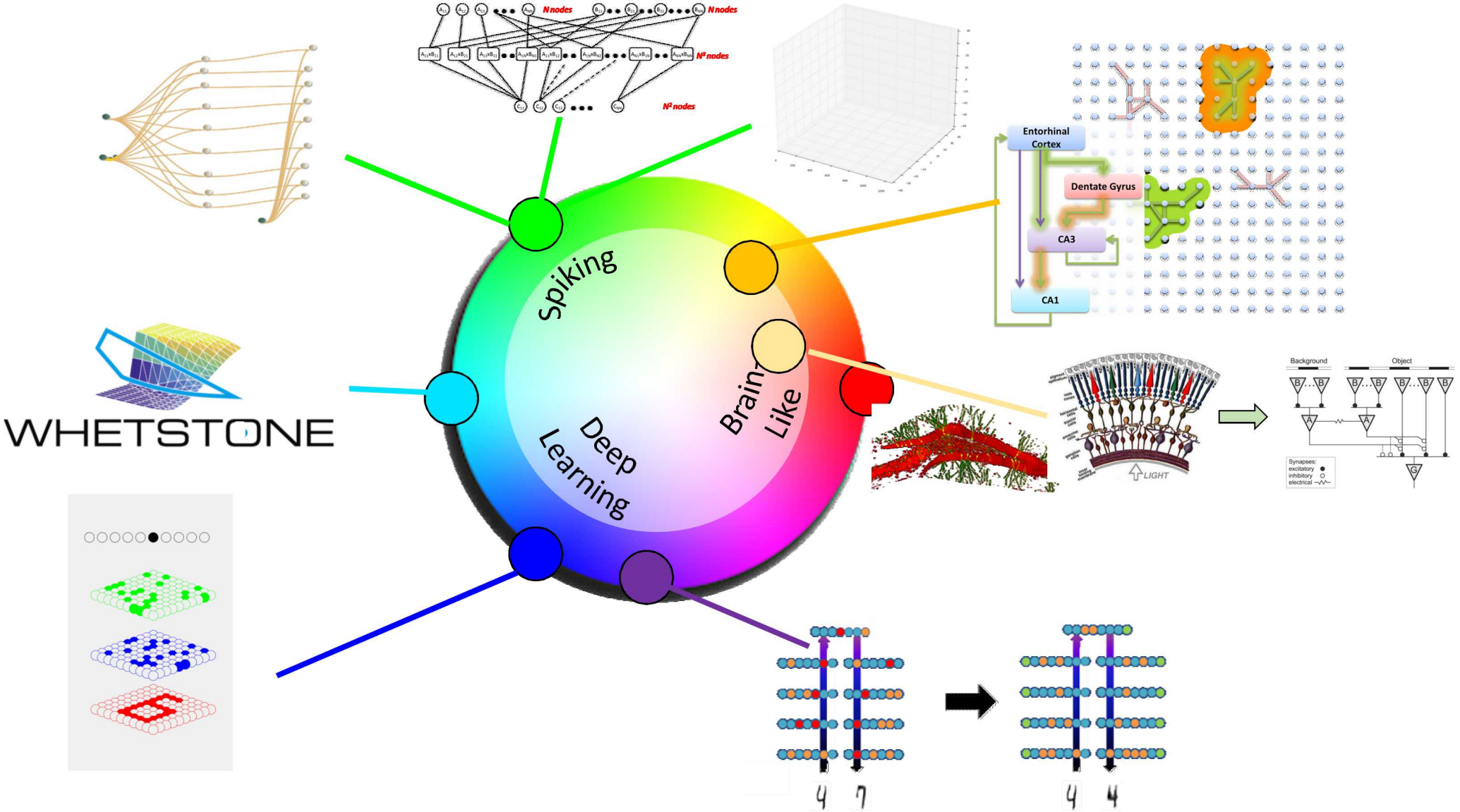
Wants to program with neurons

Neural Architecture Developer

Wants to program hardware directly



# Potential for neuromorphic computing may extend farther than anticipated



# Emerging list of reasonable spiking neural algorithms and kernels

**Machine Learning**

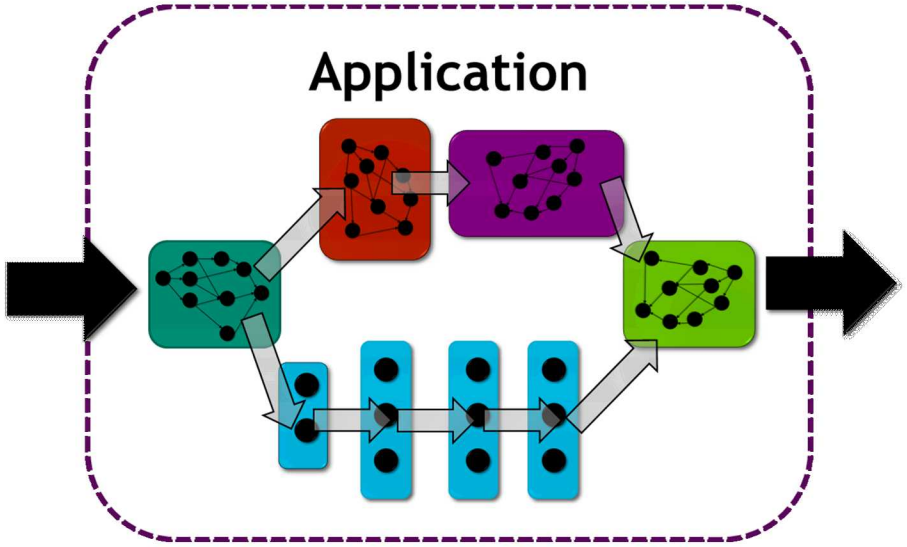
Whetstone

Convolutions

k-Nearest Neighbor

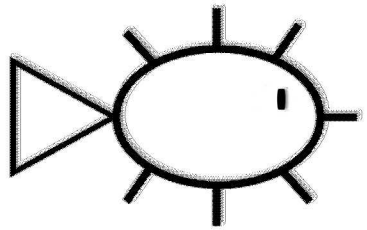
*Support Vector Machines*

- Many ‘kernels’ used for common neural computation are important for conventional algorithms as well
- Neural hardware is capable of reasonable performance on many non-ML kernels as well
- View neural algorithms as ***composable*** from linking together neural circuits to solve broadly useful kernels

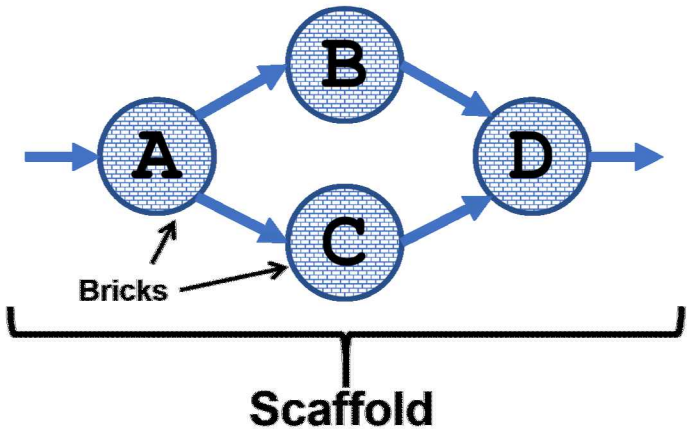


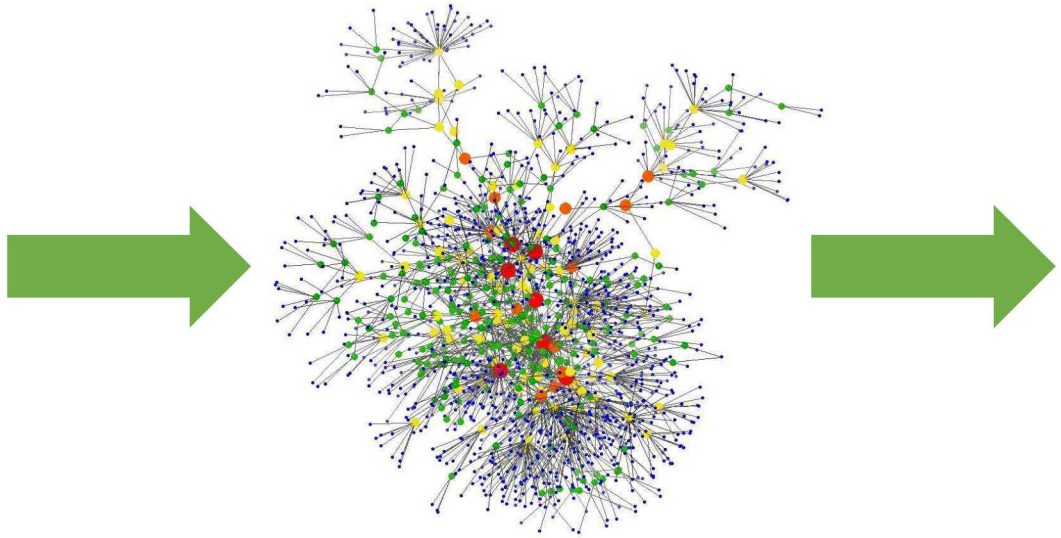
# Challenges

- For spiking neural networks, it is (very) hard to
  - Implement someone else's network
  - Integrate multiple kernels into an algorithm
  - Port networks designed for one platform to another

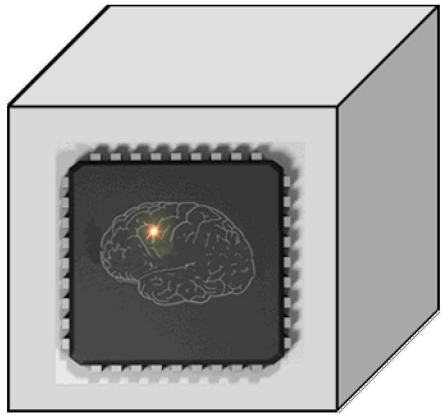


```
Algorithm
Insert Brick A, input IN     //A=f_A(in)
Insert Brick B, input A      //B=f_B(A)
Insert Brick C, input A      //C=f_C(A)
Insert Brick D, input B, C   //D=f_D(B, C)
```



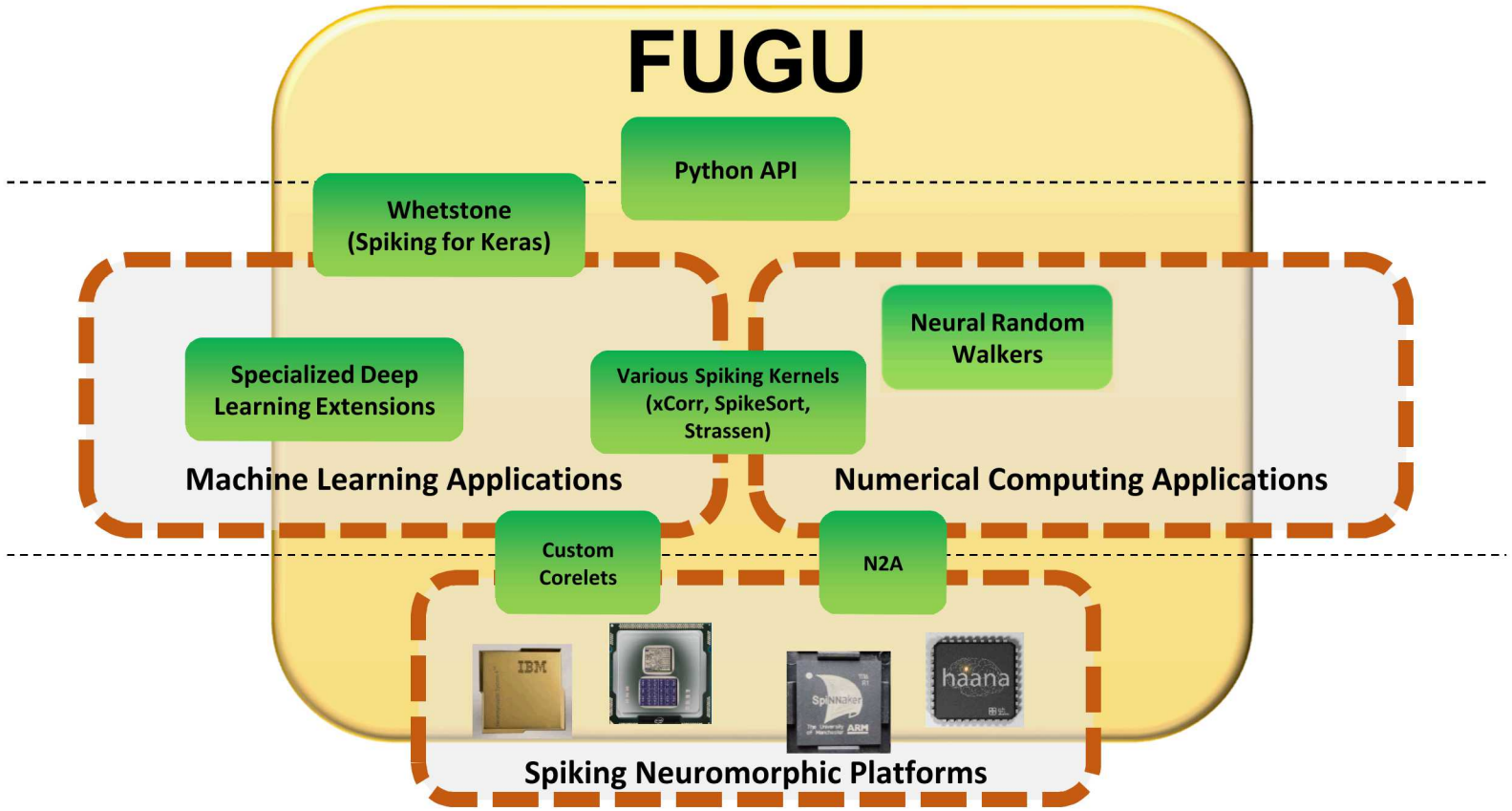



*Not actual syntax or network*



# Fugu Overview

# Fugu Overview

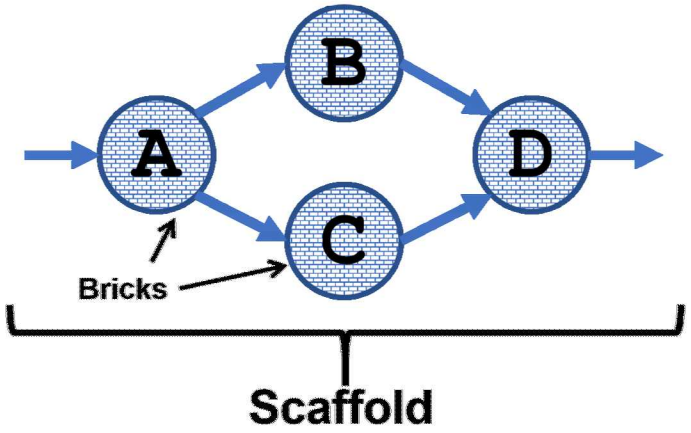


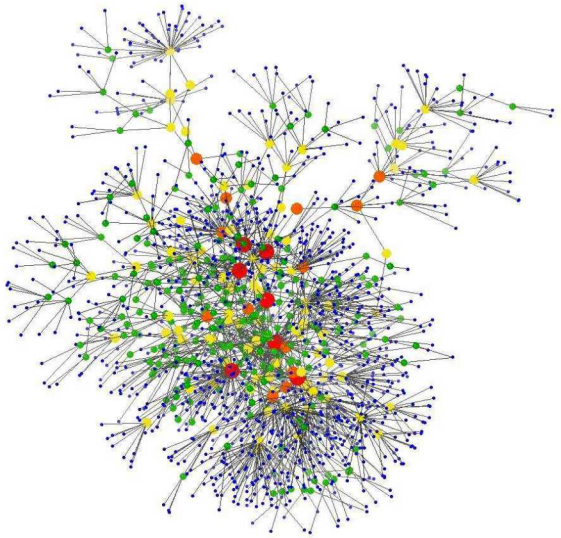

Under development
Collaborators welcome!

# Fugu Overview - Goals

- *Common linking framework for implementing spiking algorithms*
  - Leverage and combine community's recent progress
- *Hardware-independent intermediate representation*
  - Ability to rapidly change platforms
- *Procedural definition of network connectivity*
  - Kernels and algorithms can scale according to problem size
- *Flexible, pre-determined communication methods*
  - Simple interactivity

```
Algorithm
Insert Brick A, input IN     //A=f_A(in)
Insert Brick B, input A      //B=f_B(A)
Insert Brick C, input A      //C=f_C(A)
Insert Brick D, input B, C   //D=f_D(B, C)
```



Not actual syntax or network

# Fugu Overview - What Fugu *is*

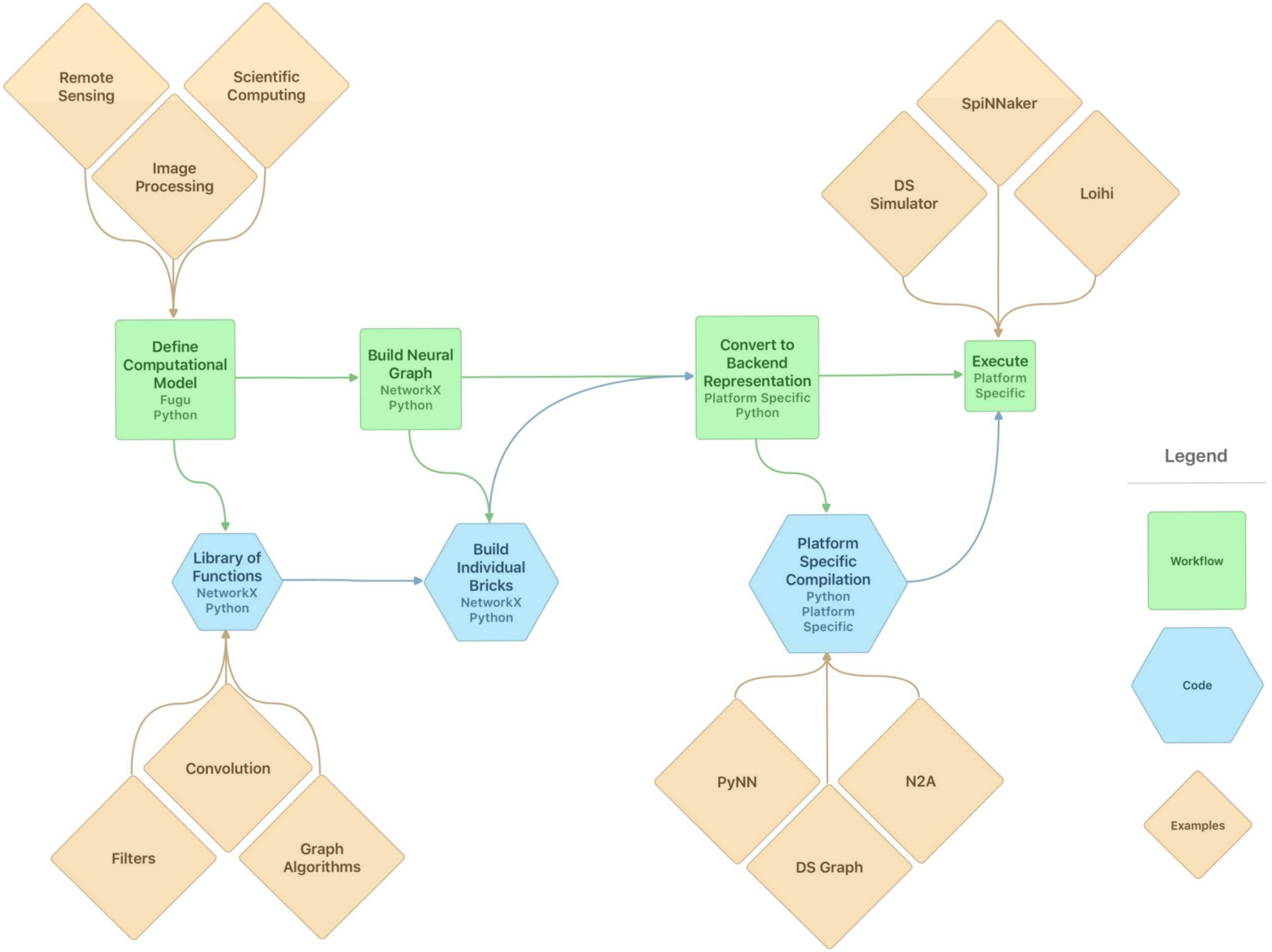
- Fugu is a linking framework
  - It's "easy" to build spiking circuits for a single computation
  - It's hard to do application-level computation on neuromorphic
  - We provide a mechanism to combine small computational kernels (Bricks) into large computational graphs
- Fugu is a spec
  - For the Bricks to transfer information, we need to agree on data formatting
  - For computation to be consistent, we need to agree on neuron behavior (lowest common denominator*)
  - For this to be useful, we need a hardware independent intermediate representation

*Usually, for most cases

# Fugu Overview – What Fugu *is not*

- Fugu includes but is NOT a simulator
  - Uses reference simulators ‘ds’ and ‘snn’ which can quickly run small-medium sized spiking networks
  - Simulators instantiates the fugu neuron model (discrete time, point synapses)
  - Fugu is designed to support a variety of backends including hardware platforms
- Fugu includes but is NOT a spiking algorithm
  - The goal of Fugu is to have a library of Fugu Bricks for many kernels
  - *We’re hoping that the community will help contribute*
- Fugu includes but is NOT a graph utility
  - NetworkX provides (nearly) all of our graph functionality
  - Node and edge properties are inherent in NetworkX and only become meaningful when interpreted by a backend

# Fugu Overview



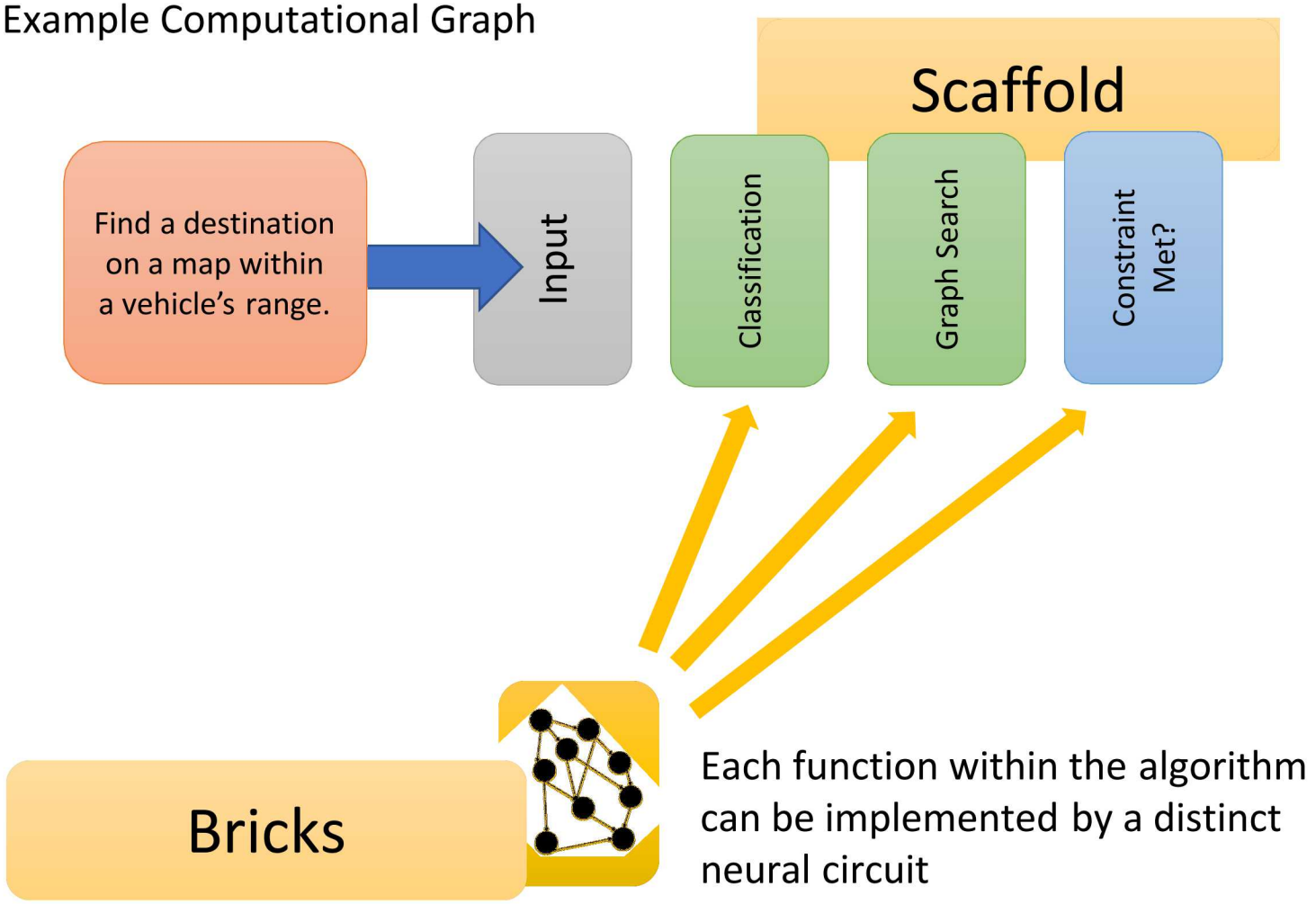
# Software Design and Organization

# Software Design

- Building a computational graph
- Fugu algorithms are designed using computational directed acyclic graphs
  - Nodes → Functions
  - Edges → Data flow

- Overall Fugu algorithm graph → *Scaffold*
- Neural circuits for functions → *Bricks*

Example Computational Graph

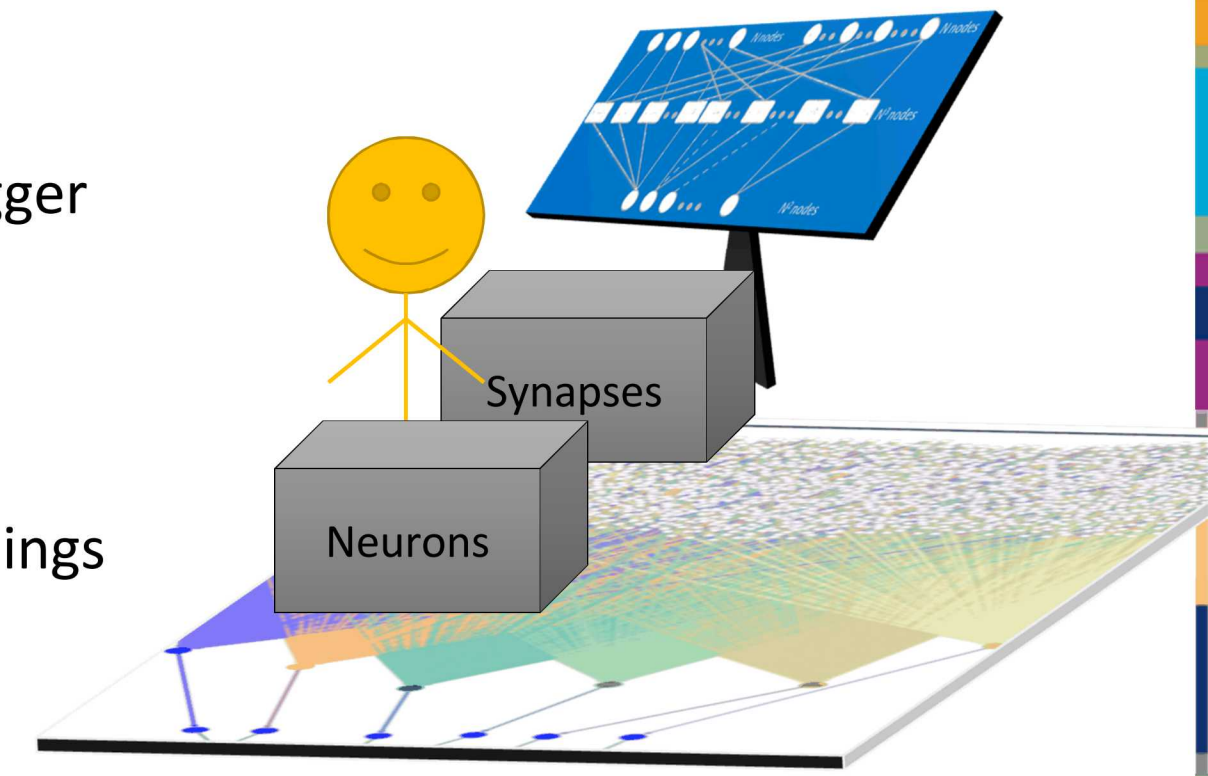


# Software Design

- Key Classes:
  - Scaffold
    - Provides the main entry point for people using Fugu
    - Manages the computational graph (`Scaffold.circuit`), metadata, backend, and network graph (`Scaffold.graph`)
    - `Fugu.Scaffold` in `Fugu.py`
  - Brick
    - Represents a fundamental spiking computational kernel
    - Spiking algorithms should inherit from Brick (or one of its subclasses); `Fugu.Brick` is an abstract class
    - Responsible for building a portion of the network graph
    - `Fugu.Brick` in `Fugu.py`

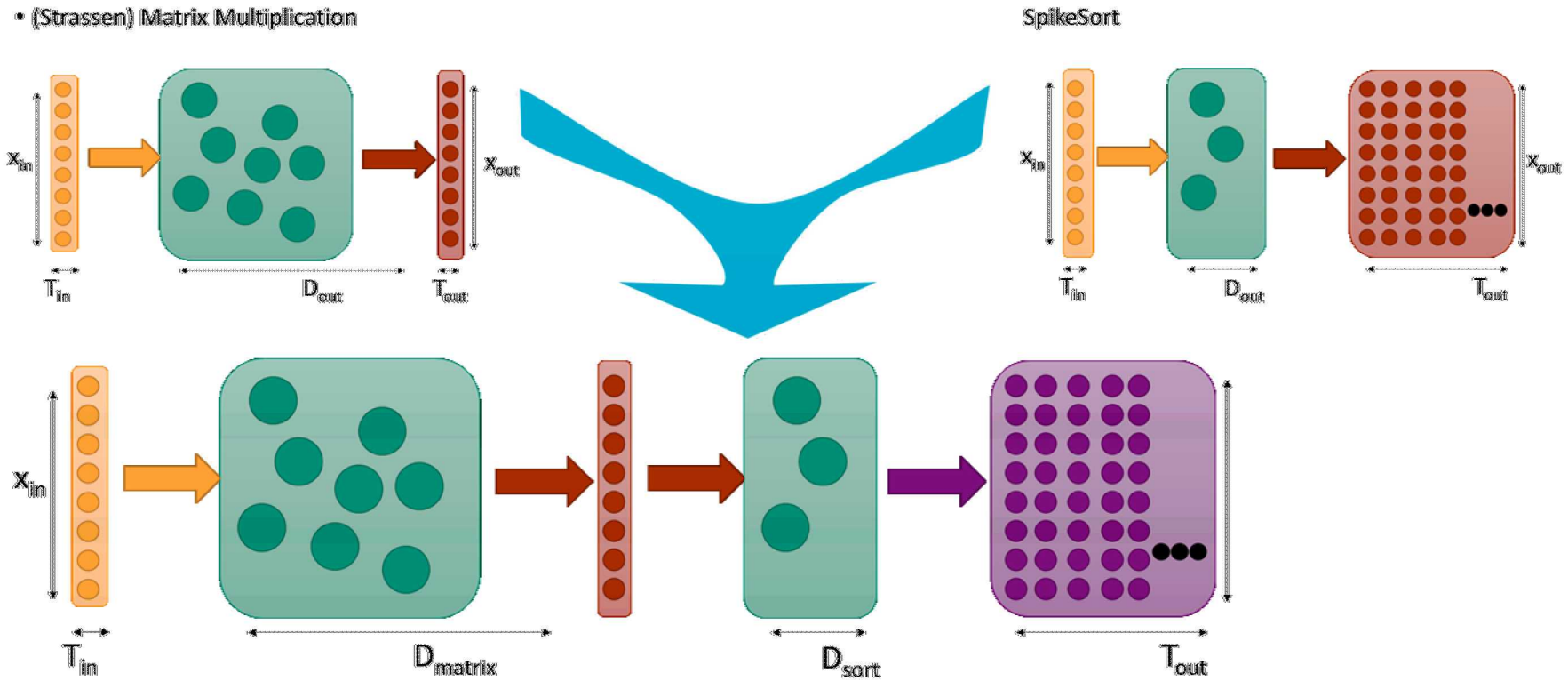
# Software Design – More about Bricks

- Fugu will contain a growing library of bricks
- Bricks can be linked together to compose bigger algorithms
- Bricks are individually responsible for
  - Building their portion of the graph
  - Adapting to a list of acceptable input codings (unary, binary, temporal, etc)
  - Scaling to the input dimensionality
  - Providing an output in a standard representation
  - Incorporating any specialized components (e.g., learning)



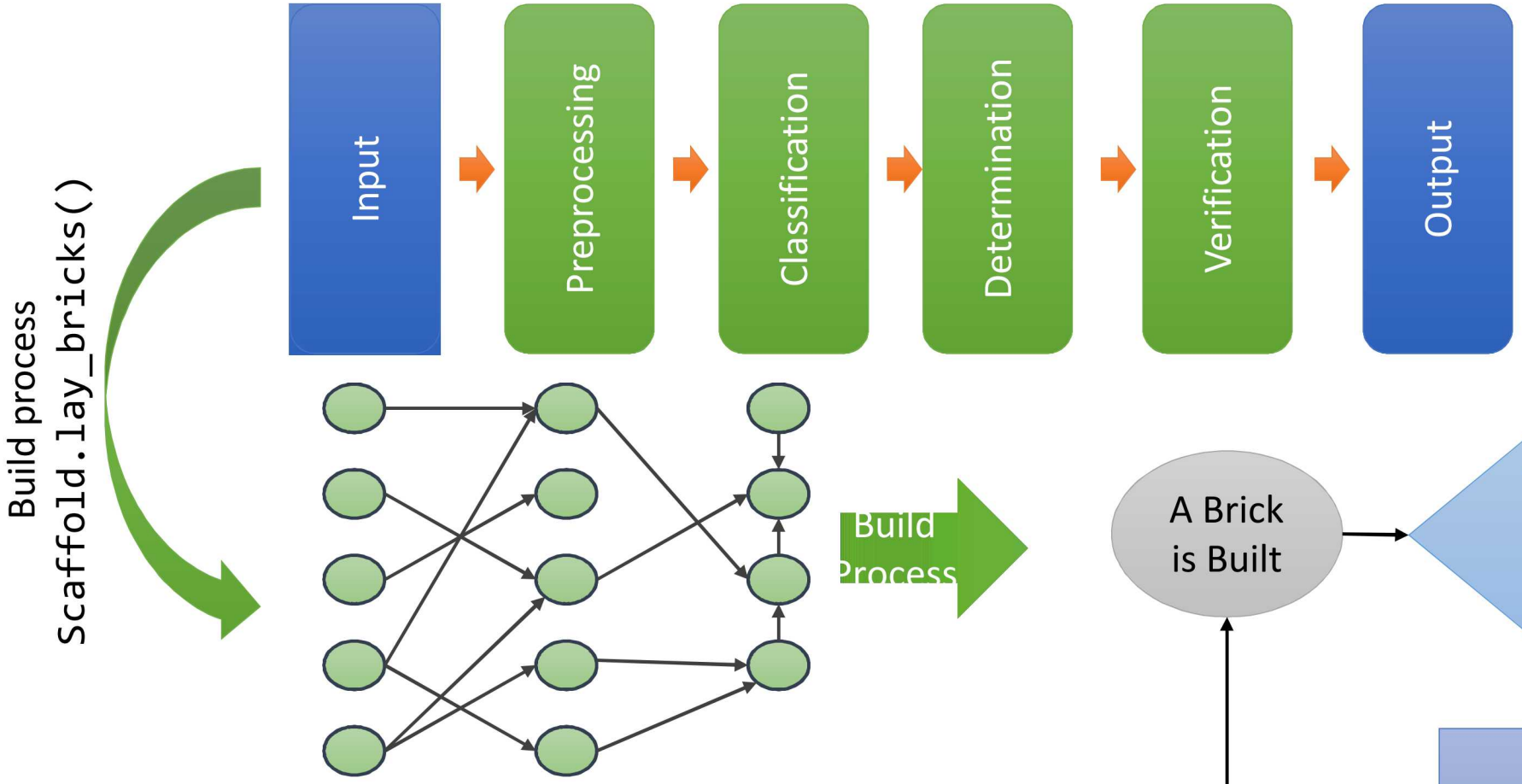
# Example of Linking

You can think of the scaffold as linking bricks' graphs together and those graphs adjust as needed.



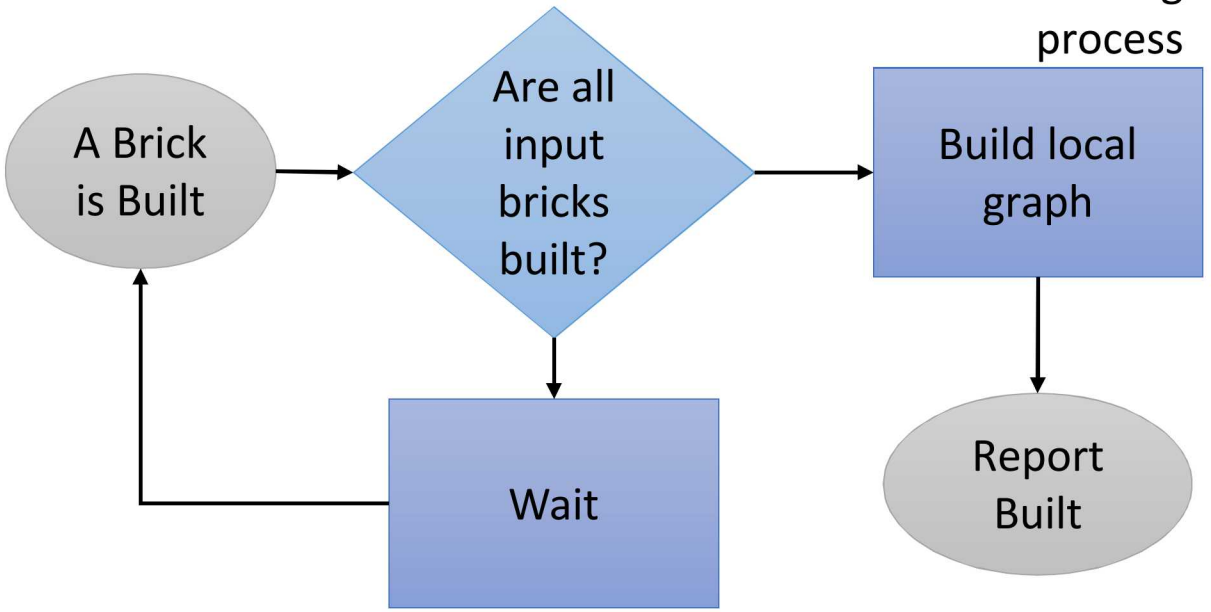
# Example of Linking

The scaffold holds references to each brick, 'lays' the bricks iteratively, and each brick builds its portion of the graph when all build conditions are satisfied.




`Scaffold.circuit` is a NetworkX Digraph of subclasses of `Fugu.Brick`

Control Flow for a Bricks' building process




These bricks have been built (i.e. their portions of `Scaffold.graph` are complete).

# Software Design

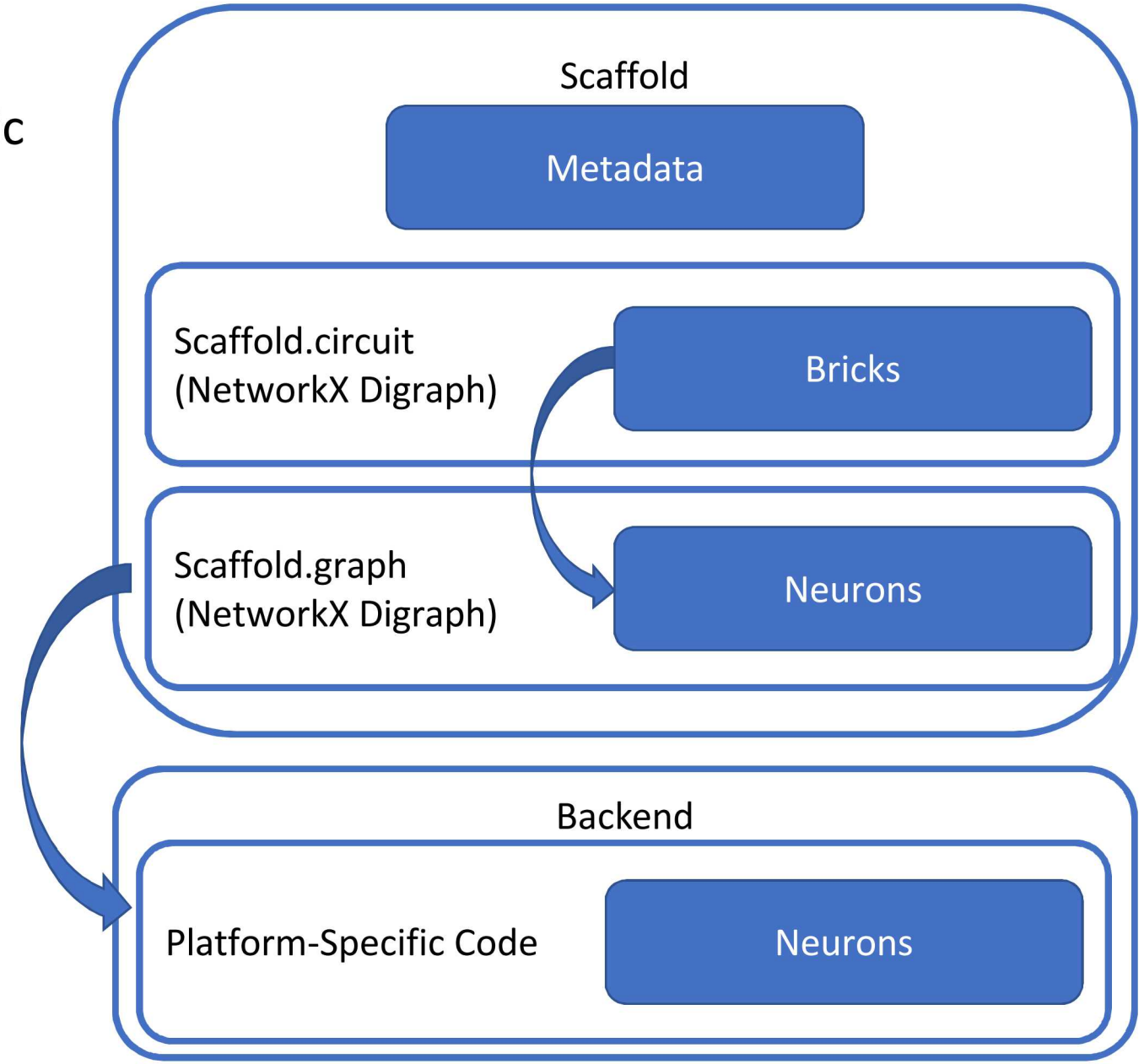
- The scaffold (mostly) handles coding issues for the User
- Scaffold may provide automatic casting between codings
- Extenders, however, do need to worry about codings; bricks are built to a list of acceptable codings

Coding Types

| Name | Description |
|---|---|
| unary-B | Unary coding, large values first |
| unary-L | Unary, small values first |
| binary-B | Binary, large values first |
| binary-L | Binary, small values first |
| temporal-B | Temporal, large values first |
| temporal-L | Temporal, small values first |
| Raster | Grid-like array |
| Population | # active represents value |
| Rate | Rate coded neurons |
| Undefined | Neurons without a coding |
| Current | Used for pre-threshold computation |

# Software Design – More about backends

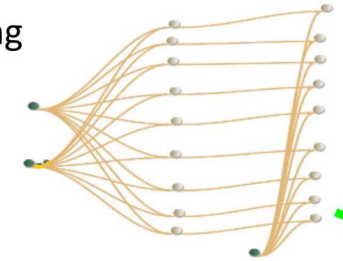
- A backend generates platform-specific code from the platform-independent network graph (`Scaffold.graph`)
- Included in Fugu today are basic reference simulators ‘ds’ and ‘snn’
- The backend handles inputs (represented by input bricks)
- Hardware platform backends can be developed by hardware partners (though we hope to provide a few as well)



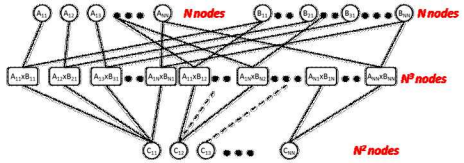

Dash
127.0.0.1:8050
Search
Sandia National Laboratories
scaffold.circuit
decode results
output neurons
Select...
view decoded results here
spike raster
neuron number
time
scaffold.graph
hover over a node to view its attributes
timestep
Sandia National Laboratories is a multimission laboratory managed and operated by National Technology & Engineering Solutions of Sandia, LLC, a wholly owned subsidiary of Honeywell International Inc., for the U.S. Department of Energyâ€™s National Security Administration under contract DE-NA0003525.

# Impacting Broad Areas of Computation

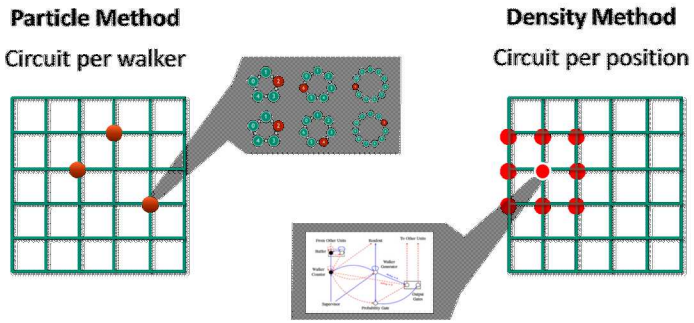
Pattern Matching



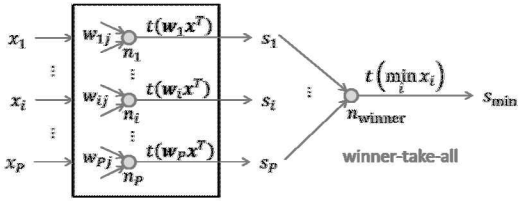
Linear Algebra

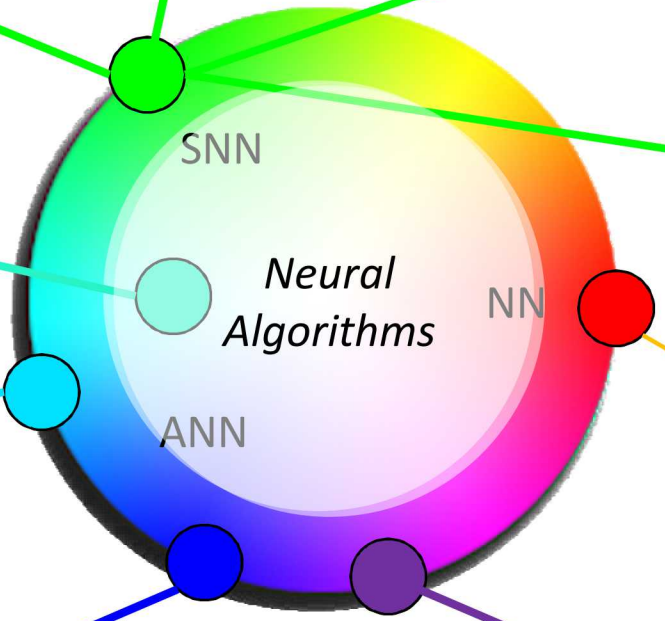


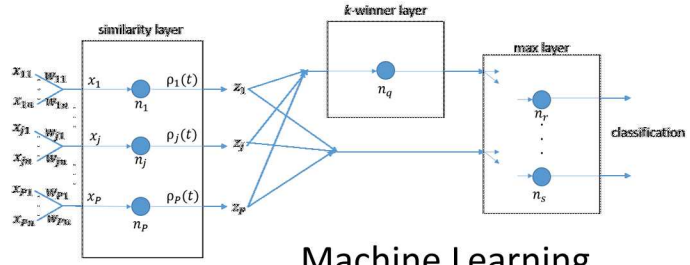

Scientific Computing



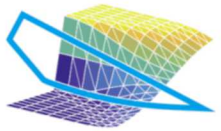

Optimizations



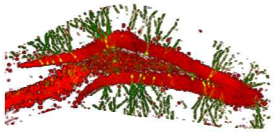



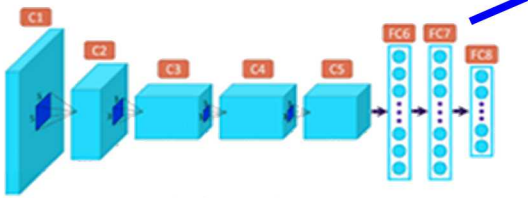

Machine Learning



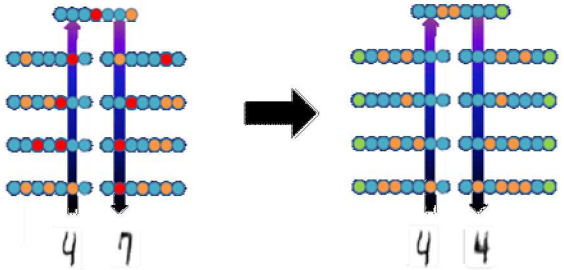
WHETSTONE



Intelligent Storage




Context Modulated Deep Learning



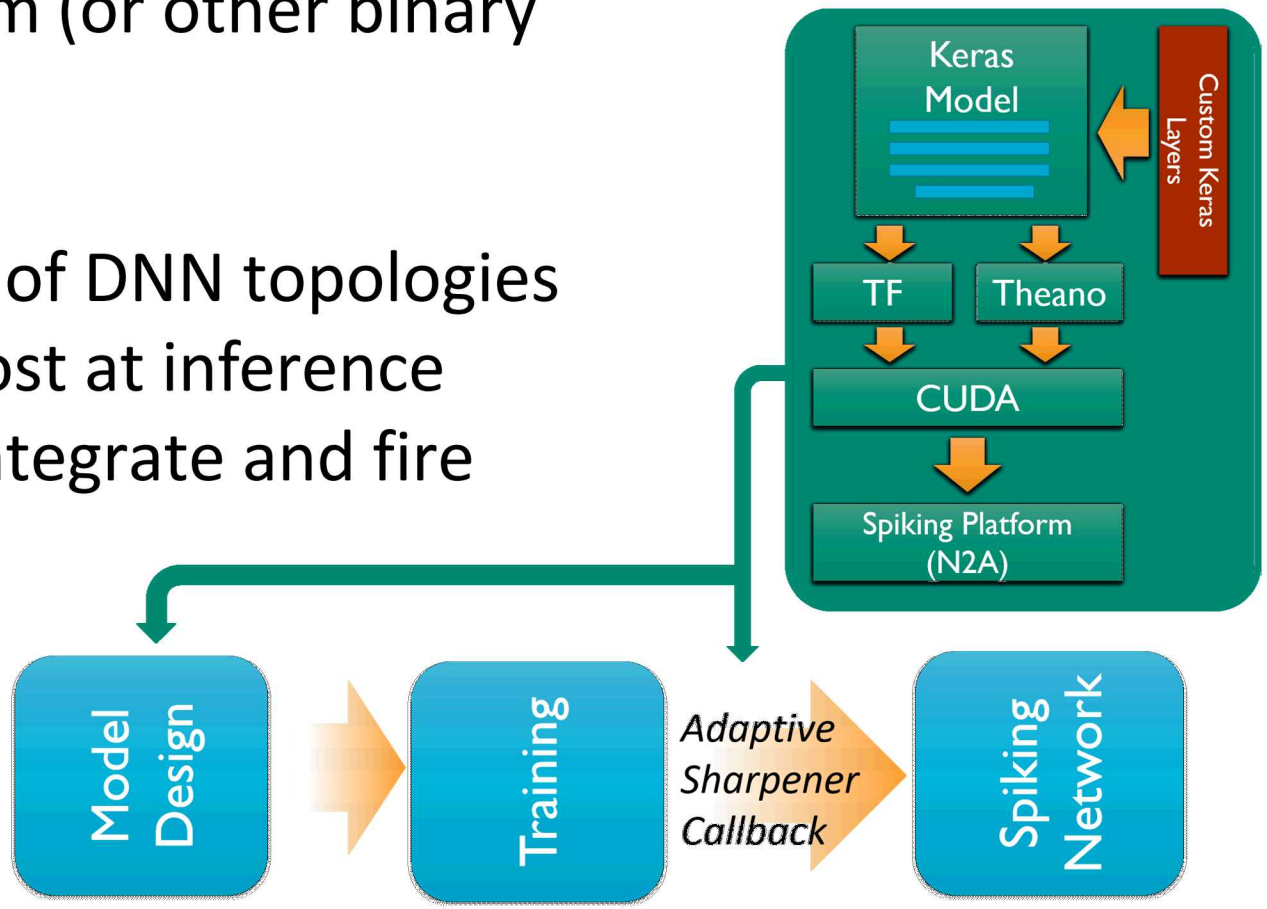
Adaptive Deep Learning

# Whetstone

# Whetstone Overview

Whetstone provides a drop-in mechanism for tailoring a DNN to a spiking hardware platform (or other binary threshold activation platforms)

- Hardware platform agnostic
- Compatible with a wide variety of DNN topologies
- No added time or complexity cost at inference
- Simple neuron requirements: Integrate and fire



# Whetstone Overview

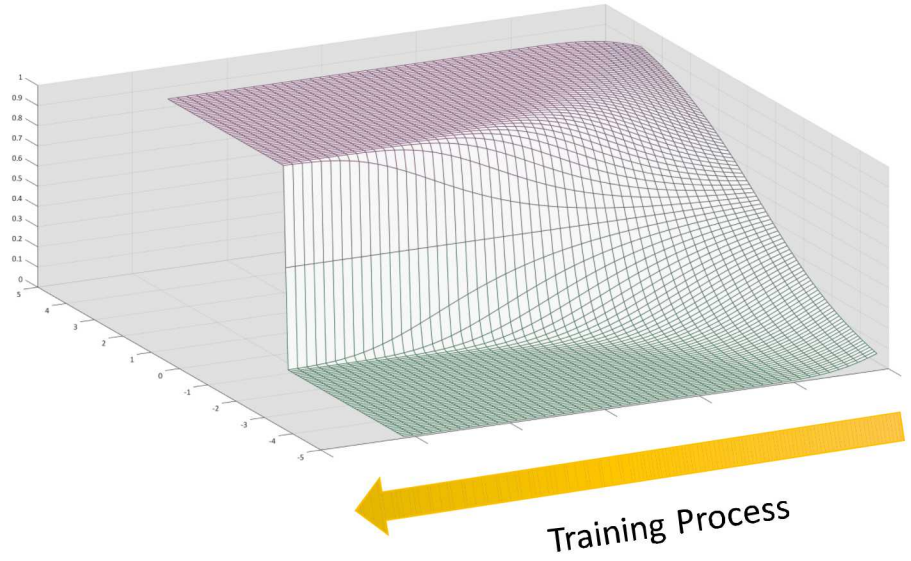
- Generally, gradient descent generates a sequence of weights $A_i$ with the goal of

  minimizing the error of $f(A_i x)$ in predicting the ground truth $y$.

- We generalize this by replacing the activation function $f$ with a sequence $f_k$ such that $f_k \rightarrow_{L_1} f$, where $f$ is now the threshold activation function.
- Now, the optimizer must

  minimize the error of $f_k(A_i x)$ in predicting y.

- Since the convergence in **neither $i$ nor $k$ is uniform**, this is a mathematically dangerous idea
- However, with a little care and a few tricks, the method reliably converges in many cases.



# Whetstone Overview

## When/Where do we decide to ‘sharpen’ the activations?

1) Bottom-up Sharpening (The ‘toothpaste tube’ method)
   - Begin sharpening at the bottom layer
   - Wait until previous layer is fully sharpened
   - Increases stability of convergence
2) Adaptive Sharpening Callback
   - Hand-tuning sharpening rates is hard
   - Instead, use loss as a guide for an *adaptive sharpener*
   - Adaptive sharpener implemented as a callback automatically adjusts sharpening based on loss thresholds

*Original Model Example*

```
⋮
model.add(Dense(256))
model.add(Activation(‘relu’))
model.add(Dense(10))
model.add(Activation(‘softmax’))
⋮
model.fit(x,y)
⋮
```

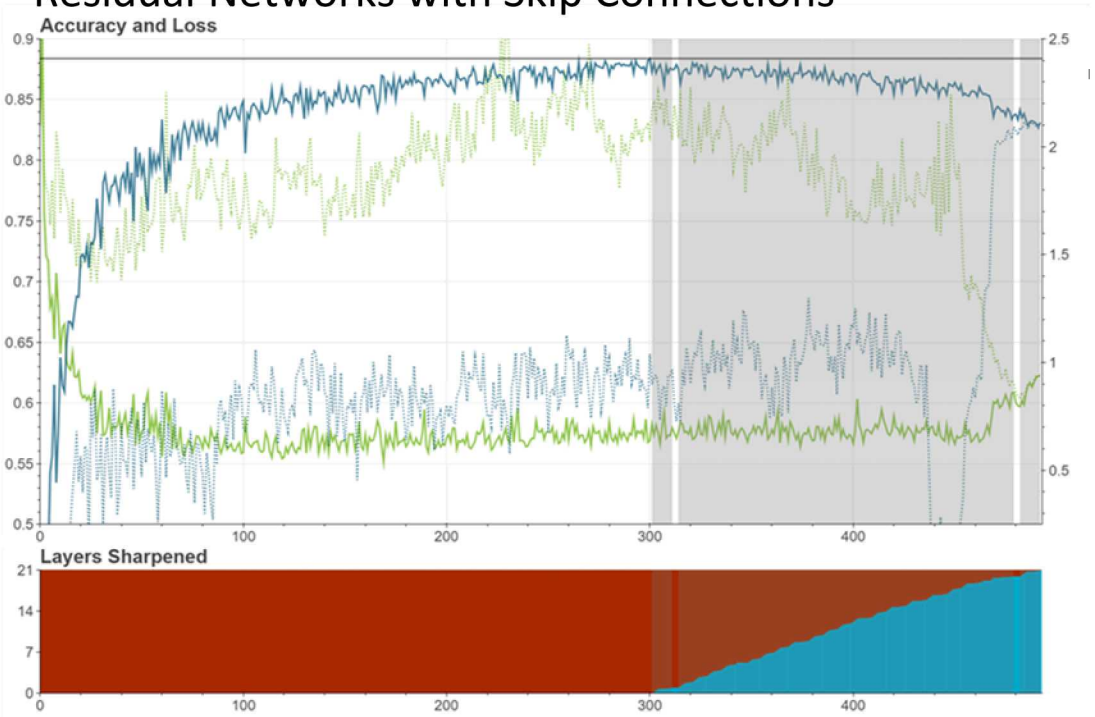
*Modified Model Example*

```
⋮
model.add(Dense(256))
model.add(Spiking_BRelu())
model.add(Dense(10))
model.add(Spiking_Brelu())
Model.add(Softmax_Decode(key))
⋮
sharpener = AdaptiveSharpener()
model.fit(x,y,callbacks=[sharpener])
⋮
```

# Effective Across Various Topologies, Datasets, and Tasks

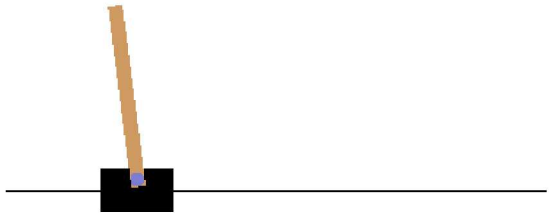
Semantic Segmentation (Trained on COCO Dataset; Videos from HMDB51 Dataset)

Residual Networks with Skip Connections



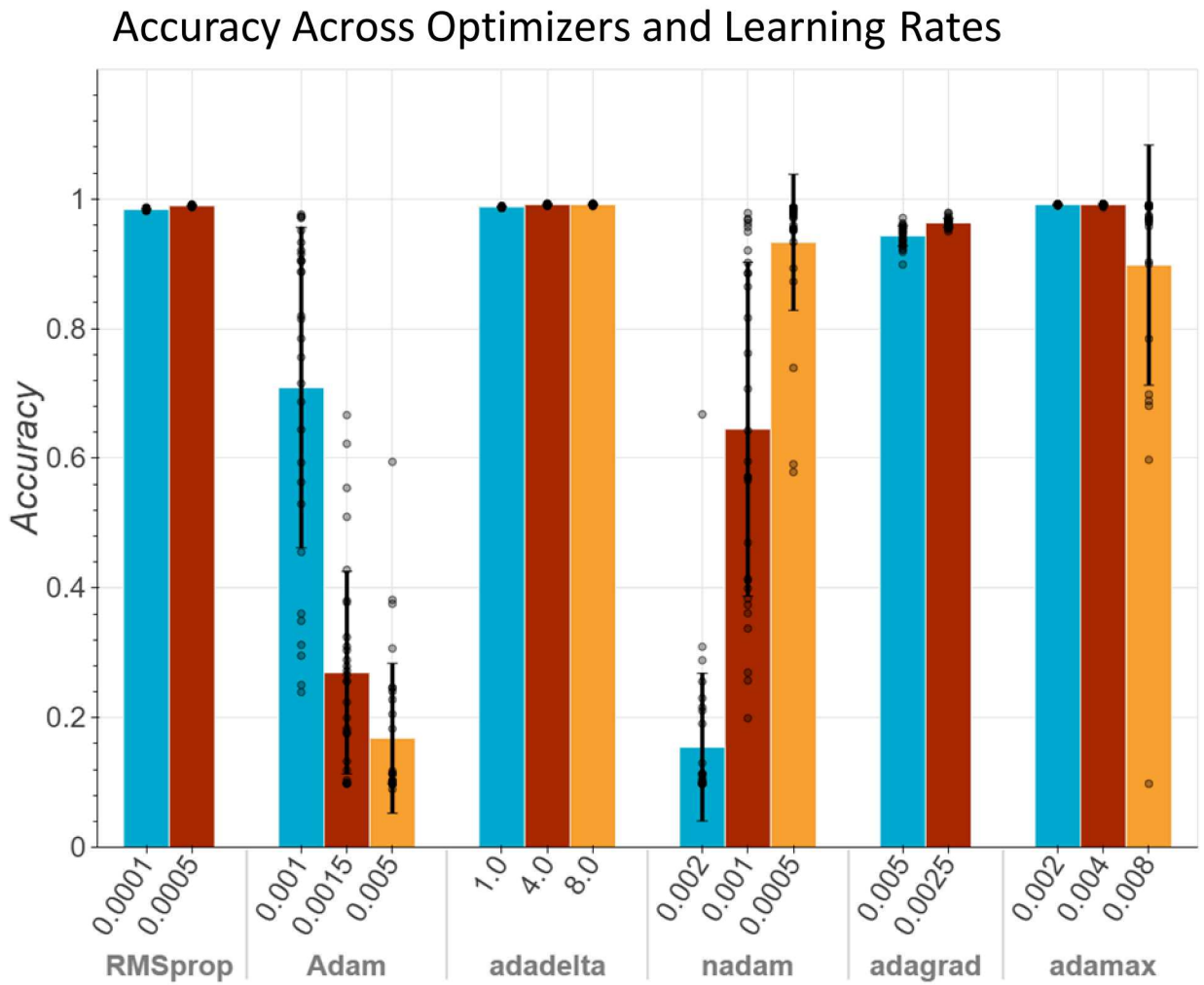

Autoencoders

DQN



# Established Deep Learning Techniques

- Sharpening process is sensitive to optimizer selection
- Adaptive optimizers often work better
- Learning rate modulation by moving average seems to help stability
- A custom Whetstone-aware optimizer is in early stages



# Scientific Computation on neuromorphic hardware

# PURPOSE, GOALS AND APPROACH

*Research Question*

Can neuromorphic platforms be used for efficient and valid numerical computation critical to Sandia's mission?

*Emerging **low-power** computing platforms can potentially dramatically change how we approach our high-performance computing mission*

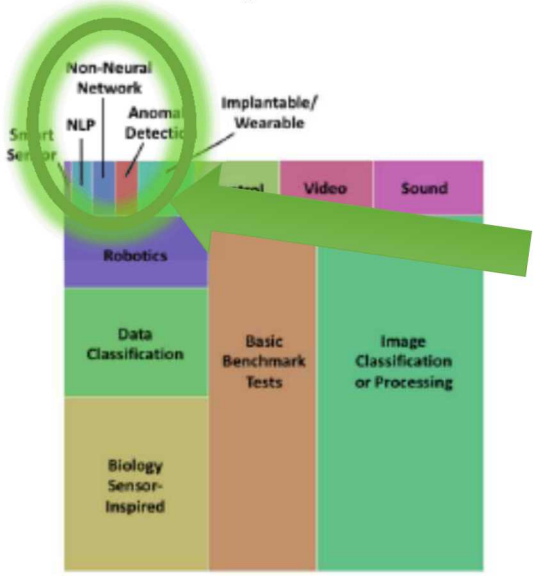




Fig. 13. Breakdown of applications to which neuromorphic systems have been applied. The size of the boxes corresponds to the number of works in which a neuromorphic system was developed for that application.

*Katie Schuman, ORNL 2017*

Only ~1% neuromorphic applications are not ANN-like

These aren't solving PDEs…

*How is this different from previous research?*

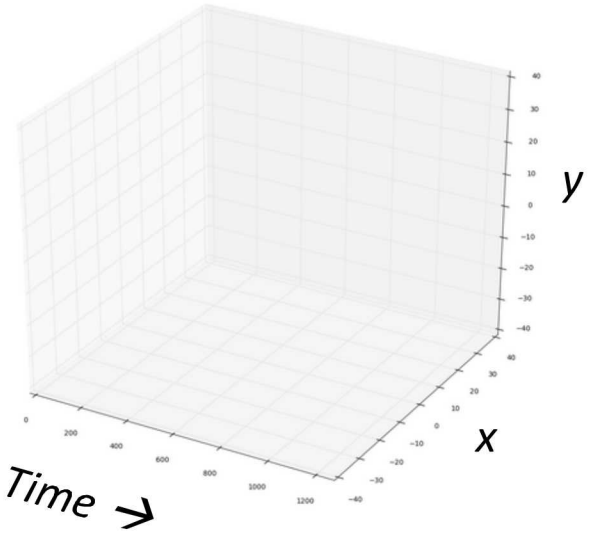
- ❑ Neuromorphic computing research has generally focused on AI applications
- ❑ Similarly, primary emphasis on application to scientific computing is on machine learning

# Spiking random walk algorithms – diffusion is a pure scientific computing task for spiking algorithms

- Diffusion can be modeled either as a deterministic PDE or a stochastic process
  - For an initial distribution of particles, $P_0$, what is distribution of particles at time *t?*
  - Diffusion can be modeled as the PDE

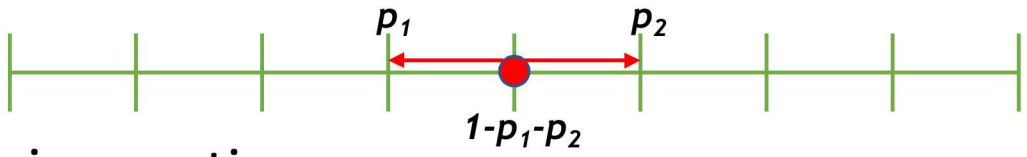
$$\frac{\partial C(x,t)}{\partial t} = D\frac{\partial^2 C(x,t)}{\partial x^2}$$

    with B.C. + I.C.
  - Stochastic process implements many random walkers to statistically approximate a solution
    - Mean position of *N* walkers approaches expected mean of deterministic solution at rate of 1/sqrt(N)



# One dimensional random walk case

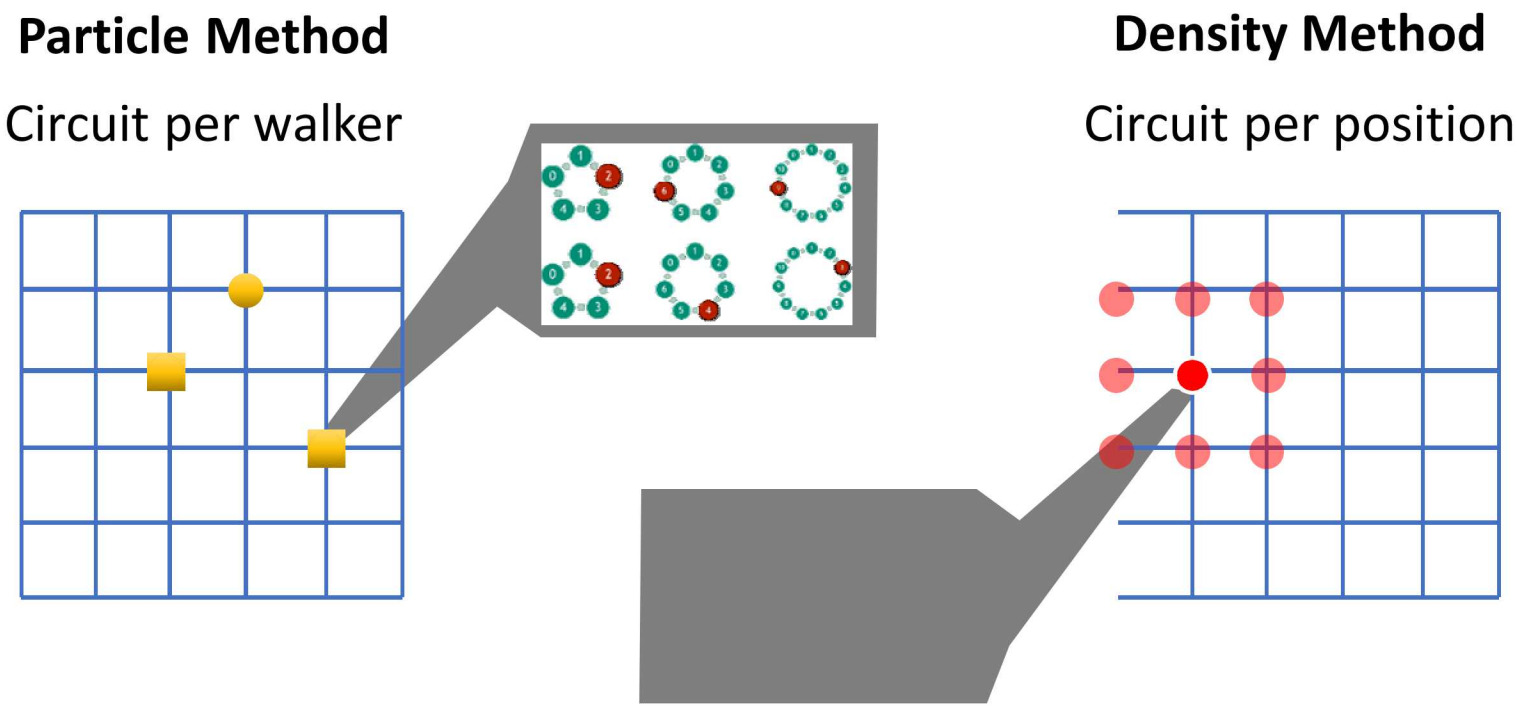
- Model:




  - Stochastic Brownian motion
  - Particle can either move or not (1-D case: probability $p_1$ right or $p_2$ left, with $1-p_1-p_2$ for no move)
  - Approximating PDE solutions requires sampling over MANY particles

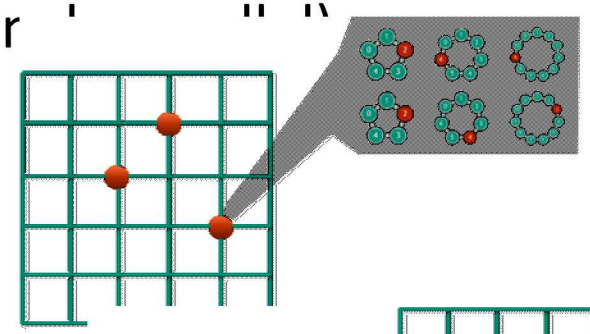
- Goal: Ensemble of neurons that represent stochastic particles, such that
  - Efficient to update (randomly add / subtract value)
  - Has sparse representation
  - Requires few neurons
  - Scalable across multiple dimensions / multiple particles

# Two spiking algorithms for random walk with different costs and benefits

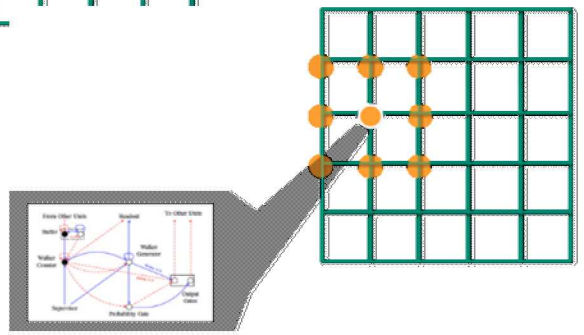


# Each method offers unique advantages

- Particle method
  - Path dependent behavior is readily available
  - Communication is entirely local within particles (embarrassir
  - With unlimited neurons, can run in constant time
  - **Ideal for sparse particles in large spaces**

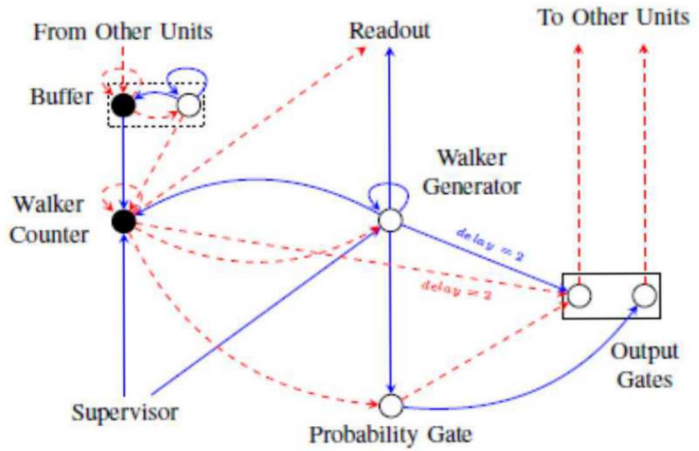


- Density method
  - Densities are readily available at all times
  - Non-local or other complex graphs can easily be implemented
  - With limited neurons, can tradeoff statistical approximation (i.e., number of walkers) with longer or shorter simulations
  - **Ideal for dense particles in small spaces**



# Density model repeats basic circuit at every vertex of mesh

- Each vertex encodes density of particles in the internal potential of certain nodes
- Each time step “hands off” particles to connected vertices according to probabilistic maps

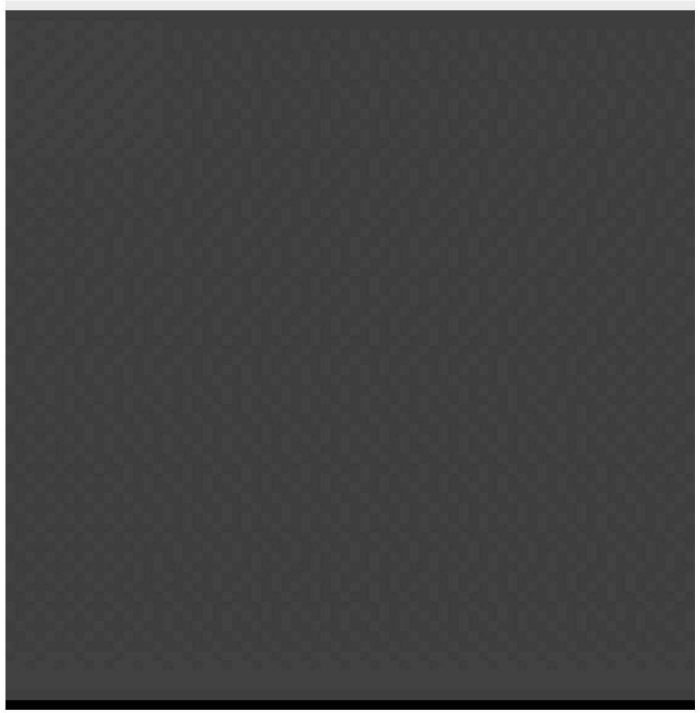



| Measure | Cost (for k locations, simulating N walkers; 1-D case) |
|---|---|
| Walker memory | O(1) |
| Connection memory | O(k) |
| Total neurons | O(k) |
| Time per physical timestep | O(max($\rho_i$)), where $\rho_i$ is the density of walkers at each location |
| Position energy per timestep | O(N) |
| Update energy per timestep | O(N) |

# *Density model* can model arbitrary graphs, enabling complex behaviors

- This example uses a mesh with one-way edges that permit walkers to move into colored parts of the image, but not out

- Scale of simulation
  - 1600 vertices
  - 8000 walkers
  - 1000 timesteps
  - 19205 neurons
  - 60802 synapses



# Applications of Random Walks to solve PDE

- Numerous applications:
  - Diffusion equations
  - Radiation transport
  - Capacitance
- Not a traditional machine learning application
- Highly scalable at low power budgets

# Concluding thoughts

- Fugu provides a framework for rapid development of general neural algorithms
- Fugu enables backend agnostic development
  - Makes benchmarking easy
- Whetstone can help with building bricks
- Neural algorithms can target a wide array of applications
  - Enables neuromorphic hardware to be used for in several domains