# NerveFlow

A Tensorflow Python Framework for Modelling Biological Neural Networks

https://github.com/technosap/nerveFlow

# TensorFlow: Google's LinAlg Package

"TensorFlow™ is an open source software library for high performance numerical computation. Its flexible architecture allows easy deployment of computation across a variety of platforms (CPUs, GPUs, TPUs), and from desktops to clusters of servers"

Essentially, it is a python package that (much like BLAS on Intel MKL) speeds up Linear Algebra Computation. What is special about this system is that it is capable of utilizing GPUs and TPUs for computation and its written in a simpler language like python.

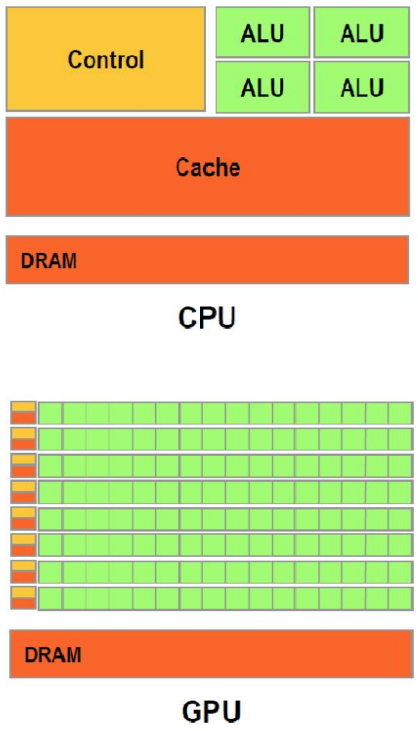
# Why GPU/TPU vs CPU?

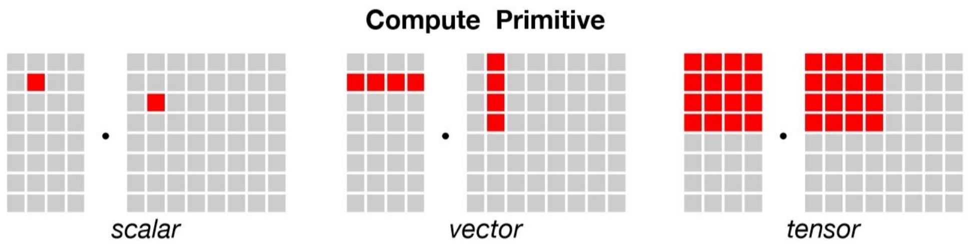
The answer lies in the architecture:

**CPU = Faster per Core Processing, Slow but Large Memory Buffer, Few Cores**

**GPU/TPU = Slower Processing, Faster but Smaller Memory Buffer, Many Cores**

Thus GPUs and TPUs are optimized for large number of simple calculations done parallely. The extent of this parallelization makes it suitable for vector/tensor manipulation.



# Problem Statement: Solving Simultaneous DEs

The simulation of a network of n neurons is essentially the solving of $F(n)$ simultaneous DEs where F is a function of the network architecture and mathematical model of neurons and synapses.

Generally, integration using RK4 is a *step-wise iterative process* that can evade the benefits of parallel processing but since we are performing similar operations on $F(n)$ different state variables, there is some room for parallel processing for the RK4 steps.

But the biggest benefit we have is that neurons have somewhat similar differential equations with different parameters and we can exploit this.

# NerveFlow Integrator

```python
class _Integrator():

    def __init__(self,n_,F_b):
        self.n_ = n_
        self.F_b = F_b

    def integrate(self, evol_func, y0, time_grid): # iterator
        time_delta_grid = time_grid[1:] - time_grid[:-1]
        scan_func = self._make_scan_func(evol_func)
        y_grid = functional_ops.scan(scan_func, (time_grid[:-1], time_delta_grid),y0)
        return array_ops.concat([[y0], y_grid], axis=0)

    def _make_scan_func(self, evol_func): # stepper function

        def scan_func(y, t_dt):

            n_ = self.n_
            F_b = self.F_b

            t,dt = t_dt

            if n_>0:


                dy = self._step_func(evol_func, t, dt, y)
                dy = math_ops.cast(dy, dtype=y.dtype)
                out = y + dy

                ## Operate on non-integral

                ft = y[-n_:]

                l = tf.zeros(tf.shape(ft),dtype=ft.dtype)
                l_ = t-ft

                z = tf.less(y[:n_],F_b)
                z_ = tf.greater_equal(out[:n_],F_b)

                df = tf.where(tf.logical_and(z,z_),l_,l)

                ft_ = ft+df

                return tf.concat([out[:-n_],ft_],0)

            else:
                dy = self._step_func(evol_func, t, dt, y)
                dy = math_ops.cast(dy, dtype=y.dtype)
                return y + dy

        return scan_func

    def _step_func(self, evol_func, t, dt, y):
        k1 = evol_func(y, t)
        half_step = t + dt / 2
        dt_cast = math_ops.cast(dt, y.dtype)

        k2 = evol_func(y + dt_cast * k1 / 2, half_step)
        k3 = evol_func(y + dt_cast * k2 / 2, half_step)
        k4 = evol_func(y + dt_cast * k3, t + dt)
        return math_ops.add_n([k1, 2 * k2, 2 * k3, k4]) * (dt_cast / 6)
```

There are a few major differences between a standard RK4 integrator and the NerveFlow integrator based on Tensorflow:

- ❑ Tensorflow Optimized ‘scan’ Iterator
- ❑ DE-less Updation with 1-bit memory to update firing time
- ❑ Tensorflow optimized RK4 Stepper
- ❑ Uses only Tensorflow Functions

# The Key to Speedup : Vectorizing

Tensorflow is only as intelligent as the coder. The key to maximizing the performance is the parallelization of functional evaluations using vectorization.

Say $\vec{X} = [x_1, x_2 \ldots x_m]$ is the state vector of a single neuron and its dynamics are defined by equations of the form:

$$\frac{d\vec{X}}{dt} = [f_1(x_1, x_2 \ldots), f_2(x_1, x_2 \ldots) \ldots f_m(x_1, x_2 \ldots)]$$

We have to somehow convert these to a form in which all evaluations are done as vector calculations and NOT scalar calculations

# The Key to Speedup : Vectorizing

So what we need for a network of neurons is to have a method to evaluate the updation of $\mathbf{X} = [\overrightarrow{X_1}, \overrightarrow{X_2} \ldots \overrightarrow{X_n}]$. Now there is a simple trick that allows us to maximize the parallel processing. Each neuron represented by $\overrightarrow{X_i}$ has a distinct set of parameters $\overrightarrow{p_i} = [p_{i1}, p_{i2} \ldots p_{im}]$ and differential equations:

$$\frac{d\overrightarrow{X_i}}{dt} = \vec{f}\,(p_{i1}, p_{i2} \ldots x_{i1}, x_{i2} \ldots)$$

Now, despite the parameters being different, the functional forms of the updation is similar for the same state variable for different neurons. Thus, the trick is to reorganize $\mathbf{X} = [\overrightarrow{X_1}, \overrightarrow{X_2} \ldots \overrightarrow{X_n}] = [(x_{11} \ldots x_{1m}), (x_{21} \ldots x_{2m}) \ldots (x_{n1} \ldots x_{nm})]$ as

$$\mathbf{X}' = [(x_{11}, x_{21} \ldots x_{n1}), (x_{12}, x_{22} \ldots x_{n2}) \ldots (x_{1m}, x_{2m} \ldots x_{nm})] = [\overrightarrow{x_1}, \overrightarrow{x_2} \ldots \overrightarrow{x_m}]$$

# The Key to Speedup : Vectorizing

Now that we know the trick, what is the benefit? Earlier, each state variable (say variable i of neuron j) had a DE of the form $\frac{dx_{ji}}{dt} = f_i(p_{jk} \ldots, x_{jl} \ldots)$

Now we can parallelize this as a simple vector computation of the form

$$\frac{d\overrightarrow{x_i}}{dt} = f_i(\overrightarrow{P_k} \ldots, \overrightarrow{x_l} \ldots)$$

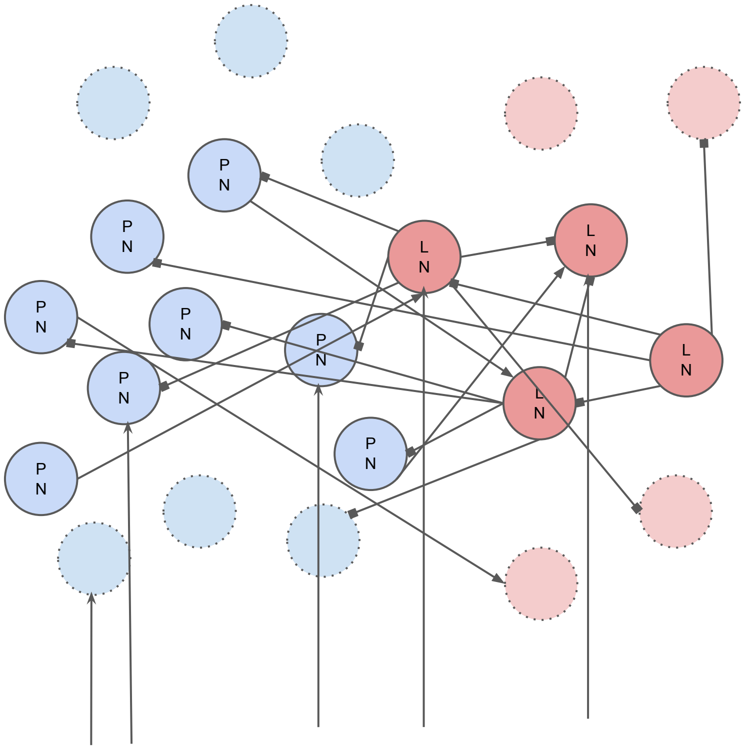
where $\mathbf{P} = [\overrightarrow{P_1}, \overrightarrow{P_2} \ldots \overrightarrow{P_m}] = [(p_{11}, p_{21} \ldots p_{n1}), (p_{12}, p_{22} \ldots p_{n2}) \ldots (p_{1m}, p_{2m} \ldots p_{nm})]$

Thus we can simply write:

$$\frac{d\mathbf{X}'}{dt} = [\frac{d\overrightarrow{x_1}}{dt}, \frac{d\overrightarrow{x_2}}{dt} \ldots \frac{d\overrightarrow{x_m}}{dt}]$$

# Implementation: The Model

# Locust AL Model (Bazhenov et al. 2001)




2 Types of Neurons:

- Excitatory Projection Neurons (PN)
- Inhibitory Local Interneuron (LN)

PN:LN = 3:1  Total 120 neurons

Connection Probability:

- PN->PN = 0%
- LN->LN = 50%
- PN->LN = 50%
- LN->PN = 50%

33% Randomly Receive Input

# Locust AL Model (Bazhenov et al. 2001)

## Excitatory Projection Neurons (PN)

$$C_m \frac{dV_{PN}}{dt} = I_{stim} - I_L - I_{Na} - I_K - I_A - I_{KL} - I_{GABA_a} - I_{Ach}$$

$$I_L = g_L(V - E_L) \quad g_L = 0.15\ \mu S \quad E_L = -55\ mV$$

$$I_{Na} = g_{Na} m_{Na}^3 h_{Na}(V - E_{Na}) \quad g_{Na} = 100\ \mu S \quad E_{Na} = 50\ mV$$

$$I_K = g_K n_K^4 (V - E_K) \quad g_K = 10\ \mu S \quad E_K = -95\ mV$$

$$I_A = g_A m_A^4 h_A (V - E_A) \quad g_A = 10\ \mu S \quad E_A = -95\ mV$$

$$I_{KL} = g_{KL}(V - E_{KL}) \quad g_{KL} = 0.05\ \mu S \quad E_{KL} = -95\ mV$$

$$\begin{aligned}
&K\ Channel:\\
&T = 22^\circ C\\
&\phi = 3.0^{(T-36.0)/10}\\
&V' = V - (-50)\\
&\alpha_n = 0.02 \frac{15.0 - V'}{e^{(15.0-V')/5.0} - 1.0}\\
&\beta_n = 0.5 e^{(10.0-V')/40.0}\\
&t_n = 1.0/((\alpha_n + \beta_n)\phi)\\
&n_\infty = \alpha_n/(\alpha_n + \beta_n)\\
&\frac{dn_k}{dt} = -\frac{1}{tn}(n_K - n_\infty)
\end{aligned}$$

$$\begin{aligned}
&Na\ Channel:\\
&T = 22^\circ C\\
&\phi = 3.0^{(T-36.0)/10}\\
&V' = V - (-50)\\
&\alpha_m = \frac{0.32(13.0 - V')}{e^{(13.0-V')/4.0} - 1.0}\\
&\beta_m = \frac{0.28(V' - 40.0)}{e^{(V_-40.0)/5.0} - 1.0}\\
&\alpha_h = 0.128 e^{(17.0-V')/18.0}\\
&\beta_h = \frac{4.0}{e^{(40.0-V')/5.0} + 1.0}\\
&t_m = 1.0/((\alpha_m + \beta_m)\phi)\\
&m_\infty = \alpha_m/(\alpha_m + \beta_m)\\
&t_h = 1.0/((\alpha_h + \beta_h)\phi)\\
&h_\infty = \alpha_h/(\alpha_h + \beta_h)\\
&\frac{dm}{dt} = -\frac{1}{tm}(m - m_\infty)\\
&\frac{dh}{dt} = -\frac{1}{th}(h - h_\infty)
\end{aligned}$$

$$\begin{aligned}
&Transient\ K\ (A)\ Channel:\\
&T = 36^\circ C\\
&\phi = 3.0^{(T-23.5)/10}\\
&m_\infty = \frac{1.0}{1.0 + e^{-(V+60.0)/8.5}}\\
&h_\infty = \frac{1.0}{1.0 + e^{(V+78)/6.0}}\\
&t_m = \frac{1.0}{e^{(V+35.82)/19.69} + e^{-(V+79.69)/12.7} + 0.37}\\
&t_h = \frac{1.0}{(e^{(V+46.05)/5.0} + e^{-(V+238.4)/37.45})\phi}, when\ V < -63\ mV\\
&\quad = 19.0/\phi, otherwise\\
&\frac{dm}{dt} = -\frac{1}{tm}(m - m_\infty)\\
&\frac{dh}{dt} = -\frac{1}{th}(h - h_\infty)
\end{aligned}$$

# Locust AL Model (Bazhenov et al. 2001)

## Inhibitory Local Interneurons (LN)

$$C_m \frac{dV_{LN}}{dt} = I_{stim} - I_L - I_{Ca} - I_K - I_{K(Ca)} - I_{KL} - I_{GABA_a} - I_{Ach}$$

$I_L = g_L(V - E_L)$
$g_L = 0.15\ \mu S$
$E_L = -50\ mV$

$I_{Ca} = g_{Ca} m_{Ca}^2 h_{Ca}(V - E_{Ca})$
$g_{Ca} = 3\ \mu S$
$E_{Ca} = 140\ mV$

$I_K = g_K n_K^4 (V - E_K)$
$g_K = 10\ \mu S$
$E_K = -95\ mV$

$I_{K(Ca)} = g_{K(Ca)} m_{K(Ca)} (V - E_A)\phi$
$g_{K(Ca)} = 0.3\ \mu S$
$E_{K(Ca)} = -90\ mV$
$\phi = 2.3^{(T-23.0)/10} \quad T = 26^\circ C$

$I_{KL} = g_{KL}(V - E_{KL})$
$g_{KL} = 0.02\ \mu S$
$E_{KL} = -95\ mV$

$K\ Channel:$
$T = 22^\circ C$
$\phi = 3.0^{(T-36.0)/10}$
$V' = V - (-50)$
$\alpha_n = 0.02 \frac{15.0 - V'}{e^{(15.0-V')/5.0} - 1.0}$
$\beta_n = 0.5 e^{(10.0-V')/40.0}$
$t_n = 1.0/((\alpha_n + \beta_n)\phi)$
$n_\infty = \alpha_n/(\alpha_n + \beta_n)$
$\frac{dn_k}{dt} = -\frac{1}{tn}(n_K - n_\infty)$

$Ca\ dependent\ K\ Channel:$
$T = 26^\circ C$
$\phi = 2.3^{(T-23.0)/10}$
$\alpha_m = 0.01[Ca^{2+}]$
$\beta_m = 0.02$
$t_m = 1.0/((\alpha_m + \beta_m)\phi)$
$m_\infty = \alpha_m/(\alpha_m + \beta_m)$
$\frac{dm_{K(Ca)}}{dt} = -\frac{1}{tm}(m_{K(Ca)} - m_\infty)$

$Ca\ dynamics:$
$\frac{d[Ca^{2+}]}{dt} = -A_{Ca} I_{Ca} - \frac{Ca - Ca_\infty}{\tau_{Ca}}$
$A_{Ca} = 2 \times 10^{-4}\ \frac{mM \cdot cm^2}{ms \cdot \mu A}$
$Ca_\infty = 2.4 \times 10^{-4}\ mM$
$\tau_{Ca} = 150\ ms$

$Ca\ Channel:$
$m_\infty = \frac{1.0}{1.0 + e^{-(V+20.0)/6.5}}$
$h_\infty = \frac{1.0}{1.0 + e^{(V+25.0)/12}}$
$t_m = 1.5$
$t_h = 0.3 e^{(V-40.0)/13.0} + 0.002 e^{(60.0-V)/29}$
$\frac{dm}{dt} = -\frac{1}{tm}(m - m_\infty)$
$\frac{dh}{dt} = -\frac{1}{th}(h - h_\infty)$

# Locust AL Model (Bazhenov et al. 2001)

Synapses

$$I_{syn} = g_{syn}[O](V - E_{syn})$$

$$\frac{d[O]}{dt} = \alpha(1 - [O])[T] - \beta[O]$$

**Cholinergic Synapse (Excitatory)**

$$E_{ach} = 0\ mV$$

$$[T] = A\theta(t - (t_{max} + t_0 + t_{delay}))\theta(t - t_0 + t_{delay}))$$

where

$A = 0.5$

$\theta(x) = Heavyside\ step\ function$

$t_0 = Receptor\ Activation\ Time$

$t_{delay} = Axonal\ Delay = 0\ ms$

$t_{max} = Maximal\ Activation\ Time = 0.3\ ms$

$\alpha = 10\ ms^{-1} \quad \beta = 0.2\ ms^{-1}$

$g_{Ach} = 0.35\mu S\ between\ PNs$

$g_{Ach} = 0.3\mu S\ between\ PNs\ and\ LNs$

**Fast GABAa Synapse (Inhibitory)**

$$E_{GABA_a} = -70\ mV$$

$$[T] = \frac{1.0}{1.0 + e^{-(V_{pre} - V_0)/\sigma}}$$

where

$V_0 = -20\ mV$

$\sigma = 1.5$

$\alpha = 10\ ms^{-1}$

$\beta = 0.16\ ms^{-1}$

$g_{GABA_a} = 0.8\mu S\ between\ LNs$

$g_{GABA_a} = 0.8\mu S\ between\ LNs\ and\ PNs$

# Actual Implementation on Python 3.6.6

# NerveFlow: Import and Simulation Parameters

```python
import tensorflow as tf
import numpy as np
import nerveflow as nv

import time
import gc
```

Essential Imports: Numpy, Tensorflow and NerveFlow

Nerveflow module contains the Integrator

```python
n_n = 120                         # number of neurons

p_n = 90                          # number of PNs
l_n = 30                          # number of LNs

t = np.arange(0.0, 300, 0.01)     # duration of simulation
```

Initialize Simulation Parameters

# NerveFlow: Define Channel Parameters

```
### NEURON PARAMETERS ###

C_m  = [1.0]*n_n                    # Capacitance

# Common Current Parameters #

g_K  = [10.0]*n_n                   # K conductance
g_L  = [0.15]*n_n                   # Leak conductance
g_KL  = [0.05]*p_n + [0.02]*l_n     # K leak conductance

E_K  = [-95.0]*n_n                  # K Potential
E_L  = [-55.0]*p_n + [-50.0]*l_n    # Leak Potential
E_KL  = [-95.0]*n_n                 # K Leak Potential

# Type Specific Current Parameters #

## PNs

g_Na = [100.0]*p_n                  # Na conductance
g_A  = [10.0]*p_n                   # Transient K conductance

E_Na = [50.0]*p_n                   # Na Potential
E_A  = [-95.0]*p_n                  # Transient K Potential

## LNs

g_Ca = [3.0]*l_n                    # Ca conductance
g_KCa = [0.3]*l_n                   # Ca dependent K conductance

E_Ca = [140.0]*l_n                  # Ca Potential
E_KCa = [-90]*l_n                   # Ca dependent K Potential

A_Ca = 2*(10**(-4))                 # Ca outflow rate
Ca0 = 2.4*(10**(-4))                # Equilibrium Calcium Concentration
t_Ca = 150                          # Ca recovery time constant
```

The next step is to define the different parameter vectors $P_i$ and initialize their values as lists. Try to segregate the neurons with different properties into successive sets to make referencing easier. Here we are following the convention [60 PN, 30 LN].

# NerveFlow: Define Synapse Parameters

```
# Synaptic Current Parameters #

## Acetylcholine

ach_mat = np.zeros((n_n,n_n))        # Ach Synapse Connectivity Matrix

ach_mat[p_n:,:p_n] = np.random.choice([0.,1.],size=(l_n,p_n)) # 50% probability of PN -> LN

np.fill_diagonal(ach_mat,0.)         # No self connection


n_syn_ach = int(np.sum(ach_mat))     # Number of Acetylcholine (Ach) Synapses
alp_ach = [10.0]*n_syn_ach           # Alpha for Ach Synapse
bet_ach = [0.2]*n_syn_ach            # Beta for Ach Synapse
t_max = 0.3                          # Maximum Time for Synapse
t_delay = 0                          # Axonal Transmission Delay
A = [0.5]*n_n                        # Synaptic Response Strength
g_ach = [0.35]*p_n+[0.3]*l_n         # Ach Conductance
E_ach = [0.0]*n_n                    # Ach Potential

## GABAa (fast GABA)

fgaba_mat = np.zeros((n_n,n_n))      # GABAa Synapse Connectivity Matrix

fgaba_mat[:,p_n:] = np.random.choice([0.,1.],size=(n_n,l_n)) # 50% probability of LN -> LN/PN

np.fill_diagonal(fgaba_mat,0.)       # No self connection

n_syn_fgaba = int(np.sum(fgaba_mat)) # Number of GABAa (fGABA) Synapses
alp_fgaba = [10.0]*n_syn_fgaba       # Alpha for fGABA Synapse
bet_fgaba = [0.16]*n_syn_fgaba       # Beta for fGABA Synapse
V0 = [-20.0]*n_n                     # Decay Potential
sigma = [1.5]*n_n                    # Decay Time Constant
g_fgaba = [0.8]*p_n+[0.8]*l_n        # fGABA Conductance
E_fgaba = [-70.0]*n_n                # fGABA Potential


# Other Parameters #

F_b = [0.0]*n_n                      # Fire potential
```

Now, we create the synaptic connectivity matrix for each type of synapse. It's advisable to maintain the convention as:

Column Number = Presynaptic Neuron
Row Number = Postsynaptic Neuron
1 = Connected, 0 = No Connection

Ensure there are no self connections.

# NerveFlow: Define Helper Functions (if required)

```python
# Property Dynamics #

def K_prop(V):

    T = 22

    phi = 3.0**((T-36.0)/10)

    V_ = V-(-50)

    alpha_n = 0.02*(15.0 - V_)/(tf.exp((15.0 - V_)/5.0) - 1.0)
    beta_n = 0.5*tf.exp((10.0 - V_)/40.0)

    t_n = 1.0/((alpha_n+beta_n)*phi)
    n_inf = alpha_n/(alpha_n+beta_n)

    return n_inf, t_n


def Na_prop(V):
    T = 22

    phi = 3.0**((T-36)/10)

    V_ = V-(-50)

    alpha_m = 0.32*(13.0 - V_)/(tf.exp((13.0 - V_)/4.0) - 1.0)
    beta_m = 0.28*(V_ - 40.0)/(tf.exp((V_ - 40.0)/5.0) - 1.0)

    alpha_h = 0.128*tf.exp((17.0 - V_)/18.0)
    beta_h = 4.0/(tf.exp((40.0 - V_)/5.0) + 1.0)

    t_m = 1.0/((alpha_m+beta_m)*phi)
    t_h = 1.0/((alpha_h+beta_h)*phi)

    m_inf = alpha_m/(alpha_m+beta_m)
    h_inf = alpha_h/(alpha_h+beta_h)

    return m_inf, t_m, h_inf, t_h
```

```python
def A_prop(V):
    T = 36

    phi = 3.0**((T-23.5)/10)

    m_inf = 1/(1+tf.exp(-(V+60.0)/8.5))
    h_inf = 1/(1+tf.exp((V+78.0)/6.0))

    tau_m = 1/(tf.exp((V+35.82)/19.69) + tf.exp(-(V+79.69)/12.7) + 0.37) / phi

    t1 = 1/(tf.exp((V+46.05)/5.0) + tf.exp(-(V+238.4)/37.45)) / phi
    t2 = (19.0/phi) * tf.ones(tf.shape(V),dtype=V.dtype)
    tau_h = tf.where(tf.less(V,-63.0),t1,t2)

    return m_inf, tau_m, h_inf, tau_h

def Ca_prop(V):

    m_inf = 1/(1+tf.exp(-(V+20.0)/6.5))
    h_inf = 1/(1+tf.exp((V+25.0)/12))

    tau_m = 1.5
    tau_h = 0.3*tf.exp((V-40.0)/13.0) + 0.002*tf.exp((60.0-V)/29)

    return m_inf, tau_m, h_inf, tau_h

def KCa_prop(Ca):
    T = 26

    phi = 2.3**((T-23.0)/10)

    alpha = 0.01*Ca
    beta = 0.02

    tau = 1/((alpha+beta)*phi)

    return alpha*tau*phi, tau
```

Conditionals should be dealt with carefully to ensure effectivity.

```python
# NEURONAL CURRENTS

# Common Currents #

def I_K(V, n):
    return g_K  * n**4 * (V - E_K)

def I_L(V):
    return g_L * (V - E_L)

def I_KL(V):
    return g_KL * (V - E_KL)

# PN Currents #

def I_Na(V, m, h):
    return g_Na * m**3 * h * (V - E_Na)

def I_A(V, m, h):
    return g_A * m**4 * h * (V - E_A)

# LN Currents #

def I_Ca(V, m, h):
    return g_Ca * m**2 * h * (V - E_Ca)

def I_KCa(V, m):
    T = 26
    phi = 2.3**((T-23.0)/10)
    return g_KCa * m * phi * (V - E_KCa)
```

Remember to perform all numerical evaluations using tensorflow functions. Note that these functions can take a scalar or vector. Thus, if we pass all the voltages at once, it will evaluate the dynamic variables, using the right parameters, parallely.

# NerveFlow: Dense Coding of Synapses

Presynaptic

Postsynaptic

| | | | |
|---|---|---|---|
| 0 | 1 | ... | 0 |
| 0 | 0 | ... | 1 |
| ... | ... | ... | 1 |
| 1 | 0 | 0 | 0 |

n x n connectivity matrix

As the number of neurons (n) increases
The number of possible synapses increase as the square of n, but realistically not all neurons are connected, thus to reduce memory and minimize number of calculations, it is best to convert the synaptic state as densely coded system.

There are two types of synaptic calculations:

1. Dynamics of [O] : Requires Sparse to Dense Conversion of Excitation Tendency [T] of Neurons

2. Calculation of Isyn : Requires Dense to Sparse Conversion of Open fraction [O] of Channels

# NerveFlow: Calculation of $I_{syn}$

Presynaptic

Postsynaptic

| | | | |
|---|---|---|---|
| [ 0 | 1 | ... | 0 ] |
| [ 0 | 0 | ... | 1 ] |
| [... | ... | ... | 1 ] |
| [ 1 | 0 | 0 | 0 ] |

$Matrix\ \mathbf{C}$

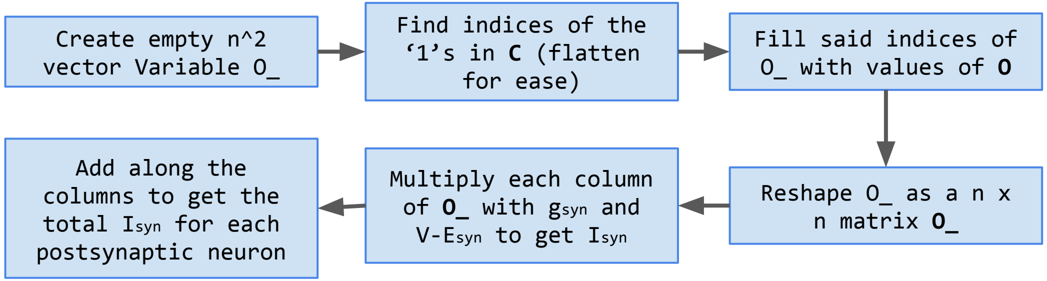
**C** has k '1's

| | | | |
|---|---|---|---|
| [$o_1$ | $o_2$ | ... | $o_k$] |

k x 1 vector to store fraction of open channels for k synapses

$Vector\ \vec{O}$

```
# SYNAPTIC CURRENTS

def I_ach(o,V):
    o_ = tf.Variable([0.0]*n_n**2,dtype=tf.float64)
    ind = tf.boolean_mask(tf.range(n_n**2),ach_mat.reshape(-1) == 1)
    o_ = tf.scatter_update(o_,ind,o)
    o_ = tf.transpose(tf.reshape(o_,(n_n,n_n)))
    return tf.reduce_sum(tf.transpose((o_*(V-E_ach))*g_ach),1)

def I_fgaba(o,V):
    o_ = tf.Variable([0.0]*n_n**2,dtype=tf.float64)
    ind = tf.boolean_mask(tf.range(n_n**2),fgaba_mat.reshape(-1) == 1)
    o_ = tf.scatter_update(o_,ind,o)
    o_ = tf.transpose(tf.reshape(o_,(n_n,n_n)))
    return tf.reduce_sum(tf.transpose((o_*(V-E_fgaba))*g_fgaba),1)
```




| | | | |
|---|---|---|---|
| [ 0 | $o_1$ | ... | 0 ] |
| [ 0 | 0 | ... | $o_a$] |
| [... | ... | ... | $o_b$] |
| [$o_k$ | 0 | 0 | 0 ] |

$Matrix\ \mathbf{O\_}$

# NerveFlow: Input Current and dX/dt function

Extract Current Input at Time Instant t (in ms)

```
# Current Input

def I_inj_t(t):
    return tf.constant(current_input.T,dtype=tf.float64)[tf.to_int32(t*100)]

# DIFFERENTIAL EQUATION FORM

def dAdt(X, t): # X is the state vector

    # Assign Current Values

    V_p   = X[0   : p_n]
    V_l   = X[p_n : n_n]

    n_K   = X[n_n : 2*n_n]

    m_Na  = X[2*n_n : 2*n_n + p_n]
    h_Na  = X[2*n_n + p_n : 2*n_n + 2*p_n]

    m_A   = X[2*n_n + 2*p_n : 2*n_n + 3*p_n]
    h_A   = X[2*n_n + 3*p_n : 2*n_n + 4*p_n]

    m_Ca  = X[2*n_n + 4*p_n : 2*n_n + 4*p_n + l_n]
    h_Ca  = X[2*n_n + 4*p_n + l_n: 2*n_n + 4*p_n + 2*l_n]

    m_KCa = X[2*n_n + 4*p_n + 2*l_n : 2*n_n + 4*p_n + 3*l_n]
    Ca    = X[2*n_n + 4*p_n + 3*l_n: 2*n_n + 4*p_n + 4*l_n]

    o_ach = X[6*n_n : 6*n_n + n_syn_ach]
    o_fgaba = X[6*n_n + n_syn_ach : 6*n_n + n_syn_ach + n_syn_fgaba]

    fire_t = X[-n_n:]

    V = X[:n_n]
```

Splice Different Variables from Common State Vector

```
# Evaluate Differentials

n0,tn = K_prop(V)

dn_k = - (1.0/tn)*(n_K-n0)

m0,tm,h0,th = Na_prop(V_p)

dm_Na = - (1.0/tm)*(m_Na-m0)
dh_Na = - (1.0/th)*(h_Na-h0)

m0,tm,h0,th = A_prop(V_p)

dm_A = - (1.0/tm)*(m_A-m0)
dh_A = - (1.0/th)*(h_A-h0)

m0,tm,h0,th = Ca_prop(V_l)

dm_Ca = - (1.0/tm)*(m_Ca-m0)
dh_Ca = - (1.0/th)*(h_Ca-h0)

m0,tm = KCa_prop(Ca)

dm_KCa = - (1.0/tm)*(m_KCa-m0)

dCa = - A_Ca*I_Ca(V_l,m_Ca,h_Ca) - (Ca - Ca0)/t_Ca

CmdV_p = - I_Na(V_p, m_Na, h_Na) - I_A(V_p, m_A, h_A)
CmdV_l = - I_Ca(V_l, m_Ca, h_Ca) - I_KCa(V_l, m_KCa)

CmdV = tf.concat([CmdV_p,CmdV_l],0)

dV = (I_inj_t(t) + CmdV - I_K(V, n_K) - I_L(V) - I_KL(V) - I_ach(o_ach,V) - I_fgaba(o_fgaba,V)) / C_m
```

$$\frac{d\vec{X}_A}{dt} = f(\vec{X}_{A_\alpha}, \vec{X}_{A_\beta}\dots) + g_A(\vec{X}_{A_\gamma}\dots)$$

$$\frac{d\vec{X}_B}{dt} = f(\vec{X}_{B_\alpha}, \vec{X}_{B_\beta}\dots) + g_B(\vec{X}_{B_\gamma}\dots)$$

$$To\ evaluate\ \frac{d\vec{X}}{dt}\ where\ \vec{X} = [\vec{X}_A, \vec{X}_B],$$

$$Find:$$

$$\left(\frac{dX_A}{dt}\right)_{unique} = g_A(\vec{X}_{A_\gamma}..)$$

$$\left(\frac{dX_B}{dt}\right)_{unique} = g_B(\vec{X}_{B_\gamma}..)$$

$$\left(\frac{dX}{dt}\right)_{unique} = \left[\left(\frac{dX_A}{dt}\right)_{unique}, \left(\frac{dX_B}{dt}\right)_{unique}\right]$$

$$\frac{dX}{dt} = \left(\frac{dX}{dt}\right)_{unique} + f(\vec{X}_\alpha, \vec{X}_\beta\dots)$$

Evaluate Unique Currents for different types and merge

Evaluate Common Currents

# NerveFlow: dX/dt function & Dynamics of [O]

```
A_ = tf.constant(A,dtype=tf.float64)
T_ach = tf.where(tf.logical_and(tf.greater(t,fire_t+t_delay),tf.less(t,fire_t+t_max+t_delay)),A_,tf.zeros(tf.shape(A_),dtype=A_.dtype))
T_ach = tf.multiply(tf.constant(ach_mat,dtype=tf.float64),T_ach)
T_ach = tf.boolean_mask(tf.reshape(T_ach,(-1,)),ach_mat.reshape(-1) == 1)
do_achdt = alp_ach*(1.0-o_ach)*T_ach - bet_ach*o_ach

T_fgaba = 1.0/(1.0+tf.exp(-(V-V0)/sigma))
T_fgaba = tf.multiply(tf.constant(fgaba_mat,dtype=tf.float64),T_fgaba)
T_fgaba = tf.boolean_mask(tf.reshape(T_fgaba,(-1,)),fgaba_mat.reshape(-1) == 1)
do_fgabadt = alp_fgaba*(1.0-o_fgaba)*T_fgaba - bet_fgaba*o_fgaba

dfdt = tf.zeros(tf.shape(fire_t),dtype=fire_t.dtype)

out = tf.concat([dV,          dn_k,
                 dm_Na,       dh_Na,
                 dm_A,        dh_A,
                 dm_Ca,       dh_Ca,
                 dm_KCa,
                 dCa,         do_achdt,
                 do_fgabadt,  dfdt    ],0)
return out
```

Merge all partial dX/dt terms by concatenating into single vector

Set changes in Firing Times as Zero as the updation is done independently by the Nerveflow integrator

$Vector\ [T\_]$

| | | | |
|---|---|---|---|
| $[T_1$ | $T_i$ | ... | $T_0]$ |

k x 1 vector to store presynaptic activation for k synapses

| | | | |
|---|---|---|---|
| [ 0 | $T_1$ | ... | 0 ] |
| [ 0 | 0 | ... | $T_n$] |
| [... | ... | ... | $T_n$] |
| $[T_0$ | 0 | 0 | 0 ] |

$Matrix\ [\mathbf{T}]$

Presynaptic

| | | | | |
|---|---|---|---|---|
| Postsynaptic | [ 0 | 1 | ... | 0 ] |
| | [ 0 | 0 | ... | 1 ] |
| | [... | ... | ... | 1 ] |
| | [ 1 | 0 | 0 | 0 ] |

$Matrix\ \mathbf{C}$

n x n matrix **C** has k '1's

From the Firing Times and Voltages of the presynaptic neurons, Determine nx1 vector[T] → Multiply [T] with each row of **C** to get matrix **[T]** which represents presynaptic activation → Flatten **C** and **[T]** and mask in the elem **[T]** where elem **C** is 1 to get k x 1 vec [T_] → Use [T_] to update [O] for the different types of synapses

# NerveFlow: Import Input and Run

```
current_input = np.load("current.npy")

state_vector =  [-70]* n_n + [0.0]* n_n + [0.0]* (4*p_n) + [0.0]* (3*l_n) + [2.4*(10**(-4))]*l_n + [0]*(n_syn_ach) + [0]*(n_syn_fgaba) + [-(sim_time+1)]*n_n
state_vector = np.array(state_vector)
state_vector = state_vector + 0.01*state_vector*np.random.normal(size=state_vector.shape)

print("Number of Neurons:",n_n)
print("Number of Synapses:",(n_syn_ach+n_syn_fgaba))

init_state = tf.constant(state_vector, dtype=tf.float64)
tensor_state = nv.odeint_fixed(dAdt, init_state, i, n_n, F_b)

with tf.Session() as sess:
    print("Session started...",end="")
    tf.global_variables_initializer().run()
    state = sess.run(tensor_state)
    sess.close()


np.savetxt("state.csv",state,delimiter=",",fmt='%.3f')
```

Create the Initial State and Add Random Noise

Define Computation Graph and Start Session

# NerveFlow: Batch and Merge

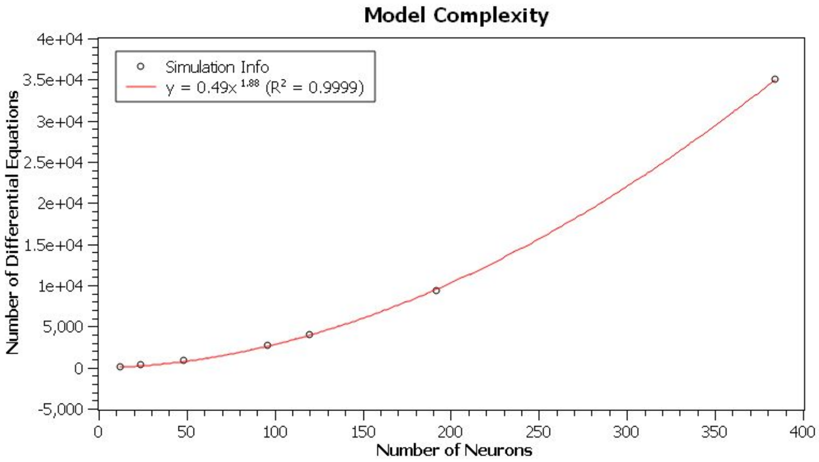
```python
n_batch = 10
t_batch = np.array_split(t,n_batch)

for n,i in enumerate(t_batch):

    print("Batch",(n+1),"Running...",end="")

    t0 = time.time()

    if n>0:
        i = np.append(i[0]-0.01,i)

    init_state = tf.constant(state_vector, dtype=tf.float64)
    tensor_state = nv.odeint_fixed(dAdt, init_state, i, n_n, F_b)

    with tf.Session() as sess:
        print("Session started...",end="")
        tf.global_variables_initializer().run()
        state = sess.run(tensor_state)
        sess.close()

    t1 = time.time()
    print("Finished in",np.round(t1-t0,2),"secs...Saving...",end="")

    state_vector = state[-1,:]
    np.save("state.batch"+str(n+1),state)

    state=None
    gc.collect()

    t2 = time.time()
    print("Saved ( Execution Time:",np.round(t2-t0,3),"secs )")
```

Maximum memory used is 2x the total state matrix size, Larger and more complicated the network and longer the simulation, larger the matrix and the run is limited by the total available memory. But this maximum limit of the 2x Matrix Size can be reduced to (1+1/K)x Matrix Size by implementing batches that run sequentially.

Ideally this should allow us to have infinitely long simulations but it doesn't. This is an intrinsic problem with Tensorflow. Memory is assigned at start and can only be cleaned after closing the python interpreter to avoid memory fragmentation.

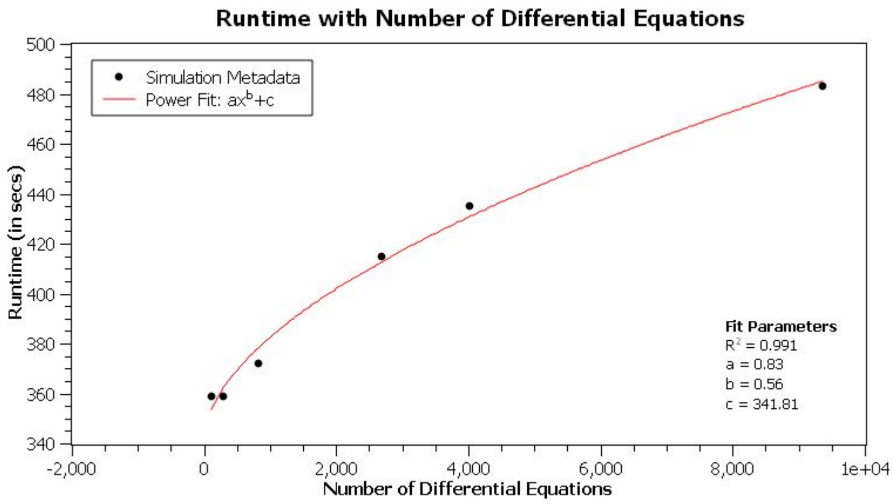
The only option is to rerun the code after each block of simulation.

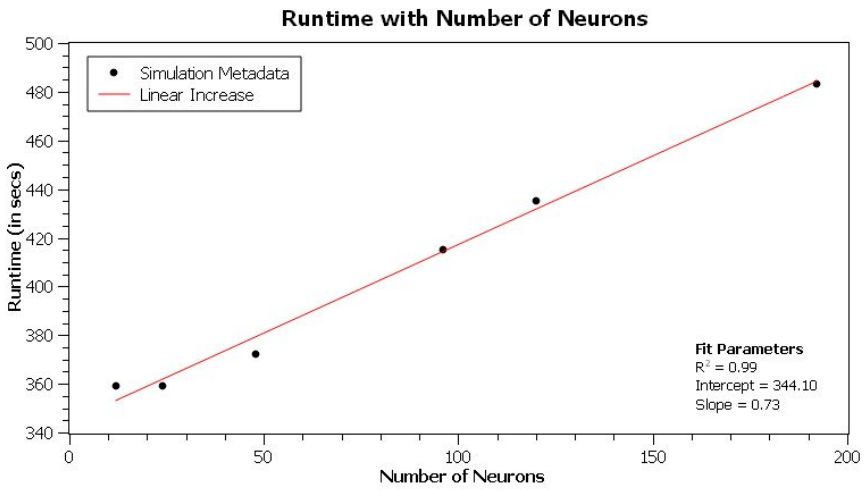
# NerveFlow: Preliminary Results



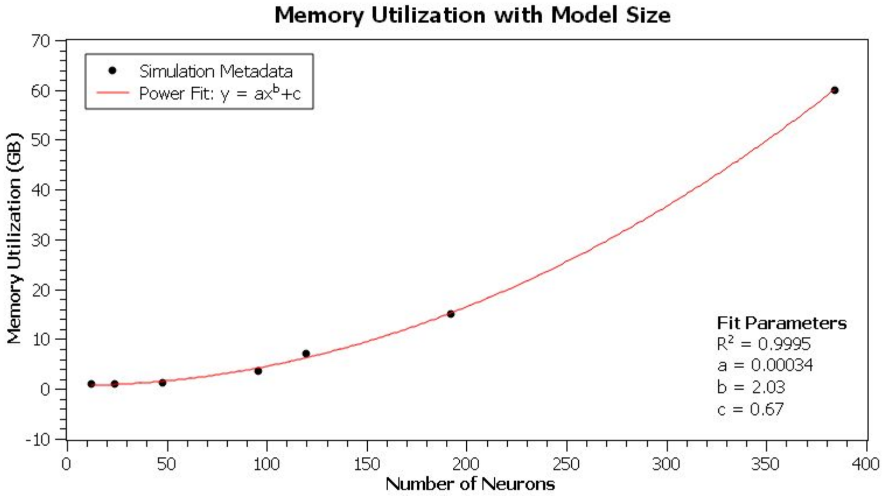

| Number of Neurons | Number of Synapses | Number of DE | Time of Simulation (ms) | Session Wall Time |
|---|---|---|---|---|
| 12 | 29 | 113 | 1000 | 359 |
| 24 | 130 | 298 | 1000 | 359 |
| 48 | 491 | 827 | 1000 | 372 |
| 96 | 2007 | 2679 | 1000 | 415 |
| 120 | 3183 | 4023 | 1000 | 435 |
| 192 | 8024 | 9368 | 1000 | 483 |
| 384 | 32295 | 34983 | 1000 | OOM |

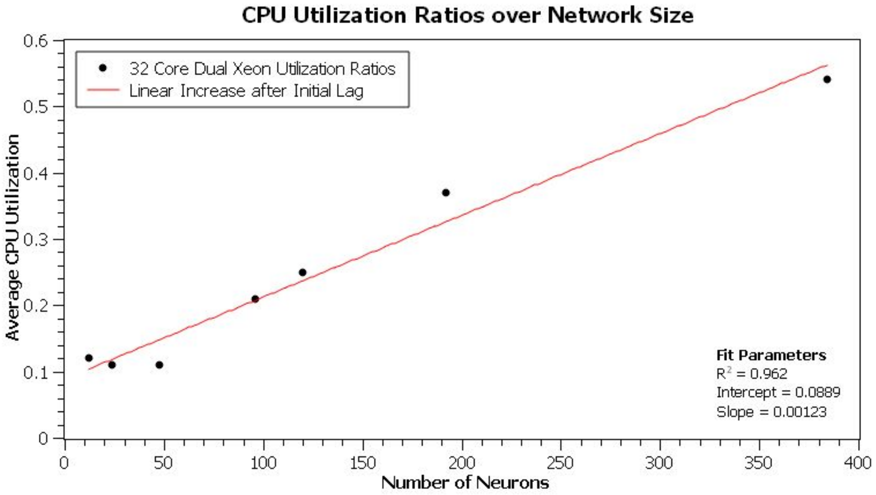
# NerveFlow: Preliminary Results



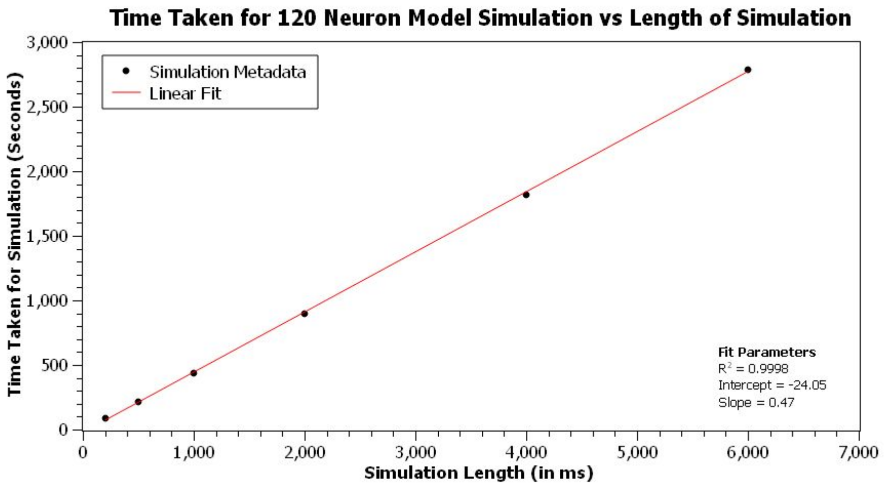



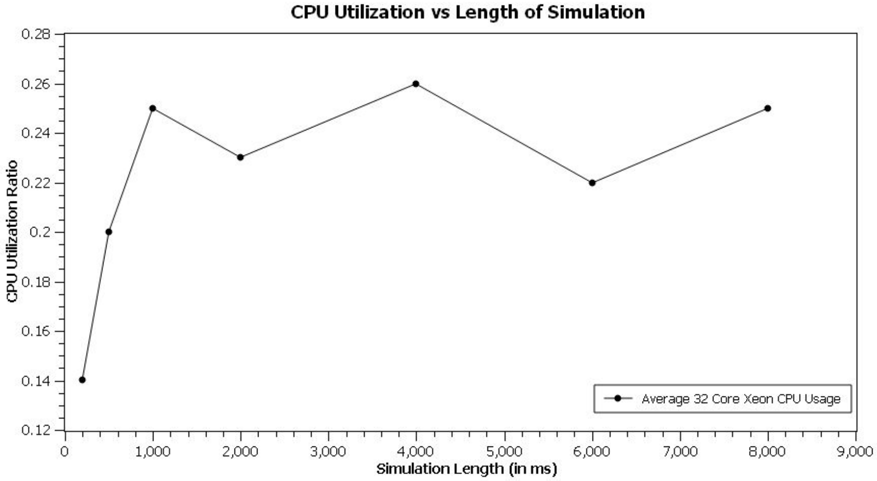
# NerveFlow: Preliminary Results



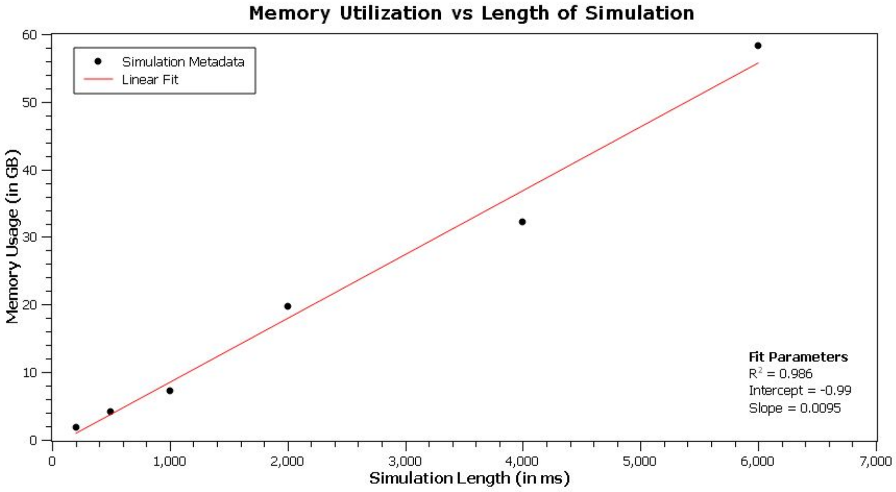



# NerveFlow: Preliminary Results



# NerveFlow: Preliminary Results




Further Testing on local GPU/TPU system for speedup is required.

**Thank You!**