# Parallelised Scalable Simulations of Biological Neural Networks using TensorFlow: A Beginners' Guide

Saptarshi Soham Mohanta[1], Collins Assisi[1], with Theoretical Neuroscience Group[¶].

**1** Indian Institute of Science Education and Research, Pune, Maharashtra, India

¤Current Address: Biology Department, Indian Institute of Science Education and Research, Pune, Maharashtra, India
¶Membership list can be found in the Acknowledgments section.
\* collins@iiserpune.ac.in

## Abstract

The dynamics of neurons and their networks have been studied extensively by modelling them as collections of differential equations, and the power of these mathematical tools are well recognised. Many tools and packages exist that allow the simulation of systems of neurons using these differential equations. However, there is a barrier of entry in terms of developing flexible general purpose simulations that are platform independent and support hardware acceleration on modern computing architectures such as GPUs/TPUs and Distributed Platforms. TensorFlow is a Python-based open-source package initially designed for machine learning algorithms, but it presents a scalable environment for a variety of computation including solving differential equations using iterative algorithms from numerical analysis such as Runge-Kutta methods. There are two significant benefits of such an implementation: high readability and scalability across a variety of computational devices. We explore the process of implementing a scalable simulation of a system of neurons based on Hodgkin-Huxley-like neuron equations using TensorFlow. We also discuss the limitations of such implementation and approaches to deal with them.

## Author summary

In the form of a 7-day tutorial, the reader is introduced to the mathematical modelling of neuronal networks based on the Hodgkin-Huxley Differential Equations and is instructed on developing highly parallelised but easily readable code for numerical methods such as Euler's Method and Runge-Kutta Methods to solve differential equations in Python and use them to simulate neuronal dynamics. To develop scalable code, Google's open-source package TensorFlow is introduced, and the reader is instructed in developing simulations using this package and handling the few limitations that come with this implementation. The reader is also introduced to coding structures that maximise the parallelizability of the simulation. Finally, the coding paradigm that was developed is used to simulate a model of Locus Antennal Lobe described in previous literature, and its efficacy is analysed.

## Motivation

The processing of information by the nervous system spans across space and time, and mathematical modelling of these dynamics have found to be an essential tool. These

models have been used extensively to study the dynamics and mechanisms of information processing at both the individual neuron level [] and the system of neurons level. [] These models generally utilise systems of simultaneous ordinary differential equations (ODEs) which are solved as initial value problems using well-studied methods from numerical analysis such as Euler's Method and Runge Kutta methods. [] From a detailed study of the mechanism of action of the neurons, ion channels, neurotransmitters or neuromodulators and their dynamics in different models, [] equations have been found that describe the behaviour of neurons and synapse. By forming interconnected systems of these individual groups of differential equations, the overall dynamics and behaviour of networks can be studied through deterministic or stochastic simulations which can be easily perturbed unlike the case for in vivo experiments.

A significant issue with such simulations is computational complexity. As the number of neurons increase, the number of possible synaptic connections increases quadratically. That is, for a system of $n$ neurons there can be at most $n^2$ different synapses of one type, each with its own set of equations. Thus, simulations can take very long times for large values of $n$. A solution to this problem is to implement some form of parallelisation in the ODE solver and the system of equations itself. One of the simplest methods of parallelizable computation is in the form of matrix algebra which can be accelerated using libraries such as BLAS which can only be used for accelerating CPU based computations. Similarly, CUDA is available for speeding up computations on Nvidia-based GPUs and TPUs. [] However, there is a barrier of entry to using low-level packages like CUDA for the general user as it sometimes requires an in-depth understanding of the architecture, particularly for troubleshooting.

This is where TensorFlow (an open-source Google product) [] gives us a massive edge. TensorFlow allows us much greater scalability and is way more flexible in terms of ease of implementation for specialised hardware. With minimal changes in the implementation, the code can be executed on a wide variety of heterogeneous and distributed systems ranging from mobile devices to clusters and specialised computing devices such as GPU and TPU cards. The modern advances in GPU/TPU design allow us to access even higher degrees of parallelisation. It is now even possible to have hundred of TeraFLOPS of computing power in a single small computing device. With TensorFlow, we can access these resources without even requiring an in-depth understanding of its architecture and technical knowledge of specific low-level packages like CUDA.

## How to use this Tutorial

For this tutorial, the reader is expected to have an understanding of Python and some commonly used packages such as Numpy and Matplotlib. The reader is also expected to know some amount of calculus particularly the theory of differential equations. Access to a computer will the following prerequisite software/packages is preferable: Python 3.6 or above, Jupyter Notebook, Numpy Python package, Matplotlib Python package, and TensorFlow 1.13 or above. All software can be installed using Anaconda Distribution of Python 3. Instructions for TensorFlow installation is available on their website.

## Day 1: Of Numerical Integration and Python

In this tutorial we are interested in solving ordinary differential equations of the form,

$$\frac{dx}{dt} = f(x, t) \tag{1}$$

where, $x$ is an $N-$dimensional vector and $t$ typically stands for time. The function $f(x, t)$ may be a complex nonlinear function of $x$. In addition to specifying the form of the differential equation, one must also specify the value of $x$ at a particular time point. Say, $x(t = t_0) = x_0$. It is often not possible to derive a closed form solution for 1. Therefore numerical methods to solve these equations are of great importance. One such example at the core of our tutorial are the Hodgkin-Huxley equations describing the dynamics of action potential generation and propagation in the giant axon of the squid. Alan Hodgkin and Andrew Huxley arrived at these equations after a series of clever experiments that tested the limits of experimental technology available at the time. It also tested the limits of computational tools available. The form of the differential equations they arrived at contained nonlinearities that made it impossible to solve analytically. In order to compute action potentials, Huxley numerically intergrated the equations using a hand operated Bunsviga mechanical calculator. The calculation took nearly three weeks to complete.

Here we discuss and code a class of numerical methods used to solve initial value problems 1 iteratively.

## Euler's Method for Numerical Integration

The simplest method to solve 1 numerically is the Euler's method. Here we start from $x(t = t_0) = x_0$ and compute the solution at subsequent time points $(t_0 + \epsilon, t_0 + 2\epsilon, t_0 + 3\epsilon \dots)$ iteratively. Each step of the computation is done using the same formula that can be derived by truncating a Taylor approximation of the solution after the first term. That is, the solution at time $t + \epsilon$ is given by,

$$x(t + \epsilon) = x_0 + \epsilon \frac{dx}{dt} + \mathcal{O}(\epsilon^2) \tag{2}$$

where $\frac{dx}{dt} = f(x, t)$. The higher order terms $\mathcal{O}(\epsilon^2)$ are ignored in this approximation. A geometric interpretation of the Euler's method is shown in figure ??.

These methods are iterative in that we start from a particular value of the solution (the initial condition, $x(t = 0) = x_0$) and compute an approximate value of the solution at subsequent steps. Euler's method is the simplest Euler's Method is used to generate a numerical solution to an initial value problem of the form:

$$\frac{dx}{dt} = f(x, t) \tag{3}$$

$$x(t_o) = x_o \tag{4}$$

Firstly, we decide the interval over which we desire to find the solution, starting at the initial condition. We break this interval into small subdivisions of a fixed length $\epsilon$. Then, using the initial condition as our starting point, we generate the rest of the solution by using the iterative formulas:

$$t_{n+1} = t_n + \epsilon \tag{5}$$

$$x_{n+1} = x_n + \epsilon f(x_n, t_n) \tag{6}$$

to find the coordinates of the points in our numerical solution. We end this process once we have reached the end of the desired interval. For an graphical description of the Euler's method, take a look at Fig **??**.

## Euler's Method in Python

Let $\frac{dx}{dt} = f(x,t)$, we want to find $x(t)$ over $t \in [0,2)$, given that $x(0) = 1$ and
$f(x,t) = 5x$. The exact solution of this equation would be $x(t) = e^{5t}$. We implement
the Euler's Method in Python as follows:

```python
import numpy as np
import matplotlib.pyplot as plt
def f(x,t): # define the function f(x,t)
    return 5*x
epsilon = 0.01 # define timestep
t = np.arange(0,2,epsilon) # define an array for t
x = np.zeros(t.shape) # define an array for x
x[0]= 1 # set initial condition
for i in range(1,t.shape[0]):
    x[i] = epsilon*f(x[i-1],t[i-1])+x[i-1] # Euler Integration Step
```

The exact solution and the numerical solution are compared in Fig **??**.

## Euler and Vectors

Euler's Method can also be easily applied to systems of equations. We can write the
IVP of multiple dynamical variables as a vector equation and by changing the Euler's
Algorithm to accomodate vectors, we can solve simultaneous differential equations.
The Initial Value problem now becomes:

$$\frac{d\vec{X}}{dt} = \vec{f}(\vec{X},t) \tag{7}$$

$$\vec{X}(t_o) = \vec{X_o} \tag{8}$$

where $\vec{X} = [X_1, X_2...]$ and $\vec{f}(\vec{X},t) = [f_1(\vec{X},t), f_2(\vec{X},t)...]$. We rewrite the Euler's
Method as:

$$t_{n+1} = t_n + \epsilon \tag{9}$$

$$\vec{X_{n+1}} = \vec{X_n} + \epsilon\vec{f}(\vec{X_n},t_n) \tag{10}$$

Let $\frac{d\vec{X}}{dt} = f(\vec{X},t)$, we want to find $\vec{X}(t)$ over $t \in [0,2)$, given that $\vec{X}(t) = [x,y]$,
$\vec{X}(0) = [1,0]$ and $f(\vec{X},t) = [x-y, y-x]$. We implement the modified algorithm as
follows:

```python
def f(X,t): # define the function f(x,t)
    x,y = X
    return np.array([x-y,y-x])
t = np.arange(0,2,epsilon) # define an array for t
X = np.zeros((2,t.shape[0])) # define an array for x
X[:,0]= [1,0] # set initial condition
for i in range(1,t.shape[0]):
    X[:,i] = epsilon*f(X[:,i-1],t[i-1])+X[:,i-1] # Euler Integration Step
```

## A Generalized function for Euler Integration

We create a generalized function that takes in 3 inputs ie. the function $\vec{f}(\vec{y}, t)$ when $\frac{d\vec{y}}{dt} = f(\vec{y}, t)$, the time array, and initial vector $\vec{y_0}$.

### Algorithm

- Get the required inputs: function $\vec{f}(\vec{y}, t)$, initial condition vector $\vec{y_0}$ and time series $t$. Entering a time series $t$ allows for greater control over $\epsilon$ as it can now vary for each timestep. The only difference in the Euler's Method is now : $\epsilon \to \epsilon(t_n)$.

- Check if the input is of the correct datatype ie. floating point decimal.

- Create a zero matrix to hold the output.

- For each timestep, perform the euler method updation with variable $\epsilon$ and store it in the output matrix.

- Return the output timeseries matrix.

```python
def check_type(y,t): # Ensure Input is Correct
    return y.dtype == np.floating and t.dtype == np.floating
class _Integrator():
    def integrate(self,func,y0,t):
        time_delta_grid = t[1:] - t[:-1]
        y = np.zeros((y0.shape[0],t.shape[0]))
        y[:,0] = y0
        for i in range(time_delta_grid.shape[0]):
            y[:,i+1]= time_delta_grid[i]*func(y[:,i],t[i])+y[:,i]
        return y
def odeint_euler(func,y0,t):
    y0 = np.array(y0)
    t = np.array(t)
    if check_type(y0,t):
        return _Integrator().integrate(func,y0,t)
    else:
        print("error encountered")
solution = odeint_euler(f,[1.,0.],t)
```

## Runge-Kutta Methods for Numerical Integration

The formula for the Euler method is $x_{n+1} = x_n + \epsilon f(x_n, t_n)$ which takes a solution from $t_n$ to $t_{n+1} = t_n + \epsilon$. One might notice there is an inherent assymetry in the formula. It advances the solution through an interval $\epsilon$, but uses the derivative information at only the start of the interval. This results in an error in the order of $O(\epsilon^2)$. But, what if we take a trial step and evaluate the derivative at the midpoint of the update interval to evaluate the value of $y_{n+1}$? Take the equations:

$$k_1 = \epsilon f(x_n, t_n) \tag{11}$$

$$k_2 = \epsilon f(x_n + \frac{k_1}{2}, t_n + \frac{\epsilon}{2}) \tag{12}$$

$$y_{n+1} = y_n + k_2 + O(\epsilon^3) \tag{13}$$

The symmetrization removes the $O(\epsilon^2)$ error term and now the method is second order and called the second order Runge-Kutta method or the midpoint method. Again, we can look at the method graphically in Fig **??**.

But we do not have to stop here. By further rewriting the equation, we can cancel higher order error terms and reach the most commonly used fourth-order Runge-Kutta Methods or RK4 method, which is described below:

$$k_1 = f(x_n, t_n) \tag{14}$$

$$k_2 = f(x_n + \epsilon\frac{k_1}{2}, t_n + \frac{\epsilon}{2}) \tag{15}$$

$$k_3 = f(x_n + \epsilon\frac{k_2}{2}, t_n + \frac{\epsilon}{2}) \tag{16}$$

$$k_4 = f(x_n + \epsilon k_3, t_n + \epsilon) \tag{17}$$

$$y_{n+1} = y_n + \frac{\epsilon}{6}(k_1 + 2k_2 + 2k_3 + k_4) + O(\epsilon^5) \tag{18}$$

Note that this numerical method is again easily converted to a vector algorithm by simply replacing $x_i$ by the vector $\vec{X}_i$. This method is what we will use to simulate our networks.

## Generalized RK4 Method in Python

Just like we had created a function for Euler Integration in Python, we create a generalized function for RK4 that takes in 3 inputs ie. the function $f(\vec{y}, t)$ when $\frac{d\vec{y}}{dt} = f(\vec{y}, t)$, the time array, and initial vector $\vec{y_0}$. We then perform the exact same integration that we had done with Euler's Method. Everything remains the same except we replace the Euler's method updation rule with the RK4 update rule.

```
# RK4 Integration Steps replace the Euler's Updation Steps
k1 = func(y[:,i], t[i])
half_step = t[i] + time_delta_grid[i] / 2
k2 = func(y[:,i] + time_delta_grid[i] * k1 / 2, half_step)
k3 = func(y[:,i] + time_delta_grid[i] * k2 / 2, half_step)
k4 = func(y[:,i] + time_delta_grid[i] * k3, t + time_delta_grid[i])
y[:,i+1]= (k1 + 2 * k2 + 2 * k3 + k4) * (time_delta_grid[i] / 6) + y[:,i]
```

As an **Exercise**, try to solve the equation of a simple pendulum and observe its dynamics using Euler Method and RK4 methods. The equation of motion of a simple pendulum is given by:

$$\frac{d^2 s}{dt^2} = L\frac{d^2\theta}{dt^2} = -g\sin\theta \tag{19}$$

where $L$ = Length of String and $\theta$ = angle made with vertical. To solve this second order differential equation you may use a dummy variable $\omega$ representing angular velocity such that:

$$\frac{d\theta}{dt} = \omega \tag{20}$$

$$\frac{d\omega}{dt} = -\frac{g}{L}\sin\theta \tag{21}$$

# Day 2: Let the Tensors Flow! <sub>150</sub>

In this section, we introduce TensorFlow followed by the implementation of Numerical <sub>151</sub>
Integration techniques in TensorFlow. <sub>152</sub>

## An Introduction to TensorFlow <sub>153</sub>

TensorFlow is an open-source software library. TensorFlow was originally developed by <sub>154</sub>
researchers and engineers working on the Google Brain Team within Google's Machine <sub>155</sub>
Intelligence research organization for the purposes of conducting machine learning and <sub>156</sub>
deep neural networks research, but the system is general enough to be applicable in a <sub>157</sub>
wide variety of other domains as well. [] <sub>158</sub>

Essentially, TensorFlow library for high performance numerical computation. Its <sub>159</sub>
flexible architecture allows easy deployment of computation across a variety of <sub>160</sub>
platforms (CPUs, GPUs, TPUs), and from desktops to clusters of servers. It is a python <sub>161</sub>
package that (much like BLAS on Intel MKL) speeds up Linear Algebra Computation. <sub>162</sub>
What is special about this system is that it is capable of utilizing GPUs and TPUs for <sub>163</sub>
computation and its written in a simpler language like python. <sub>164</sub>

## Why GPU/TPU vs CPU? <sub>165</sub>

The answer lies in the architecture Fig **??**: <sub>166</sub>

- CPU = Faster per Core Processing, Slow but Large Memory Buffer, Few Cores <sub>167</sub>

- GPU/TPU = Slower Processing, Faster but Smaller Memory Buffer, Many Cores <sub>168</sub>

Thus GPUs and TPUs are optimized for large number of simple calculations done <sub>169</sub>
parallely. The extent of this parallelization makes it suitable for vector/tensor <sub>170</sub>
manipulation. <sub>171</sub>

## How TensorFlow works? <sub>172</sub>

TensorFlow essentially performs numerical computations on multidimensional arrays <sub>173</sub>
(called tensors) using "data flow graphs" or "computation graphs". In these graphs, <sub>174</sub>
nodes represent variables/placeholders/constants or mathematical operations such as <sub>175</sub>
matrix multiplication or elementwise addition and the edges in the graph represent the <sub>176</sub>
flow of tensors between operations (nodes). Tensor is the central unit of data in <sub>177</sub>
TensorFlow giving the package its name. For example, look at the following computation <sub>178</sub>
graph in Fig **??** where "matmul" is the node which represents the matrix multiplication <sub>179</sub>
operation. a and b are input matrices (2-D tensors) and c is the resultant matrix. <sub>180</sub>

## Implementing a computation graph in TensorFlow <sub>181</sub>

```python
# Creating nodes in the computation graph
a = tf.constant([[1.],[2.],[3.]], dtype=tf.float64) # a 3x1 column matrix
b = tf.constant([[1.,2.,3.]], dtype=tf.float64) # a 1x3 row matrix
c = tf.matmul(a, b)
# To run the graph, we need to create a session.
# Creating the session initializes the computational device.
sess = tf.Session() # start a session
output = sess.run(c) # compute the value of c
sess.close() # end the session
print(output)
```

```
11    # TO automatically close the session after computation, Use:
12    # with tf.Session() as sess:
13    #     output = sess.run(c)
```

## Iterating recursively over lists/arrays in TensorFlow

Say, one wants to recursively apply a function on an initial value but the function takes in additional input at every recursive call, for example, to find a cumulative sum over a list. Every step is addition of the new element from the list onto the last addition. The TensorFlow function tf.scan allows us to easily implement such an iterator.

```
1     # define the recursive function that takes in two values the
2     # accumulated value and the additional input from a list.
3     def recursive_addition(accumulator,new_element):
4         return accumulator+new_element
5     # define the list over which we iterate
6     elems = np.array([1, 2, 3, 4, 5, 6])
7     # tf.scan takes in three inputs: the recursive function, the
8     # list to iterate over and the initial value. If an initial
9     # value is not provided, its taken as the first element of elems.
10    # accumulate with no initializer
11    cum_sum_a = tf.scan(recursive_addition, elems)
12    # accumulate with initializer as the number 5
13    cum_sum_b = tf.scan(recursive_addition, elems, tf.constant(5,dtype=tf.int64))
14    with tf.Session() as sess:
15        output_a = sess.run(cum_sum_a)
16        output_b = sess.run(cum_sum_b)
17    print(output_a)
18    print(output_b)
19    # This prints :
20    #[ 1   3   6 10 15 21]
21    #[ 6   8 11 15 20 26]
```

As an **Exercise** try using tf.scan to compute the fibonacci sequence. One may have noticed that our integration is essentially an recursive process over the time series. Both the updation rules can just be written as a recursive function $F$ such that $X_{i+1} = F(X_i, t_i, \epsilon_i)$. We would want our output to be the array $[X_0, F(X_0, t_0, \epsilon_0), F(F(X_0, t_0, \epsilon_0), t_1, \epsilon_1)...]$. To get this, we just need to recursively iterate over the array of time and $\epsilon$ and apply a updation function $F$ recursively on the initial condition, which exactly what tf.scan enables us to do.

## Euler Integration Function in TensorFlow

Transitioning to TensorFlow is not a trivial process, but once you get a gist of how it works, it is as simple as using numpy. Because of the way the TensorFlow architecture is designed, there are a few limitations to how one can do simpler operations/manipulation. But it is easy to overcome using the correct function and code patterns which can be easily learnt.

```
1     def tf_check_type(t, y0): # Ensure Input is Correct
2         if not (y0.dtype.is_floating and t.dtype.is_floating):
3             # The datatype of any tensor t is accessed by t.dtype
4             raise TypeError('Error in Datatype')
5     class _Tf_Integrator():
```

```
6       def integrate(self, func, y0, t):
7           time_delta_grid = t[1:] - t[:-1]
8           def scan_func(y, t_dt):
9               t, dt = t_dt
10              dy = dt*func(y,t)
11              return y + dy
12          # iterating over (a,b) where a and b are lists of same size
13          # results in the ith accumulative step in tf.scan receiving
14          # the ith elements of a and b zipped together
15          y = tf.scan(scan_func, (t[:-1], time_delta_grid),y0)
16          return tf.concat([[y0], y], axis=0)
17  def tf_odeint_euler(func, y0, t):
18      # Convert input to TensorFlow Objects
19      t = tf.convert_to_tensor(t, preferred_dtype=tf.float64, name='t')
20      y0 = tf.convert_to_tensor(y0, name='y0')
21      tf_check_type(y0,t)
22      return _Tf_Integrator().integrate(func,y0,t)
23
24  # Define a function using Tensorflow math operations.
25  # This creates the computation graph.
26  def f(X,t):
27      # extracting a single value eg. X[0] returns a single value but
28      # we require a tensor, so we extract a range with one element.
29      x = X[0:1]
30      y = X[1:2]
31      out = tf.concat([x-y,y-x],0)
32      return out
33  y0 = tf.constant([1,0], dtype=tf.float64)
34  epsilon = 0.01
35  t = np.arange(0,2,epsilon)
36  # Define the final value (output of scan) that we wish to compute
37  state = tf_odeint_euler(f,y0,t)
38  # Start a TF session and evaluate state
39  with tf.Session() as sess:
40      state = sess.run(state)
```

## RK4 Integration Function in TensorFlow

Now, we implement the exact same RK4 integrator in TensorFlow, thus we make a small change in the integrate function. Also to make the code more modular, we introduce the differential step calculator to a different function, everything else remains the same.

```
1   def integrate(self, func, y0, t):
2           time_delta_grid = t[1:] - t[:-1]
3           def scan_func(y, t_dt):
4               t, dt = t_dt
5               dy = self._step_func(func,t,dt,y) # Make code more modular.
6               return y + dy
7           y = tf.scan(scan_func, (t[:-1], time_delta_grid),y0)
8           return tf.concat([[y0], y], axis=0)
9       def _step_func(self, func, t, dt, y):
10          k1 = func(y, t)
11          half_step = t + dt / 2
```

```
12          dt_cast = tf.cast(dt, y.dtype) # Failsafe
13          k2 = func(y + dt_cast * k1 / 2, half_step)
14          k3 = func(y + dt_cast * k2 / 2, half_step)
15          k4 = func(y + dt_cast * k3, t + dt)
16          return tf.add_n([k1, 2 * k2, 2 * k3, k4]) * (dt_cast / 6)
```

As an **Exercise**, try to simulate the non-linear Lorentz Attractor using Euler Method and RK4 on TensorFlow which is given by the equations:

$$\frac{dx}{dt} = \sigma(y - x) \tag{22}$$

$$\frac{dy}{dt} = x(\rho - z) - y \tag{23}$$

$$\frac{dz}{dt} = xy - \beta z \tag{24}$$

Use the values $\sigma = 10$, $\beta = \frac{8}{3}$, $\rho = 28$. You can try simulating this system at two nearby starting conditions and comment on the difference.

# Day 3: Cells in Silicon

We start with our discussion with the Hodgkin Huxley Neurons and how we can simulate them in python using Tensorflow and Numerical Integration.

## What is the Hodgkin Huxley Neuron Model?

Hodgkin and Huxley performed many experiments on the giant axon of the squid and found three different types of ion currents - sodium, potassium, and a leak current. They found that specific voltage-dependent ion channels, for sodium and for potassium, control the flow of those ions through the cell membrane from different electrophysiology studies involving phamacological blocking of ion channels. The leak current essentially takes care of other channel types which are not described explicitly.

The Hodgkin-Huxley model of neurons can easily be understood with the help of a RC circuit diagram.(Fig) The semipermeable cell membrane separates the interior of the cell from the extracellular liquid and acts as a capacitor. If an input current I(t) is injected into the cell, it may add further charge on the capacitor, or leak through the channels in the cell membrane. Because of active ion transport through the cell membrane, the ion concentration inside the cell is different from that in the extracellular liquid. The Nernst potential generated by the difference in ion concentration is represented by a battery.

We now convert the above considerations into mathematical equations. The conservation of electric charge on a piece of membrane implies that the applied current $I(t)$ may be split in a capacitive current $I_C$ which charges the capacitor $C_m = 1\mu F/cm^2$ and further components $I_k$ which pass through the ion channels. Thus $I(t) = I_C(t) + \sum_k I_k(t)$ where the sum runs over all ion channels.

In the standard Hodgkin-Huxley model, there are only three types of channel: a Sodium channel, a Potassium channel and an unspecific leakage channel. From the definition of a capacitance $C_m = \frac{q}{u}$, $I_C = C_m \frac{du}{dt}$ where $q$ is a charge and $u$ the voltage across the capacitor. Thus the model becomes:

$$C_m \frac{du}{dt} = -I_{Na}(t) - I_K(t) - I_L(t) + I(t) \tag{25}$$

In biological terms, $u$ is the voltage across the membrane. Hogkin and Huxley found the Na and K ion currents to be dependent on the voltage and of the form given below:

$$I_{Na} = g_{Na} m^3 h (u - E_{Na}) \tag{26}$$

$$I_K = g_K n^4 (u - E_K) \tag{27}$$

$$I_L = g_L (u - E_L) \tag{28}$$

where $E_{Na} = 50\ mV$, $E_K = -95\ mV$ and $E_L = -55\ mV$ are the reversal potentials; $g_{Na} = 100\ \mu S/cm^2$, $g_K = 10\ \mu S/cm^2$ and $g_L = 0.15\ \mu S/cm^2$ are the channel conductances; and m,h, and n are gating variables that follow the dynamics given by:

$$\frac{dm}{dt} = -\frac{1}{\tau_m}(m - m_0) \tag{29}$$

$$\frac{dh}{dt} = -\frac{1}{\tau_h}(h - h_0) \tag{30}$$

$$\frac{dn}{dt} = -\frac{1}{\tau_n}(n - n_0) \tag{31}$$

where $\tau_m$, $\tau_h$ and $\tau_n$ are voltage dependent time constants and $m_0$, $h_0$ and $n_0$ are voltage dependent asymptotic gating values. These functions are empirically determined for different types of neurons. For an example, take a look at figure.

## Implementing the Dynamical Function for an Hodkin Huxley Neuron

For integration, we will use the TensorFlow RK4 integrator that we created. A simple Hodgkin Huxley Neuron has a 4 main dynamical variables: V = Membrane Potential, m = Sodium Activation Gating Variable, h = Sodium Inactivation Gating Variable, and n = Potassium Channel Gating Variable. And the dynamics are given by Eq (25), Eq (29),Eq (30) and Eq (31) where the values of $\tau_m$, $\tau_h$, $\tau_n$, $m_0$, $h_0$, $n_0$ are given empirically derived formulae (Fig) and channel currents are determined by Eq (26),Eq (27) and Eq (28).

```
# Step 1: Defining Parameters of the Neuron
C_m = 1
g_K = 10
E_K = -95
g_Na = 100
E_Na = 50
g_L = 0.15
E_L = -55
# Step 2: Defining functions to calculate tau_x and x_0
```

```
10   # Note: Always use TensorFlow functions for all operations.
11   def K_prop(V):
12       T = 22
13       phi = 3.0**((T-36.0)/10)
14       V_ = V-(-50)
15       alpha_n = 0.02*(15.0 - V_)/(tf.exp((15.0 - V_)/5.0) - 1.0)
16       beta_n = 0.5*tf.exp((10.0 - V_)/40.0)
17       t_n = 1.0/((alpha_n+beta_n)*phi)
18       n_0 = alpha_n/(alpha_n+beta_n)
19       return n_0, t_n
20   def Na_prop(V):
21       T = 22
22       phi = 3.0**((T-36)/10)
23       V_ = V-(-50)
24       alpha_m = 0.32*(13.0 - V_)/(tf.exp((13.0 - V_)/4.0) - 1.0)
25       beta_m = 0.28*(V_ - 40.0)/(tf.exp((V_ - 40.0)/5.0) - 1.0)
26       alpha_h = 0.128*tf.exp((17.0 - V_)/18.0)
27       beta_h = 4.0/(tf.exp((40.0 - V_)/5.0) + 1.0)
28       t_m = 1.0/((alpha_m+beta_m)*phi)
29       t_h = 1.0/((alpha_h+beta_h)*phi)
30       m_0 = alpha_m/(alpha_m+beta_m)
31       h_0 = alpha_h/(alpha_h+beta_h)
32       return m_0, t_m, h_0, t_h
33   # Step 3: Defining function that calculate Neuronal currents
34   def I_K(V, n):
35       return g_K  * n**4 * (V - E_K)
36   def I_Na(V, m, h):
37       return g_Na * m**3 * h * (V - E_Na)
38   def I_L(V):
39       return g_L * (V - E_L)
40   # Step 4: Define the function dX/dt where X is the State Vector
41   def dXdt(X, t):
42       V = X[0:1]
43       m = X[1:2]
44       h = X[2:3]
45       n = X[3:4]
46       dVdt = (5 - I_Na(V, m, h) - I_K(V, n) - I_L(V)) / C_m
47       # Here the current injection I_injected = 5 uA
48       m0,tm,h0,th = Na_prop(V)
49       n0,tn = K_prop(V)
50       dmdt = - (1.0/tm)*(m-m0)
51       dhdt = - (1.0/th)*(h-h0)
52       dndt = - (1.0/tn)*(n-n0)
53       out = tf.concat([dVdt,dmdt,dhdt,dndt],0)
54       return out
55   # Step 5: Define Initial Condition and Integrate
56   y0 = tf.constant([-71,0,0,0], dtype=tf.float64)
57   epsilon = 0.01
58   t = np.arange(0,200,epsilon)
59   state = odeint(dXdt,y0,t)
60   with tf.Session() as sess:
61       state = sess.run(state)
```

## Simulating Multiple Independent HH Neurons at the Same Time

Although, simulating a Single Hodgkin-Huxley Neuron is possible in TensorFlow, the real ability of tensorflow can be seen only when a large number of simultaneous diffential equations are to be solved at the the same time. Let's try to simulate 20 independent HH neurons with different input currents and characterise the firing rates.

### Methods of Parallelization

TensorFlow has the intrinsic ability to speed up any and all Tensor computations using available multi-cores, and GPU/TPU setups. There are two major parts of the code where TensorFlow can help us really speed up the computation:

1. **RK4 Steps:** Since the TensorFlow implementation of the Integrator utilizes Tensor calculations, TensorFlow will automatically speed it up.

2. **Functional Evaluations:** Looking at Dynamical Equations that describe the neuronal dynamics, its easy to notice that all simple HH Neurons share the same or atleast similar dynamical equations but will vary only in the values of parameters. We can exploit this to speed up the computations.

Say $\vec{X} = [V, m, n, h]$ is the state vector of a single neuron and its dynamics are defined using parameters $C_m, g_K, ...E_L$ equations of the form:

$$\frac{d\vec{X}}{dt} = [f_1(\vec{X}, C_m, g_K, ...E_L), f_2(\vec{X}, C_m, g_K, ...E_L)...f_m(\vec{X}, C_m, g_K, ...E_L)] \qquad (32)$$

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

So, what we need for a system of n neurons is to have a method to evaluate the updation of $\mathbf{X} = [\vec{X_1}, \vec{X_2}...\vec{X_n}]$ where $\vec{X_i} = [V_1, m_1, n_1, h_1]$ is the state vector of the $i$th neuron. Now there is a simple trick that allows us to maximize the parallel processing. Each neuron represented by $\vec{X_i}$ has a distinct set of parameters and differential equations.

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 $\mathbf{X}$ as $\mathbf{X'} = [(V_1, V_2, ...V_n), (m_1, m_2, ...m_n), (h_1, h_2, ...h_n), (n_1, n_2, ...n_n)] = [\vec{V}, \vec{m}, \vec{h}, \vec{n}]$. And the parameters as $\vec{C_m}, \vec{g_K}$ and so on.

Now that we know the trick, what is the benefit? Earlier, each state variable (say $V_i$) had a DE of the form:

$$\frac{dV_i}{dt} = f(V_i, m_i, h_i, n_i, C_{m_i}, g_{K_i}...) \qquad (33)$$

This is now easily parallelizable using a vector computation of a form:

$$\frac{d\vec{V}}{dt} = f(\vec{V}, \vec{m}, \vec{h}, \vec{n}, \vec{C_m}, \vec{g_K}...) \qquad (34)$$

Thus we can do the calculations as:

$$\frac{d\mathbf{X'}}{dt} = \Big[\frac{d\vec{V}}{dt}, \frac{d\vec{m}}{dt}, \frac{d\vec{h}}{dt}, \frac{d\vec{n}}{dt}\Big] \qquad (35)$$

**Implementation**

Notice that the functions calculating gating dynamics and channel currents already are capable of vector input and output, so we do not need to change them. What we do need to change is the parameters which should now be vectors, the differential evaluation function which should now be designed to handle grouped parameters, and finally the initial condition.

```
n_n = 20 # number of simultaneous neurons to simulate
# parameters will now become n_n-vectors
C_m = [1.0]*n_n
g_K = [10.0]*n_n
E_K = [-95.0]*n_n
g_Na = [100]*n_n
E_Na = [50]*n_n
g_L = [0.15]*n_n
E_L = [-55.0]*n_n

# The state vector definition will change
def dXdt(X, t):
    V = X[:1*n_n]          # First n_n values are Membrane Voltage
    m = X[1*n_n:2*n_n]    # Next n_n values are Sodium Activation Gating
    h = X[2*n_n:3*n_n]    # Next n_n values are Sodium Inactivation Gating
    n = X[3*n_n:]          # Last n_n values are Potassium Gating
    dVdt = (np.linspace(0,10,n_n)-I_Na(V, m, h)-I_K(V, n)-I_L(V))/ C_m
    # Input current is linearly varied between 0 and 10
    m0,tm,h0,th = Na_prop(V)
    n0,tn = K_prop(V)
    dmdt = - (1.0/tm)*(m-m0)
    dhdt = - (1.0/th)*(h-h0)
    dndt = - (1.0/tn)*(n-n0)
    out = tf.concat([dVdt,dmdt,dhdt,dndt],0)
    return out
y0 = tf.constant([-71]*n_n+[0,0,0]*n_n, dtype=tf.float64)
```

The output can be seen in Fig.

## Quantifying the Firing Rates against Input Current

One way to quantify the firing rate is to perform a fourier analysis and find peak frequency, but an easier way to find the rate is to see how many times it crosses a threshold say 0 mV in a given time, here it is for 200ms = 0.2s, and find the rate. The output can be seen in Fig.

```
rate = np.bitwise_and(state[:-1,:20]<0,state[1:,:20]>0).sum(axis=0)/0.2
```

# Day 4: Neurons and Networks

We discuss the realistic modelling of synaptic interactions for networks of Hogkin Huxley Neurons and how to implement synapses in TensorFlow.

## How do we model Synapses?

A synapse is defined between two neurons. There can be multiple synapses between two neurons (even of the same type) but here we will consider only single synapses between two neuron. So, there can be atmost $n^2$ synapses of the same type between $n$ different neurons. Each synapse will have their own state variables the dynamics of which can be defined using differential equations.

For most networks, not all neurons will be connected by all types of synapses. The network of neurons with have a certain connectivity that can be represented as a adjacency matrix in the language of graphs. Each type of synapse will have its own connectivity/adjacency matrix.

### Types of Synapses

There are two major categories of synapses: Exitatory and Inhibitory Synapses.

Here we will focus on implementing one of each category of synapses, Excitatory and Inhibitory.

### Modelling Synapses

The current that passes through a synaptic channel depends on the difference between its reversal potential ($E_{syn}$) and the actual value of the membrane potential ($u$), and is $I_{syn}(t) = g_{syn}(t)(u(t) - E_{syn})$. We can describe the synaptic conductance $g_{syn}(t) = g_{max}[O](t)$, by its maximal conductance $g_{max}$ and a gating variable $[O]$, where $[O](t)$ is the fraction of open synaptic channels. Channels open when a neurotransmitter binds to the synapse which is a function of the presynaptic activation $[T]$.

$$\frac{d[O]}{dt} = \alpha[T](1 - [O]) - \beta[O] \tag{36}$$

where $\alpha$ is the binding constant, $\beta$ the unbinding constant and $(1 - [O])$ the fraction of closed channels where binding of neurotransmitter can occur. The functional form of T depends on the type and nature of the synapse.

### Acetylcholine Synapses (Excitatory)

$$[T]_{ach} = A\ \Theta(t_{max} + t_{fire} + t_{delay} - t)\ \Theta(t - t_{fire} - t_{delay}) \tag{37}$$

### GABAa Synapses (Inhibitory)

$$[T]_{gaba} = \frac{1}{1 + e^{-\frac{V(t) - V_0}{\sigma}}} \tag{38}$$

## Iterating over conditionals in TensorFlow

How would you solve a problem where you have to choose between two options based on the condition provided in a list/array using TensorFlow. Say, you have a array of 10 random variables (say x) between 0 and 1, and you want the output of your code to be 10, if the random variable is greater than 0.5, but -10 when it is not. To perform choices based on the conditions given in a boolean list/array, we can use the TensorFlow function tf.where(). tf.where(cond,a,b) chooses elements from a/b based on conditional array/list cond. Essentially it performs masking between a and b.

```
1   # create the Tensor with the random variables
2   x = tf.constant(np.random.uniform(size = (10,)),dtype=tf.float64)
3   # a list of 10s to select from if true
4   if_true = tf.constant(10*np.ones((10,)),dtype=tf.float64)
5   # a list of -10s to select from if false
6   if_false = tf.constant(-10*np.ones((10,)),dtype=tf.float64)
7   # perform the conditional masking
8   selection = tf.where(tf.greater(x,0.5),if_true,if_false)
9   with tf.Session() as sess:
10      x_out = sess.run(x)
11      selection_out = sess.run(selection)
12  # If x_out = [0.13 0.08 0.58 0.17 0.34 0.58 0.97 0.66 0.30 0.29 ],
13  # selection_out = [-10. -10.  10. -10. -10.  10.  10.  10. -10. -10.]
```

## Recalling and Redesigning the Generalized TensorFlow Integrator

Recall the RK4 based numerical integrator we had created on day 2. You might have noticed that there is an additional dynamical state variable required for the implementation of synapses which is cannot be trivially changed by a memoryless differential equation. Here, we use the word memoryless because till now, all our dynamical variables have only depended on the value immediately before. The dynamical state variable in question is the time of last firing, lets call it fire_t.

One limitation of using TensorFlow in this implentation is that, when we are calculating the change in the dynamical state variable, we only have access to the values for variables immediately before unless we explicity save it as a different variable. Note that if we want to check if a neuron has "fired", we need to know the value of the voltage before and after the updation to check if it crossed the threshold. This means we have to change our implementation of the integrator to be able to update the varaible fire_t
We do this as follows:

1. The Integrator needs to know two more properties, the number of neurons ($n$) and firing threshold ($threshold$) for each of these neurons. We provide this information as inputs to the Integrator Class itself as arguments.

2. Our state vector will now have an additional $n$ many variables representing the firing time that will not undergo the standard differential updation but be updated by a single bit memory method.

3. Inside our Integrator, we have access to the initial values of the state variable and the change in the state variable. We use this to check if the voltages have crossed the firing threshold. For this, we need to define a convection for the state vector, we will store the voltage of the neurons in the first $n$ elements and the fire times fire_t in the last $n$ elements.

4. The differential update function ie. step_func will take all variables but not update the last n values ie. $\frac{d\ fire\_t}{dt} = 0$, the updation will be performed by the scan function itself after the $\Delta y$ has been calculated. It will check for every neuron if the firing threshold of that neuron lies between $V$ and $V + \Delta V$ and update the variable fire_t of the appropriate neurons with the current time.

To implement this, we need to change the integrate function and the integration caller.

```
1  def integrate(self, func, y0, t):
2          time_delta_grid = t[1:] - t[:-1]
3          def scan_func(y, t_dt):
4              # recall the necessary variables
5              n_ = self.n_
6              F_b = self.F_b
7              t, dt = t_dt
8              # Differential updation
9              dy = self._step_func(func,t,dt,y) # Make code more modular.
10             dy = tf.cast(dy, dtype=y.dtype) # Failsafe
11             out = y + dy # the result after differential updation
12             # Use specialized Integrator vs Normal Integrator (n=0)
13             if n_>0:
14                 # Extract the last n variables for fire times
15                 fire_t = y[-n_:]
16                 # Change in fire_t if neuron didnt fire = 0
17                 l = tf.zeros(tf.shape(fire_t),dtype=fire_t.dtype)
18                 # Change in fire_t if neuron fired = Current-Last Fire
19                 l_ = t-fire_t
20                 # Check if previous Voltage is less than Threshold
21                 z = tf.less(y[:n_],F_b)
22                 # Check if Voltage is more than Threshold after update
23                 z_ = tf.greater_equal(out[:n_],F_b)
24                 df = tf.where(tf.logical_and(z,z_),l_,l)
25                 fire_t_ = fire_t+df # Update firing time
26                 return tf.concat([out[:-n_],fire_t_],0)
27             else:
28                 return out
29         y = tf.scan(scan_func, (t[:-1], time_delta_grid),y0)
30         return tf.concat([[y0], y], axis=0)
31
32 def odeint(func, y0, t, n_, F_b):
33     t = tf.convert_to_tensor(t, preferred_dtype=tf.float64, name='t')
34     y0 = tf.convert_to_tensor(y0, name='y0')
35     tf_check_type(y0,t)
36     return _Tf_Integrator(n_, F_b).integrate(func,y0,t)
```

## Implementing the Dynamical Function for an Hodkin Huxley Neuron

Recall, each Hodgkin Huxley Neuron in a $n$ network has 4 main dynamical variables V, m, n, h which we represent as $n$ vectors. Now we need to add some more state variables for representing each synapse ie. the fraction of open channels. For each neuron, we will have Eq (29),Eq (30), Eq (31) but Eq (25) will be replaced by:

$$C_m \frac{dV}{dt} = I_{injected} - I_{Na} - I_K - I_L - I_{ach} - I_{gaba} \tag{39}$$

For each synapse, we will have Eq (36), Eq (37) and:

$$\frac{d[O]_{ach/gaba}}{dt} = \alpha(1 - [O]_{ach/gaba})[T]_{ach/gaba} - \beta[O]_{ach/gaba} \tag{40}$$

$$I_{ach/gaba}(t) = g_{max}[O]_{ach/gaba}(V - E_{ach/gaba}) \tag{41}$$

## Synaptic Memory Management

As discussed earlier, there are atmost $n^2$ synapses of each type but at a time, unless the network is fully connected/very dense, mostly we need a very small subset of these synapses. We could, in principle, calculate the dynamics of all $n^2$ but it would be pointless. So have to devise a matrix based sparse-dense coding system for evaluating the dynamics of these variables and also using their values. This will reduce memory usage and minimize number of calculations at the cost of time for encoding and decoding into dense data from sparse data and vice versa. This is why we use a matrix approach, so that tensorflow can speed up the process.

## Defining the Connectivity

Lets take a very simple 3 neuron network to test how the two types of synapses work. Let $X_1$ be an Excitatory Neuron and it forms Acetylcholine Synapses, $X_2$ be a Inhibitory Neuron and it forms GABAa Synapses and $X_3$ be an output neuron that doesn't synapse onto any neurons. Take the network of the form: $X_1 \to X_2 \to X_3$.

We create the connectivity matrices for both types of synapses. We need to define a convention for the ordering of connectivity. We set the presynaptic neurons as the column number, and the postsynaptic neurons as the row number.

Let $X_1, X_2, X_3$ be indexed by 0, 1 and 2 respectively. The Acetylcholine connectivity matrix takes the form of:

$$Ach_{n \times n} = \begin{bmatrix} 0 & 0 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix} \tag{42}$$

Similarly, the GABAa connectivity matrix becomes:

$$GABA_{n \times n} = \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 1 & 0 \end{bmatrix} \tag{43}$$

```
1   n_n = 3 # number of simultaneous neurons to simulate
2   # Acetylcholine
3   ach_mat = np.zeros((n_n,n_n))          # Ach Synapse Connectivity Matrix
4   ach_mat[1,0]=1
5   ## Parameters for Acetylcholine synapses ##
6   n_ach = int(np.sum(ach_mat))      # Number of Acetylcholine (Ach) Synapses
7   alp_ach = [10.0]*n_ach            # Alpha for Ach Synapse
8   bet_ach = [0.2]*n_ach             # Beta for Ach Synapse
9   t_max = 0.3                       # Maximum Time for Synapse
10  t_delay = 0                       # Axonal Transmission Delay
11  A = [0.5]*n_n                     # Synaptic Response Strength
12  g_ach = [0.35]*n_n                # Ach Conductance
13  E_ach = [0.0]*n_n                 # Ach Potential
14  # GABAa
15  gaba_mat = np.zeros((n_n,n_n))         # GABAa Synapse Connectivity Matrix
16  gaba_mat[2,1] = 1
```

```
17   ## Parameters for GABAa synapses ##
18   n_gaba = int(np.sum(gaba_mat)) # Number of GABAa Synapses
19   alp_gaba = [10.0]*n_gaba       # Alpha for GABAa Synapse
20   bet_gaba = [0.16]*n_gaba       # Beta for GABAa Synapse
21   V0 = [-20.0]*n_n                      # Decay Potential
22   sigma = [1.5]*n_n                     # Decay Time Constant
23   g_gaba = [0.8]*n_n                    # fGABA Conductance
24   E_gaba = [-70.0]*n_n                  # fGABA Potential
25   ## Storing Firing Thresholds ##
26   F_b = [0.0]*n_n                       # Fire threshold
27   ## Store our input to each neuron as a n x timesteps matrix
28   ## called current_input, and extract value at each timepoint
29   def I_inj_t(t):
30       # Turn indices to integer and extract from matrix
31       index = tf.cast(t/epsilon,tf.int32)
32       return tf.constant(current_input.T,dtype=tf.float64)[index]
```

## Working with Sparse-Dense Dynamics

For performing the dynamical updates for the synapses, we need only as many variables as the number of synapse x number of equations required for each synapse. Here our synapse models require only one dynamical variable ie. open fraction [O] which we will store as an k-vector where k is the number of synapses.

We will need to work with this [O] vector on two seperate instances and each time we will have to go to a dense form to speed up computation.

## A. Calculation of Synaptic Currents

The formula for $I_{syn}$ is given by:

$$I_{syn} = \sum_{presynaptic} g_{syn}[O](V - E_{syn}) \tag{44}$$

The best way to represent this calculation is to use the connectivity matrix $\mathbf{C}$ for the synapses and the open fraction vector $[\vec{O}]$ to create an open fraction matrix $\mathbf{O}$ and perform the following computations.

$$\mathbf{C} = \begin{bmatrix} 0 & 1 & ... & 0 \\ 0 & 0 & ... & 1 \\ ... & ... & ... & 1 \\ 1 & 0 & 0 & 0 \end{bmatrix} \tag{45}$$

$$[\vec{O}] = [O_1, O_2...O_k] \tag{46}$$

$$\mathbf{O} = \begin{bmatrix} 0 & O_1 & ... & 0 \\ 0 & 0 & ... & O_a \\ ... & ... & ... & O_b \\ O_k & 0 & 0 & 0 \end{bmatrix} \tag{47}$$

$$[\vec{I_{syn}}] = \sum_{columns} \mathbf{O} \diamond (\vec{g}_{syn} \odot (\vec{V} - \vec{E}_{syn})) \tag{48}$$

where $\diamond$ is columnwise multiplication and $\odot$ is elementwise multiplication. $[\vec{I_{syn}}]$ is now the total synaptic current input to the each of the neurons.

**Algorithm for Synaptic Currents**

1. Firstly we need to convert from the sparse $[O]$ vector to the dense $\mathbf{O}$ matrix. TensorFlow does not allow to make changes to a defined tensor directly, thus we create a $n^2$ vector TensorFlow variable o_ which we will later reshape to a $n \times n$ matrix.

2. We then flatten the synaptic connectivity matrix and find the indices where there is a connection. For this we use the boolean mask function to choose the correct k (total number of synapses) indices from the range 1 to $n^2$ and store in the variable ind.

3. Using the scatter_update function of TensorFlow, we fill the correct indices of the variable o_ that we created with the values of open fraction from the $[O]$ vector.

4. We now reshape the vector as a $n \times n$ matrix. Since python stores matrices as array of arrays, with each row as an inner array, for performing columnswise multiplication, the easiest way is to tranpose the matrix, so that each column is the inner array, perform element wise multiplication with each inner array and apply transpose again.

5. Finally using reduce_sum, we sum over the columns to get our $I_{syn}$ vector.

```python
## Acetylcholine Synaptic Current ##
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)
## GABAa Synaptic Current ##
def I_gaba(o,V):
    o_ = tf.Variable([0.0]*n_n**2,dtype=tf.float64)
    ind = tf.boolean_mask(tf.range(n_n**2),gaba_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_gaba))*g_gaba),1)
## Other Currents remain the same ##
```

## B. Updation of Synaptic Variables

For the updation, the first we need to calculate the values of the presynaptic activation [T] for both types of synapses. We will essentially calculate this for each neuron and then redirect the values to the correct post synaptic neuron. Recall:

$$[T]_{ach} = A\ \Theta(t_{max} + t_{fire} + t_{delay} - t)\ \Theta(t - t_{fire} - t_{delay}) \tag{49}$$

$$[T]_{gaba} = \frac{1}{1 + e^{-\frac{V(t)-V_0}{\sigma}}} \tag{50}$$

Thus $[T]_{ach}$ is function of the last firing time, and $[T]_{gaba}$ depends on the presynaptic voltage. Once we calculate the values of [T]-vector for both types of synapse, we need to redirect them to the correct synapses in a sparse $k \times 1$ vector form.

**Algorithm for Dynamics**

1. For $[T]_{ach}$, use a boolean logical_and function to check is the current timepoint t is greater than the last fire time (fire_t) + delay (t_delay) and less than last fire time (fire_t) + delay (t_delay) + activation length (t_max) for each neuron as a vector. Use the result of these boolean operations to choose between zero or an constant A. This serves as the heaviside step function. For $[T]_{gaba}$, simply use the V vector to determine T.

2. For making the sparse vector, we follow as two step process. First we multiply each row of the connectivity matrices **C** with the respective $[T]$ vector to get a activation matrix **T**, and then we just flatten **T** and **C** and, using tf.boolean_mask, remove all the zeros from **T** to get a $k \times 1$ vector which now stores the presynaptic activation for each of the synapses where $k = n_{gaba}$ or $n_{ach}$.

3. Calculate the differential change in the open fractions (OF) using the $k \times 1$ vector.

```python
def dXdt(X, t):
    V = X[:1*n_n]          # First n_n: Membrane Voltage
    m = X[1*n_n:2*n_n]     # Next n_n: Sodium Activation Gating
    h = X[2*n_n:3*n_n]     # Next n_n: Sodium Inactivation Gating
    n = X[3*n_n:4*n_n]     # Next n_n: Potassium Gating
    # Next n_ach and n_gaba: Ach and GABAa Open Fraction respectively
    o_ach = X[4*n_n : 4*n_n + n_ach]
    o_gaba = X[4*n_n + n_ach : 4*n_n + n_ach + n_gaba]
    fire_t = X[-n_n:]      # Last n_n: last fire times
    dVdt = (I_inj_t(t)-I_Na(V, m, h)-I_K(V, n)-
                    I_L(V)-I_ach(o_ach,V)-I_gaba(o_gaba,V))/C_m
    ## Updation for gating variables ##
    m0,tm,h0,th = Na_prop(V)
    n0,tn = K_prop(V)
    dmdt = - (1.0/tm)*(m-m0)
    dhdt = - (1.0/th)*(h-h0)
    dndt = - (1.0/tn)*(n-n0)
    ## Updation for o_ach ##
    A_ = tf.constant(A,dtype=tf.float64)
    Z_ = tf.zeros(tf.shape(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_,Z_)
    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
    ## Updation for o_gaba ##
    T_gaba = 1.0/(1.0+tf.exp(-(V-V0)/sigma))
    T_gaba = tf.multiply(tf.constant(gaba_mat,dtype=tf.float64),T_gaba)
```

```
30    T_gaba = tf.boolean_mask(tf.reshape(T_gaba,(-1,)),
31                             gaba_mat.reshape(-1) == 1)
32    do_gabadt = alp_gaba*(1.0-o_gaba)*T_gaba - bet_gaba*o_gaba
33    ## Updation for fire times ##
34    dfdt = tf.zeros(tf.shape(fire_t),dtype=fire_t.dtype) # no change
35    out = tf.concat([dVdt,dmdt,dhdt,dndt,do_achdt,do_gabadt,dfdt],0)
36    return out
```

## Defining the Gating Variable Updation Function and the Initial Conditions

Like earlier, we again define the function that returns us the values of $\tau_m$, $\tau_h$, $\tau_n$, $m_0$, $h_0$, $n_0$ and we prepare the parameters and initial conditions.

   Note: If we initialize the last firing time as 0, then the second neuron $X_2$ will get an EPSP immediately after the start of the simulation. To avoid this the last firing time should be initialized to a large negative number ¿= the length of the simulation.

```
1    # The initialization of the Parameters and Gating Variable
2    # Updation Function remain the same.
3
4    # Initialize the State Vector
5    y0 = tf.constant([-71]*n_n+[0,0,0]*n_n+[0]*n_ach+[0]*n_gaba+
6                             [-9999999]*n_n,dtype=tf.float64)
```

## Creating the Current Input and Run the Simulation

We will run an 700 ms simulation with 100ms current injection at neuron $X_1$ of increasing amplitude with 100ms gaps in between.

```
1    current_input= np.zeros((n_n,t.shape[0]))
2    current_input[0,int(100/epsilon):int(200/epsilon)] = 2.5
3    current_input[0,int(300/epsilon):int(400/epsilon)] = 5.0
4    current_input[0,int(500/epsilon):int(600/epsilon)] = 7.5
5    state = odeint(dXdt,y0,t,n_n,F_b)
6    with tf.Session() as sess:
7        # Since we are using variables we have to initialize them
8        tf.global_variables_initializer().run()
9        state = sess.run(state)
```

   The output is plotted in Fig.

   We can see that the current injection triggers the firing of action potentials with increasing frequency with increasing current. Also we see that as soon as the first neuron $X_1$ crosses its fireing threshold, an EPSP is triggered in the next neuron $X_2$ causing a firing with a slight delay from the firing of $X_1$. Finally, as the second neuron depolarizes, we see a corresponding hyperpolarization in the next neuron $X_3$ caused by an IPSP. We can also plot the dynamics of the channels itself by plotting o_ach and o_gaba which are the 5th and 4th last elements respectively (Fig). Thus we are now capable of making complex networks of neurons with both excitatory and inhibitory connections.

# Day 5: Optimal Mind Control

Now that we can simulate a model of a network of conductance-based neurons, we discuss the limitations of our algorithm and try to find solutions to the problems.

## Memory Management

This TensorFlow implementation allows us to make our simulation code not only easier to read but also makes it highly parallizable and scalable across a variety of computational devices. But there are some major limitations to our system. The biggest of these issues is that despite making the simulation faster, this implementation makes it very memory intensive.

The iterators in TensorFlow do not follow the normal process of Memory Allocation and Garbage Collection. Since, TensorFlow is designed to work on sophisticated hardware like GPUs, TPUs and distributed platforms, the memory allocation is done during TensorFlow session adaptively, but the memory is NOT cleared until the python kernel has stopped execution.

The memory used increases linearly with time as the state matrix is computed recursively by the Scan function. The maximum memory used by the computational graph is 2 times the total state matrix size at the point when the computation finishes and copies the final data into the memory. Larger and more complicated the network and longer the simulation, larger the matrix and then, each run is limited by the total available memory. As we increase the number of neurons (n) and scale the degree distributions, the size of the memory used grows $O(n^2)$ as the number of differential equations grows as a square of the number of neurons. Similarly, as we increase the length of the simulation(L), the memory used grows $O(L)$. Which is why for a system with a limited memory of K bytes, The length of a given simulation (L timesteps) of a given network (N differential equations) with 64-bit floating-point precision will follow:

$$2 \times 64 \times L \times N = K \tag{51}$$

That is, for any given network, our maximum simulation length is limited. One way to improve our maximum length is to divide the simulation into smaller batches. There will be a small queuing time between batches, which will slow down our code by a small amount but we will be able to simulate longer times. Thus, if we split the simulation into K sequential batches, the maximum memory for the simulation becomes $(1 + \frac{1}{K})$ times the total matrix size. Thus the memory relation becomes:

$$\left(1 + \frac{1}{K}\right) \times 64 \times L \times N = K \tag{52}$$

This way, we can maximize the length of out simulation that we can run in a single python kernel.

Let us implement this batch system for our 3 neuron feed-forward model.

## Implementing the Model

To improve out readability and make our code more modular we seperate the integrator into a independent import module. Take the integrator code that we developed in the last day and place it in a file called "tf_integrator.py". Make sure the file is present in the same directory as the implementation of the model.

Note: If you are using Jupyter Notebook, remember to remove the %matplotlib inline command as it is specific to jupyter.

### Importing tf_integrator and other requirements

Once the Integrator is saved in tf_integrator.py in the same directory as the Notebook, we can start importing the essentials including the integrator.

```
1  import tensorflow as tf
2  import numpy as np
3  import tf_integrator as tf_int
4  import matplotlib.pyplot as plt
5  import seaborn as sns
```

For implementing a Batch system, we do not need to change how we construct our model only how we execute it.

**Splitting Time Series into independent batches and Run Each Batch Sequentially**

Since we will be dividing the computation into batches, we have to split the time array into batches which will be passed to the each successive call to the integrator. For each new call, the last state vector of the last call will be the new initial condition.

In np.array_split(), the split edges are present in only one array and since our initial vector to successive calls is corresponding to the last output our first element in the later time array should be the last element of the previous output series. Thus, we append the last time to the beginning of the current time array batch.

```
1  # Define the Number of Batches
2  n_batch = 2
3  # Split t array into batches using numpy
4  t_batch = np.array_split(t,n_batch)
5  # Iterate over the batches of time array
6  for n,i in enumerate(t_batch):
7      # Inform start of Batch Computation
8      print("Batch",(n+1),"Running...",end="")
9      # Re-adjusting edges
10     if n>0:
11         i = np.append(i[0]-sim_res,i)
12     # Set state_vector as the initial condition
13     init_state = tf.constant(state_vector, dtype=tf.float64)
14     # Create the Integrator computation graph
15     tensor_state = tf_int.odeint(dXdt, init_state, i, n_n, F_b)
16     # Initialize variables and run session
17     with tf.Session() as sess:
18         tf.global_variables_initializer().run()
19         state = sess.run(tensor_state)
20         sess.close()
21     # Reset state_vector as the last element of output
22     state_vector = state[-1,:]
23     # Save the output of the simulation to a binary file
24     np.save("part_"+str(n+1),state)
25     # Clear output
26     state=None
27     print("Finished")
```

**Putting the Output Together**

The output from our batch implementation is a set of binary files that store subparts of our total simulation. To get the overall output we have to stitch them back together.

```
1   overall_state = []
2   # Iterate over the generated output files
3   for n,i in enumerate(["part_"+str(n+1)+".npy" for n in range(n_batch)]):
4       # Since the first element in the series was the last output,
5       # we remove them
6       if n>0:
7           overall_state.append(np.load(i)[1:,:])
8       else:
9           overall_state.append(np.load(i))
10  # Concatenate all the matrix to get a single state matrix
11  overall_state = np.concatenate(overall_state)
```

By this method, we have maximized the usage of our available memory but we can 530
go further and develop a method to allow indefinitely long simulation. The issue behind 531
this entire algorithm is that the memory is not cleared until the python kernel finishes. 532
One way to overcome this is to save the parameters of the model (such as connectivity 533
matrix) and the state vector in a file, and start a new python kernel from a python 534
script to compute successive batches. This way after each large batch, the memory gets 535
cleaned. By combining the previous batch implementation and this system, we can 536
maximize our computability. 537

### Implementing a Runner and a Caller 538

Firstly, we have to create an implementation of the model that takes in previous input 539
as current parameters. Thus, we create a file, which we call "run.py" that takes an 540
argument ie. the current batch number. The implementation for "run.py" is mostly 541
same as the above model but there is a small difference. 542
When the batch number is 0, we initialize all variable parameters and save them, but 543
otherwise we use the saved values. The parameters we save include: Acetylcholine 544
Matrix, GABAa Matrix and Final/Initial State Vector. It will also save the files with 545
both batch number and sub-batch number listed. 546
The time series will be created and split initially by the caller, which we call 547
"call.py", and stored in a file. Each execution of the Runner will extract its relevant 548
time series and compute on it. 549

#### Implementing the Caller code 550

The caller will create the time series, split it and use python subprocess module to call 551
"run.py" with appropriate arguments. The code for "call.py" is given below. 552

```
1   from subprocess import call
2   import numpy as np
3   total_time = 1000
4   n_splits = 2
5   time = np.split(np.arange(0,total_time,0.01),n_splits)
6   # Append the last time point to the beginning of the next batch
7   for n,i in enumerate(time):
8       if n>0:
9           time[n] = np.append(i[0]-0.01,i)
10  np.save("time",time)
11  # call successive batches with a new python subprocess
12  # and pass the batch number
13  for i in range(n_splits):
```

```
14      call(['python','run.py',str(i)])
15  print("Simulation Completed.")
```

**Implementing the Runner code**

"run.py" is essentially identical to the batch-implemented model we developed earlier
with the changes described below:

```
1   # Additional Imports #
2   import sys
3   # Duration of Simulation #
4   # Replace t = np.arange(0,sim_time,sim_res) by
5   t = np.load("time.npy")[int(sys.argv[1])] # get first argument to run.py
6   # Connectivity Matrix Definitions #
7   if sys.argv[1] == '0':
8       ach_mat = np.zeros((n_n,n_n)) # Ach Synapse Connectivity Matrix
9       ach_mat[1,0]=1
10      # If connectivity is random, once initialized it will be the same.
11      np.save("ach_mat",ach_mat)
12  else:
13      ach_mat = np.load("ach_mat.npy")
14  if sys.argv[1] == '0':
15      gaba_mat = np.zeros((n_n,n_n)) # GABAa Synapse Connectivity Matrix
16      gaba_mat[2,1] = 1
17      # If connectivity is random, once initialized it will be the same.
18      np.save("gaba_mat",gaba_mat)
19  else:
20      gaba_mat = np.load("gaba_mat.npy")
21  # Current Input Definition #
22  if sys.argv[1] == '0':
23      current_input= np.zeros((n_n,int(sim_time/sim_res)))
24      current_input[0,int(100/sim_res):int(200/sim_res)] = 2.5
25      current_input[0,int(300/sim_res):int(400/sim_res)] = 5.0
26      current_input[0,int(500/sim_res):int(600/sim_res)] = 7.5
27      np.save("current_input",current_input)
28  else:
29      current_input = np.load("current_input.npy")
30  # State Vector Definition #
31  if sys.argv[1] == '0':
32      state_vector = [-71]*n_n+[0,0,0]*n_n+[0]*n_ach+[0]*n_gaba
33                                          +[-9999999]*n_n
34      state_vector = np.array(state_vector)
35      state_vector = state_vector + 0.01*state_vector
36                              *np.random.normal(size=state_vector.shape)
37      np.save("state_vector",state_vector)
38  else:
39      state_vector = np.load("state_vector.npy")
40  # Saving of Output #
41  # Replace np.save("part_"+str(n+1),state) by
42  np.save("batch"+str(int(sys.argv[1])+1)+"_part_"+str(n+1),state)
```

**Combining all Data**

Just like we merged all the batches, we merge all the sub-batches and batches.

```
1   overall_state = []
2   # Iterate over the generated output files
3   for n,i in enumerate(["batch"+str(x+1) for x in range(n_splits)]):
4       for m,j in enumerate(["_part_"+str(x+1) for x in range(n_batch)]):
5           # Since the first element in the series was the last output,
6           # we remove them
7           if n>0 and m>0:
8               overall_state.append(np.load(i+j+".npy")[1:,:])
9           else:
10              overall_state.append(np.load(i+j+".npy"))
11  # Concatenate all the matrix to get a single state matrix
12  overall_state = np.concatenate(overall_state)
```

## Other Limitations 558

With the runner-caller implementation, we have solved the issue of memory limitation. 559
But there are some issues we havent solved yet. 560
    1. **Delay Dependent Activity:** If the implementation of an model requires a 561
delay differential equation (DDEs) that accesses the value of the parameters at a earlier 562
time, we cannot use this programming paradigm directly as it has only, at most, one 563
timestep memory. This is also a limitation of using the scan function. Exploring 564
alternatives to tf.scan() and other techniques of solving DDEs is an option. 565
    2. **Optimal Distributed Computing:** To maximize the utility of distributed 566
systems, the algorithm needs to be capable of splitting the computation into tasks that 567
can be run parallely (simultaneously) in different computational units. Since we are 568
working with numerical integration, the results of the last step need to be known to 569
perform the next computation. This, in a way limits out ability to use distributed 570
computing. TensorFlow automatically tries to utilize all available resources efficiently 571
but it may not be the optimal usage. One way to enforce distribution is to manually 572
distribute parts of the computational graph to specialized computing units. For 573
example, the actual session can be forced to be executed on the server with maximum 574
memory but the sub-computations such as the RK4 integration steps can be distributed 575
to servers with more GPUs. 576

# Discussion 577

# Conclusion 578

# Supporting information 579

**S1 Fig.**   **Bold the title sentence.** Add descriptive text after the title of the item 580
(optional). 581

# Acknowledgments 582

# References

    1. Conant GC, Wolfe KH. Turning a hobby into a job: how duplicated genes find
       new functions. Nat Rev Genet. 2008 Dec;9(12):938–950.

2. Ohno S. Evolution by gene duplication. London: George Alien & Unwin Ltd. Berlin, Heidelberg and New York: Springer-Verlag.; 1970.

3. Magwire MM, Bayer F, Webster CL, Cao C, Jiggins FM. Successive increases in the resistance of Drosophila to viral infection through a transposon insertion followed by a Duplication. PLoS Genet. 2011 Oct;7(10):e1002337.