# 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.

## Requirements for the 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

Our discussion begins with what Numerical Integration is and how we can use it to solve differential equations given the initial condition in Python.

## What is Numerical Integration?

From the point of view of a theoretician, the ideal form of the solution to a differential equation given the initial conditions, i.e. an initial value problem (IVP), would be a formula for the solution function. But sometimes obtaining a formulaic solution is not always easy, and in many cases is absolutely impossible. So, what do we do when faced with a differential equation that we cannot solve? If you are only looking for long term behavior of a solution you can always sketch a direction field. This can be done without too much difficulty for some fairly complex differential equations that we can't solve to get exact solutions. But, what if we need to determine how a specific solution behaves, including some values that the solution will take? In that case, we have to rely on numerical methods for solving the IVP such as euler's method or the Runge-Kutta Methods.

## Euler's Method for Numerical Integration

We use Euler's Method to generate a numerical solution to an initial value problem of the form:

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

$$x(t_o) = x_o \tag{2}$$

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:

$$t_{n+1} = t_n + \epsilon \tag{3}$$

$$x_{n+1} = x_n + \epsilon f(x_n, t_n) \tag{4}$$

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 1.

## 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 $x(t) = e^{5t}$.

```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 2.

## Euler and Vectors

Euler's Method also applies to vectors and can solve simultaneous differential equations.
The Initial Value problem now becomes:

$$\frac{d\vec{X}}{dt} = \vec{f}(\vec{X}, t) \tag{5}$$

$$\vec{X}(t_o) = \vec{X}_o \tag{6}$$

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

$$t_{n+1} = t_n + \epsilon \tag{7}$$

$$\vec{X_{n+1}} = \vec{X}_n + \epsilon\vec{f}(\vec{X}_n, t_n) \tag{8}$$

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 $f(\vec{X}, t) = [x - y, y - x]$.

```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

Now, 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 : $\epsilon \rightarrow \epsilon(t_n)$.
- 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:

$$k_1 = \epsilon f(x_n, t_n) \tag{9}$$

$$k_2 = \epsilon f(x_n + \frac{k_1}{2}, t_n + \frac{\epsilon}{2}) \tag{10}$$

$$y_{n+1} = y_n + k_2 + O(\epsilon^3) \tag{11}$$

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:

$$k_1 = f(x_n, t_n) \tag{12}$$

$$k_2 = f(x_n + \epsilon\frac{k_1}{2}, t_n + \frac{\epsilon}{2}) \tag{13}$$

$$k_3 = f(x_n + \epsilon\frac{k_2}{2}, t_n + \frac{\epsilon}{2}) \tag{14}$$

$$k_4 = f(x_n + \epsilon k_3, t_n + \epsilon) \tag{15}$$

$$y_{n+1} = y_n + \frac{\epsilon}{6}(k_1 + 2k_2 + 2k_3 + k_4) + O(\epsilon^5) \tag{16}$$

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.

```python
# 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:

$$\frac{d^2s}{dt^2} = L\frac{d^2\theta}{dt^2} = -g\sin\theta \tag{17}$$

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:

$$\frac{d\theta}{dt} = \omega \tag{18}$$

$$\frac{d\omega}{dt} = -\frac{g}{L}\sin\theta \tag{19}$$

## Day 2: Let the Tensors Flow!

We start with our discussion with an introduction to TensorFlow followed by implementation of Numerical Integration techniques in TensorFlow.

### An Introduction to TensorFlow

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

Essentially, TensorFlow 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. 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 (Fig):

- 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.

## How TensorFlow works?

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

## Implementing a computation graph in TensorFlow

```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)
# TO automatically close the session after computation, Use:
# with tf.Session() as sess:
#    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.

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

```python
def tf_check_type(t, y0): # Ensure Input is Correct
    if not (y0.dtype.is_floating and t.dtype.is_floating):
        # The datatype of any tensor t is accessed by t.dtype
        raise TypeError('Error in Datatype')
class _Tf_Integrator():
    def integrate(self, func, y0, t):
        time_delta_grid = t[1:] - t[:-1]
        def scan_func(y, t_dt):
            t, dt = t_dt
            dy = dt*func(y,t)
            return y + dy
        # iterating over (a,b) where a and b are lists of same size
        # results in the ith accumulative step in tf.scan receiving
        # the ith elements of a and b zipped together
        y = tf.scan(scan_func, (t[:-1], time_delta_grid),y0)
        return tf.concat([[y0], y], axis=0)
def tf_odeint_euler(func, y0, t):
    # Convert input to TensorFlow Objects
    t = tf.convert_to_tensor(t, preferred_dtype=tf.float64, name='t')
    y0 = tf.convert_to_tensor(y0, name='y0')
    tf_check_type(y0,t)
    return _Tf_Integrator().integrate(func,y0,t)

# Define a function using Tensorflow math operations.
# This creates the computation graph.
def f(X,t):
    # extracting a single value eg. X[0] returns a single value but
    # we require a tensor, so we extract a range with one element.
    x = X[0:1]
    y = X[1:2]
    out = tf.concat([x-y,y-x],0)
    return out
y0 = tf.constant([1,0], dtype=tf.float64)
epsilon = 0.01
t = np.arange(0,2,epsilon)
# Define the final value (output of scan) that we wish to compute
state = tf_odeint_euler(f,y0,t)
# Start a TF session and evaluate state
with tf.Session() as sess:
    state = sess.run(state)
```

## RK4 Integration Function in TensorFlow 183

Now, we implement the exact same RK4 integrator in TensorFlow, thus we make a small 184
change in the integrate function. Also to make the code more modular, we introduce 185
the differential step calculator to a different function, everything else remains the same. 186

```
def integrate(self, func, y0, t):
        time_delta_grid = t[1:] - t[:-1]
        def scan_func(y, t_dt):
            t, dt = t_dt
            dy = self._step_func(func,t,dt,y) # Make code more modular.
            return y + dy
        y = tf.scan(scan_func, (t[:-1], time_delta_grid),y0)
        return tf.concat([[y0], y], axis=0)
    def _step_func(self, func, t, dt, y):
        k1 = func(y, t)
        half_step = t + dt / 2
        dt_cast = tf.cast(dt, y.dtype) # Failsafe
        k2 = func(y + dt_cast * k1 / 2, half_step)
        k3 = func(y + dt_cast * k2 / 2, half_step)
        k4 = func(y + dt_cast * k3, t + dt)
        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 187
Method and RK4 on TensorFlow which is given by the equations: 188

$$\frac{dx}{dt} = \sigma(y - x) \tag{20}$$

189

$$\frac{dy}{dt} = x(\rho - z) - y \tag{21}$$

190

$$\frac{dz}{dt} = xy - \beta z \tag{22}$$

Use the values $\sigma = 10$, $\beta = \frac{8}{3}$, $\rho = 28$. You can try simulating this system at two 191
nearby starting conditions and comment on the difference. 192

## 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:

$$C_m \frac{du}{dt} = -I_{Na}(t) - I_K(t) - I_L(t) + I(t) \tag{23}$$

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:

$$I_{Na} = g_{Na} m^3 h (u - E_{Na}) \tag{24}$$

$$I_K = g_K n^4 (u - E_K) \tag{25}$$

$$I_L = g_L (u - E_L) \tag{26}$$

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:

$$\frac{dm}{dt} = -\frac{1}{\tau_m}(m - m_0) \tag{27}$$

$$\frac{dh}{dt} = -\frac{1}{\tau_h}(h - h_0) \tag{28}$$

$$\frac{dn}{dt} = -\frac{1}{\tau_n}(n - n_0) \tag{29}$$

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 (23), Eq (27),Eq (28) and Eq (29) 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 (24),Eq (25) and Eq (26).

```python
# 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
# Note: Always use TensorFlow functions for all operations.
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_0 = alpha_n/(alpha_n+beta_n)
    return n_0, 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_0 = alpha_m/(alpha_m+beta_m)
    h_0 = alpha_h/(alpha_h+beta_h)
    return m_0, t_m, h_0, t_h
# Step 3: Defining function that calculate Neuronal currents
def I_K(V, n):
    return g_K  * n**4 * (V - E_K)
def I_Na(V, m, h):
    return g_Na * m**3 * h * (V - E_Na)
def I_L(V):
    return g_L * (V - E_L)
# Step 4: Define the function dX/dt where X is the State Vector
def dXdt(X, t):
    V = X[0:1]
    m = X[1:2]
    h = X[2:3]
    n = X[3:4]
    dVdt = (5 - I_Na(V, m, h) - I_K(V, n) - I_L(V)) / C_m
    # Here the current injection I_injected = 5 uA
    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
# Step 5: Define Initial Condition and Integrate
y0 = tf.constant([-71,0,0,0], dtype=tf.float64)
epsilon = 0.01
t = np.arange(0,200,epsilon)
state = odeint(dXdt,y0,t)
with tf.Session() as sess:
    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:

$$\frac{d\vec{X}}{dt} = [f_1(\vec{X}, C_m, g_K, ...E_L), f_2(\vec{X}, C_m, g_K, ...E_L)...f_m(\vec{X}, C_m, g_K, ...E_L)] \tag{30}$$

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:

$$\frac{dV_i}{dt} = f(V_i, m_i, h_i, n_i, C_{m_i}, g_{K_i}...) \tag{31}$$

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

$$\frac{d\vec{V}}{dt} = f(\vec{V}, \vec{m}, \vec{h}, \vec{n}, \vec{C_m}, \vec{g_K}...) \tag{32}$$

Thus we can do the calculations as:

$$\frac{d\mathbf{X'}}{dt} = \left[\frac{d\vec{V}}{dt}, \frac{d\vec{m}}{dt}, \frac{d\vec{h}}{dt}, \frac{d\vec{n}}{dt}\right] \tag{33}$$

**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.

```python
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 $[T]$.

$$\frac{d[O]}{dt} = \alpha[T](1 - [O]) - \beta[O] \tag{34}$$

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)**

$$[T]_{ach} = A\ \Theta(t_{max} + t_{fire} + t_{delay} - t)\ \Theta(t - t_{fire} - t_{delay}) \tag{35}$$

**GABAa Synapses (Inhibitory)**

$$[T]_{gaba} = \frac{1}{1 + e^{-\frac{V(t)-V_0}{\sigma}}} \tag{36}$$

## 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.

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

```python
def integrate(self, func, y0, t):
        time_delta_grid = t[1:] - t[:-1]
        def scan_func(y, t_dt):
            # recall the necessary variables
            n_ = self.n_
            F_b = self.F_b
            t, dt = t_dt
            # Differential updation
            dy = self._step_func(func,t,dt,y) # Make code more modular.
            dy = tf.cast(dy, dtype=y.dtype) # Failsafe
            out = y + dy # the result after differential updation
            # Use specialized Integrator vs Normal Integrator (n=0)
            if n_>0:
                # Extract the last n variables for fire times
                fire_t = y[-n_:]
                # Change in fire_t if neuron didnt fire = 0
                l = tf.zeros(tf.shape(fire_t),dtype=fire_t.dtype)
                # Change in fire_t if neuron fired = Current-Last Fire
                l_ = t-fire_t
                # Check if previous Voltage is less than Threshold
                z = tf.less(y[:n_],F_b)
                # Check if Voltage is more than Threshold after update
                z_ = tf.greater_equal(out[:n_],F_b)
                df = tf.where(tf.logical_and(z,z_),l_,l)
                fire_t_ = fire_t+df # Update firing time
                return tf.concat([out[:-n_],fire_t_],0)
            else:
                return out
        y = tf.scan(scan_func, (t[:-1], time_delta_grid),y0)
        return tf.concat([[y0], y], axis=0)

```

```python
def odeint(func, y0, t, n_, F_b):
    t = tf.convert_to_tensor(t, preferred_dtype=tf.float64, name='t')
    y0 = tf.convert_to_tensor(y0, name='y0')
    tf_check_type(y0,t)
    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 (27),Eq (28), Eq (29) but Eq (23) will be replaced by:

$$C_m \frac{dV}{dt} = I_{injected} - I_{Na} - I_K - I_L - I_{ach} - I_{gaba} \tag{37}$$

For each synapse, we will have Eq (34), Eq (35) and:

$$\frac{d[O]_{ach/gaba}}{dt} = \alpha(1 - [O]_{ach/gaba})[T]_{ach/gaba} - \beta[O]_{ach/gaba} \tag{38}$$

$$I_{ach/gaba}(t) = g_{max}[O]_{ach/gaba}(V - E_{ach/gaba}) \tag{39}$$

## 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: $X_1 \rightarrow X_2 \rightarrow X_3$.

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:

$$Ach_{n\times n} = \begin{bmatrix} 0 & 0 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix} \tag{40}$$

Similarly, the GABAa connectivity matrix becomes: 382

$$GABA_{n\times n} = \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 1 & 0 \end{bmatrix} \tag{41}$$

```python
n_n = 3 # number of simultaneous neurons to simulate
# Acetylcholine
ach_mat = np.zeros((n_n,n_n))        # Ach Synapse Connectivity Matrix
ach_mat[1,0]=1
## PARAMETERS FOR ACETYLCHLOLINE SYNAPSES ##
n_ach = int(np.sum(ach_mat))     # Number of Acetylcholine (Ach) Synapses
alp_ach = [10.0]*n_ach           # Alpha for Ach Synapse
bet_ach = [0.2]*n_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]*n_n                   # Ach Conductance
E_ach = [0.0]*n_n                    # Ach Potential
# GABAa
gaba_mat = np.zeros((n_n,n_n))       # GABAa Synapse Connectivity Matrix
gaba_mat[2,1] = 1
## PARAMETERS FOR GABAa SYNAPSES ##
n_gaba = int(np.sum(gaba_mat)) # Number of GABAa Synapses
alp_gaba = [10.0]*n_gaba       # Alpha for GABAa Synapse
bet_gaba = [0.16]*n_gaba       # Beta for GABAa Synapse
V0 = [-20.0]*n_n                     # Decay Potential
sigma = [1.5]*n_n                    # Decay Time Constant
g_gaba = [0.8]*n_n                 # fGABA Conductance
E_gaba = [-70.0]*n_n               # fGABA Potential
```

(42)

## Discussion 383

## Conclusion 384

## Supporting information 385

**S1 Fig. Bold the title sentence.** Add descriptive text after the title of the item (optional). 386 387

## Acknowledgments 388

Cras egestas velit mauris, eu mollis turpis pellentesque sit amet. Interdum et malesuada fames ac ante ipsum primis in faucibus. Nam id pretium nisi. Sed ac quam id nisi malesuada congue. Sed interdum aliquet augue, at pellentesque quam rhoncus vitae. 389 390 391

## 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.