Imperial College London

Undergraduate Research Opportunities Program

Applying a Gradient Free Sampling-Based Approach For Training Deep Spiking Neural Networks

Supervisor:
Dr Dan Goodman
Department of Electrical and Electronic Engineering

Research Group:
Neural Reckoning Group

September 25, 2020

Contents

1 Introduction

This is a short report summarising a research project within the field of machine learning. The objective was to implement a new optimisation algorithm (in the context of supervised machine learning) when analysing spiking neural networks. The need for this arises because more commonly used techniques such as linear regression are not viable since they require a differentiable loss function, which we cannot obtain from the discrete spiking patterns you get from a neural network. Random Sampling Optimisation, from a paper written by Rohun Tripathi & Bharat Singh [5] , was used instead, leading to promising results.

2 Theory

2.1 Computational Neuroscience

The first task was to research neuroscience, specifically in the context of computation. The following information is derived from a Computational Neuroscience course on coursera [1].

Neuron Basics

Before proceeding it is important to understand how neurons work in general terms. At the very start there are multiple receptive fields within the body, such as light receptors in the retina. These receptive fields have varying shapes that mean that they react differently to different stimuli. An example is a 'on-centre off-surround' receptive field from a retina (See figure 1). This particular receptive field will respond well to a bright point of light surrounded by darkness.

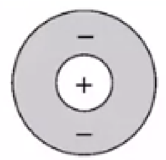

Figure 1: Basic Receptive Field

Then these cells can all feed into further receptive fields to create more complex shapes in later stages (See figure 2). It is clear that any number of shapes can be created by building up receptive fields as shown. In other words we are considering linear combinations of these RFs (i.e $\tilde{I} = \sum_i RF_i r_i$ where $\tilde{I}$ is the recreation of the real image using receptive fields and r is the neural response).The sparse coding constraints are as follows; you must use as few components as possible and you must faithfully represent an image. These constraints allow for us to find an efficient way to represent images.

In other words we are attempting to minimise the difference between the image I and $\tilde{I}$. We want to make I and $\tilde{I}$ as similar as possible while making RF_is as independent as possible. It is interesting to note that is appears that the human body finds components that follow these guides very well.

Now we can see how neurons themselves work (See figure 3). Neurons have multiple inputs (dendrites) feeding into the nucleus, and multiple outputs (axons), that will in turn feed into the inputs of other neurons. Whenever a neuron spikes it releases a burst of electric potential from its axons. If the number of dendrites in a neuron receiving spikes goes above a certain threshold (i.e its electric potential exceeds a certain threshold) the neuron spikes.

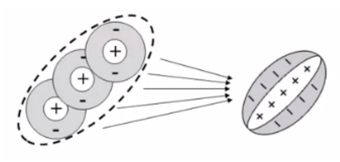

Figure 2: Merging RFs

The final piece of the neuronal puzzle is the way in which they are all interconnected. The connections between neurons are called synapses, and these synapses come in many shapes and sizes (e.g. electrical vs chemical synapses), but they all fundamentally work in the same way. That is, they stimulate the post-synaptic neuron to spike when the pre-synaptic one spikes. The synapse doctrine states that synapses are the basis for learning and memory, and they way in which they achieve this is through synaptic plasticity. Hebbian plasticity states that if a pre-synaptic neuron repeatedly takes part in firing a post-synaptic neuron, then the synapse between them strengthens. This means any future spike in the pre-synaptic neuron will be more likely to cause a spike in the post-synaptic one.

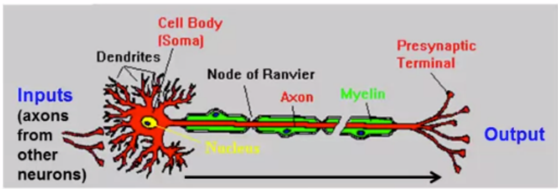

Figure 3: Neuron

The amount by which these weightings change is dictated by the relative timings of the input and output spikes, as shown in figure 4 which is created by the following equations written using the Brian [3] package (shown in A 2) The equations that is being simulated are:

$$\Delta w = \sum_{t_{pre}} \sum_{t_{post}} W(t_{post} - t_{pre})$$

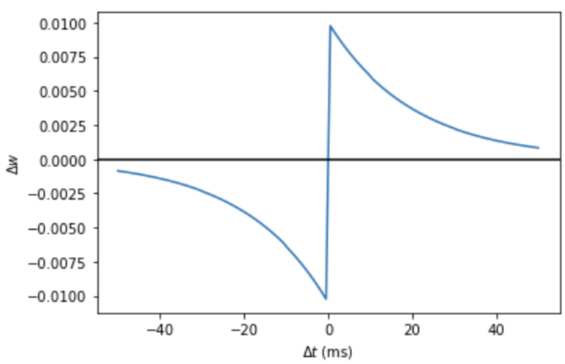

Figure 4: Hebbian Plasticity

$$W(\Delta t) = \begin{cases} A_{pre}e^{-\Delta t/\tau_{pre}} & \Delta t > 0 \\ A_{post}e^{\Delta t/\tau_{post}} & \Delta t < 0 \end{cases}$$

Neural Encoding

This is a slight side-note as it doesn't particularly apply to the project as such. Neural encoding is where you find how a stimulus causes a particular pattern of response, or in other words $p(response|stimulus)$. Essentially you are trying to find the particular pattern in the stimulus that causes a spike. These patterns are known as features, and can be found in a variety of ways. One way is to find all the points in time at which a spike occur, and then note the stimulus a short time before it (see code in A 3). You can then take an average of these stimuli to find the shape of the feature that caused the spikes (see figure 7 in appendix).

Neural Decoding

Neural decoding, on the other hand, is finding what you can about what the response tells us about the stimulus that caused it. There are many methods for doing this, but most rely on statistical analysis, and making use of Bayes Law ($p[s|r] = \frac{p[r|s]p[s]}{p[r]}$). Again this is a side-note since we will be using machine learning to find the stimulus that caused a certain response by emulating the neural network itself.

Neuron Models

It is important to be able to derive mathematical equations to represent the function of a neuron. This is so that we can accurately simulate them in code. The electric potential of a neuron is controlled by the flow of charged ions in or out of a cell. This means that there are certain gates that control this flow based on whether the neuron is being stimulated to spike or not. These gates and the potential of a neuron can be depicted by a network of parallel resistor-capacitor circuits 5.

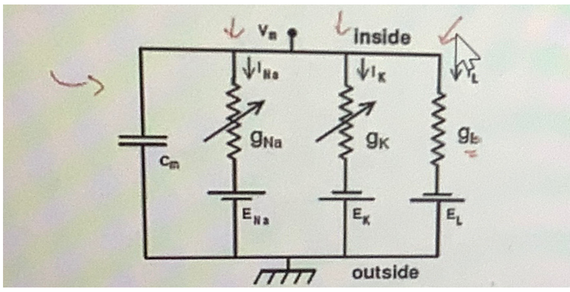

Figure 5: Circuit diagram of neuron

This circuit can then be converted into the following formula:

$$C_m \frac{dv}{dt} = -\sum_i g_i(V - E_i) + I_e$$

Where C_m sis the capacitive current, $g_i(V - E_i)$ are the ionic currents and I_e is the external current. This then gave rise to the famous Hodgkin-Huxley model, which is comprised of various joint differential equations, one for each ionic channel. There are multiple models for a neuron, but during this project only two basic models were used for simplicity's sake.

2.2 Brain

The next component in the project is the Brian package developed by the Neural Reckoning research group [3]. It is used to create and simulate neural networks. The aim of the project is to apply machine learning techniques to spiking neural networks, and so rather than just using PyTorch to create neural network, Brian was used instead. The main reason for doing so is stated on the Brian page itself: "Brian has a powerful, easy to understand syntax that can define, run and plot neural models in just a few lines of code".

2.3 Machine Learning

The final component was to learn machine learning. To learn the basics a blog by Assaad MOAWAD [4] and a freeCodeCamp.org tutorial [2] were used. A code snippet showing how linear regression code looks using PyTorch is in A 4 in the appendix. Then Friedmann Zenke's tutorial [6] showed how PyTorch could be used for spiking neural networks. It was decided, however, to make use of the Brian Package rather than to use Pytorch.

2.4 Random Sampling Optimisation

In order to apply the machine leaning algorithm to a network simulated by Brian, a new optimisation technique needs to be used to replace the default linear regression approach. The technique chosen was found in a paper written by Rohun Tripathi & Bharat Singh [5]. The basics of the approach is that rather than differentiating the loss function to travel against the gradient to reach a minimum, you instead sample what the result would be if we increase/decrease the weight slightly. Then we can simply choose the weight that decreases the loss function. More formally the formula is given by:

$$w_{i+1} = \begin{cases} w_i & f(x, w_i) <= f(x, w_i + \Delta w_i) \\ w_i + \Delta w_i & f(x, w_i) > f(x, w_i + \Delta w_i) \end{cases}$$

Where Δw_i is either a small increment or decrement on the current weight.

3 Application

I applied all the concepts learnt to a scorpion example. The final code is shown in the appendix (See A 1). A scorpion uses 8 neurons on its legs to detect where a potential prey is located. The receptor neurons detect oncoming waves at different times, and send inhibitory or excitatory signals to the 8 control neurons for it to successfully locate and hunt the prey. For the machine learning algorithm we have recreated the network but now the control neurons are simple integrate and fire neurons with a weight attribute.

The network iterates over multiple angles and finds the synaptic weights that minimise the mean error of angle estimation. The optimisation technique seemed to be very efficient in that it needed very few iterations overall to minimise the error, but the problem was that it had to iterate over each synaptic weight, meaning for larger networks this algorithm slowed down exponentially.

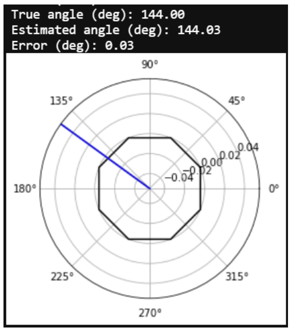

Figure 6: The system learning to find angle of prey

4 Challenges and Reflection

4.1 The Project

The main challenge when tackling this project was wrapping my head around a brand new field that I had never looked at before. With the timescale I had I thought I would be able to create more complex networks that I could analyse, but unfortunately after learning from all the resources there was very little time to actually put it into practice. I am very happy that I was able to apply this optimisation technique at all, and am satisfied that I have at least demonstrated the viability of this technique on spiking networks. If I had the time I would have loved to continue making this algorithm more efficient so that I could later attempt to use it on more complex networks. The algorithm also struggled to find a network to find all angles effectively, I hope to continue and change the algorithm to either use more accurate neuron models or to learn with more angles to allow it to work with all angles will lower errors.

4.2 Personal Progression

This project has allowed me to widen my knowledge on a topic that has always been an enigma. I think that what I have learnt about machine learning in general will help me tremendously moving forward, especially since I hope to undertake machine learning modules in next year. I thoroughly enjoyed the experience and I loved the independence researching gave me, and would love to continue with this as well as hopefully find other interesting projects to undertake during my time at university. Personally, I think this independence has also made me realise the importance of self-management and self-motivation. It was imperative for me to set myself deadlines in order to keep on top of my work, and managing the workload was a struggle since everything I was working with was so new to me.

5 Conclusion

In conclusion, this project undertaking research into the field of machine learning shows promise going further. The aim was to use a surrogate optimisation technique in the place of linear regression for use on spiking neural networks, and that was met. Random sampling optimisation technique worked well on simple networks, and took very few iterations to reduce the error drastically. It did take longer to run on longer networks, but this could be tweaked in the future by streamlining the program. The fact that it worked at all is significant though, as now there is way to use machine learning in a field where it was difficult to do so before. The premise of this algorithm is very simple, but it also turns out to be very effective.

References

[1] Rajesh P. N. Rao & Adrienne Fairhall. Computational neuroscience [online course]. https://www.coursera.org/learn/computational-neuroscience. Accessed: 29/07/2020.

[2] Aakash N S (freeCodeCamp.org). Pytorch for deep learning - full course / tutorial. https://www.youtube.com/watch?v=GIsg-ZUy0MY. Accessed: 12/08/2020.

[3] Neural Reckoning Group. Brian [software]. https://briansimulator.org/. Accessed: 15/08/2020.

[4] Assaad MOAWAD. Neural networks and back-propagation explained in a simple way. https://medium.com/datathings/neural-networks-and-backpropagation-explained-in-a-simple-way-f540a3611f5e. Accessed: 10/08/2020.

[5] Bharat Tripathi, Rohun & Singh. Rso: A gradient free sampling based approach for training deep neural networks. *arXiv preprint arXiv:2005.05955*, 2020.

[6] Friedmann Zenke. Spytorch [a tutorial on surrogate gradient learning in spiking neural networks]. https://github.com/fzenke/spytorch. Accessed: 08/09/2020.

A Code Listings

Source Code 1: Final Code

```
from brian2 import *
from brian2tools import *
import numpy as np
import torch
import torch.nn as nn
import torch.nn.functional as F
from torch.utils.data import TensorDataset
from torch.utils.data import DataLoader

# Parameters
degree = 2 * pi / 360.
duration = 500*ms
R = 2.5*cm  # radius of scorpion
vr = 50*meter/second  # Rayleigh wave speed
phi = 144*degree  # angle of prey
phi_1 = phi
A = 250*Hz
deltaI = .7*ms  # inhibitory delay
gamma = (22.5 + 45 * arange(8)) * degree  # leg angle
delay = R / vr * (1 - cos(phi - gamma))   # wave delay

# Inputs

# Wave (vector w)
time = arange(int(duration / defaultclock.dt) + 1) * defaultclock.dt
Dtot = 0.
w = 0.
for f in arange(150, 451)*Hz:
    D = exp(-(f/Hz - 300) ** 2 / (2 * (50 ** 2)))
    rand_angle = 2 * pi * rand()
    w += 100 * D * cos(2 * pi * f * time + rand_angle)
    Dtot += D
w = .01 * w / Dtot

# Rates from the wave
rates = TimedArray(w, dt=defaultclock.dt)
```

```

# Targets

# Leg mechanical receptors
print("SIMULATING LEG SPIKES FOR THE 4 TEST ANGLES: ")
print()
tau_legs = 1 * ms
sigma = .01
eqs_legs = """
dv/dt = (1 + rates(t - d) - v)/tau_legs + sigma*(2./tau_legs)**.5*xi:1
d : second
"""
fake_legs = NeuronGroup(8, model=eqs_legs, threshold='v > 1', reset='v = 0',
↪ refractory=1*ms, method='euler')
fake_legs.d = delay
fake_spikes_legs = SpikeMonitor(fake_legs)

# We use a Network object because later on we don't
# want to include these objects
leg_net = Network(fake_legs, fake_spikes_legs)
leg_net.store()
leg_net.run(duration, report ='text')
# And keep a copy of those spikes
spikes_i = fake_spikes_legs.i
spikes_t = fake_spikes_legs.t
# Now construct the network that we run each time
# SpikeGeneratorGroup gets the spikes that we created before
SGG_1 = SpikeGeneratorGroup(8, spikes_i, spikes_t)
leg_net.restore()

phi = 289*degree  # angle of prey
phi_2 = phi

# Wave (vector w)
time = arange(int(duration / defaultclock.dt) + 1) * defaultclock.dt
Dtot = 0.
w = 0.
for f in arange(150, 451)*Hz:
    D = exp(-(f/Hz - 300) ** 2 / (2 * (50 ** 2)))
    rand_angle = 2 * pi * rand()
```

```
    w += 100 * D * cos(2 * pi * f * time + rand_angle)
    Dtot += D
w = .01 * w / Dtot

# Rates from the wave
rates = TimedArray(w, dt=defaultclock.dt)

# Targets

leg_net.store()
leg_net.run(duration, report ='text')
# And keep a copy of those spikes
spikes_i = fake_spikes_legs.i
spikes_t = fake_spikes_legs.t
# Now construct the network that we run each time
# SpikeGeneratorGroup gets the spikes that we created before
SGG_2 = SpikeGeneratorGroup(8, spikes_i, spikes_t)
leg_net.restore()

phi = 25*degree  # angle of prey
phi_3 = phi

# Wave (vector w)
time = arange(int(duration / defaultclock.dt) + 1) * defaultclock.dt
Dtot = 0.
w = 0.
for f in arange(150, 451)*Hz:
    D = exp(-(f/Hz - 300) ** 2 / (2 * (50 ** 2)))
    rand_angle = 2 * pi * rand()
    w += 100 * D * cos(2 * pi * f * time + rand_angle)
    Dtot += D
w = .01 * w / Dtot

# Rates from the wave
rates = TimedArray(w, dt=defaultclock.dt)

# Targets

leg_net.store()
leg_net.run(duration, report ='text')
```

```
# And keep a copy of those spikes
spikes_i = fake_spikes_legs.i
spikes_t = fake_spikes_legs.t
# Now construct the network that we run each time
# SpikeGeneratorGroup gets the spikes that we created before
SGG_3 = SpikeGeneratorGroup(8, spikes_i, spikes_t)
leg_net.restore()

phi = 312*degree  # angle of prey
phi_4 = phi

# Wave (vector w)
time = arange(int(duration / defaultclock.dt) + 1) * defaultclock.dt
Dtot = 0.
w = 0.
for f in arange(150, 451)*Hz:
    D = exp(-(f/Hz - 300) ** 2 / (2 * (50 ** 2)))
    rand_angle = 2 * pi * rand()
    w += 100 * D * cos(2 * pi * f * time + rand_angle)
    Dtot += D
w = .01 * w / Dtot

# Rates from the wave
rates = TimedArray(w, dt=defaultclock.dt)

# Targets

leg_net.store()
leg_net.run(duration, report ='text')
# And keep a copy of those spikes
spikes_i = fake_spikes_legs.i
spikes_t = fake_spikes_legs.t
# Now construct the network that we run each time
# SpikeGeneratorGroup gets the spikes that we created before
SGG_4 = SpikeGeneratorGroup(8, spikes_i, spikes_t)
leg_net.restore()
print()

# Command neurons

```

```
model_weights = np.random.rand(64)

tau = 1 * ms
eqs_model = '''
dv/dt = (1-v)/tau : 1
'''

fake_neurons_1 = NeuronGroup(8, model=eqs_model, threshold='v>1', reset='v=0', method='exact')
fake_neurons_2 = NeuronGroup(8, model=eqs_model, threshold='v>1', reset='v=0', method='exact')
fake_neurons_3 = NeuronGroup(8, model=eqs_model, threshold='v>1', reset='v=0', method='exact')
fake_neurons_4 = NeuronGroup(8, model=eqs_model, threshold='v>1', reset='v=0', method='exact')
fake_synapses_1 = Synapses(SGG_1, fake_neurons_1, 'weight : 1',  on_pre='v_post += weight')
fake_synapses_2 = Synapses(SGG_2, fake_neurons_2, 'weight : 1',  on_pre='v_post += weight')
fake_synapses_3 = Synapses(SGG_3, fake_neurons_3, 'weight : 1',  on_pre='v_post += weight')
fake_synapses_4 = Synapses(SGG_4, fake_neurons_4, 'weight : 1',  on_pre='v_post += weight')
fake_synapses_1.connect()
fake_synapses_2.connect()
fake_synapses_3.connect()
fake_synapses_4.connect()
fake_synapses_1.weight = model_weights
fake_synapses_2.weight = model_weights
fake_synapses_3.weight = model_weights
fake_synapses_4.weight = model_weights
fake_spikes_1 = SpikeMonitor(fake_neurons_1)
fake_spikes_2 = SpikeMonitor(fake_neurons_2)
fake_spikes_3 = SpikeMonitor(fake_neurons_3)
fake_spikes_4 = SpikeMonitor(fake_neurons_4)

fake_1 = Network(SGG_1, fake_neurons_1, fake_synapses_1, fake_spikes_1)
fake_2 = Network(SGG_2, fake_neurons_2, fake_synapses_2, fake_spikes_2)
fake_3 = Network(SGG_3, fake_neurons_3, fake_synapses_3, fake_spikes_3)
fake_4 = Network(SGG_4, fake_neurons_4, fake_synapses_4, fake_spikes_4)
```

```

# loss function

def abs_err(target, model):
    delta = abs(target - model)
    delta_min = abs(target - model - 360)
    delta_plu = abs(target - model + 360)

    return min(set([delta, delta_min, delta_plu]))

def me(errors):
    mean_err = 0.
    for x in range(4):
        mean_err += errors[1]

    return mean_err/4

# storing origional state
fake_1.store()
fake_2.store()
fake_3.store()
fake_4.store()

errors = ([0.,0.,0.,0.])

# initial configuration
print('FINDING THE INTIAL MEAN ERROR:')
print()
fake_1.run(duration, report='text')
nspikes = fake_spikes_1.count
phi_est = imag(log(sum(nspikes * exp(gamma * 1j))))
print("True angle (deg): %.2f" % (phi_1/degree))
print("Estimated angle (deg): %.2f" % (phi_est/degree))
errors[0] = abs_err((phi_1/degree), (phi_est/degree))
fake_1.restore()

fake_2.run(duration, report='text')
nspikes = fake_spikes_2.count
phi_est = imag(log(sum(nspikes * exp(gamma * 1j))))
print("True angle (deg): %.2f" % (phi_2/degree))
```

```
print("Estimated angle (deg): %.2f" % (phi_est/degree))
errors[1] = abs_err((phi_2/degree), (phi_est/degree))
fake_2.restore()

fake_3.run(duration, report='text')
nspikes = fake_spikes_3.count
phi_est = imag(log(sum(nspikes * exp(gamma * 1j))))
print("True angle (deg): %.2f" % (phi_3/degree))
print("Estimated angle (deg): %.2f" % (phi_est/degree))
errors[2] = abs_err((phi_3/degree), (phi_est/degree))
fake_3.restore()

fake_4.run(duration, report='text')
nspikes = fake_spikes_4.count
phi_est = imag(log(sum(nspikes * exp(gamma * 1j))))
print("True angle (deg): %.2f" % (phi_4/degree))
print("Estimated angle (deg): %.2f" % (phi_est/degree))
errors[3] = abs_err((phi_4/degree), (phi_est/degree))
fake_4.restore()

me_err = me(errors)

print("Mean error (deg): %.2f" % me_err)
print()

print("RUNNING THE LEARNING ALGORITHM:")
print()
l_rate = 0.05
e = 0
for epoch in range(3):
    p = 0
    for s in fake_synapses_1.weight:

        # finding error with no change

        fake_1.run(duration)
        nspikes = fake_spikes_1.count
        phi_est = imag(log(sum(nspikes * exp(gamma * 1j))))
        errors[0] = abs_err((phi_1/degree), (phi_est/degree))
        fake_1.restore()
```

```
        fake_2.run(duration)
        nspikes = fake_spikes_2.count
        phi_est = imag(log(sum(nspikes * exp(gamma * 1j))))
        errors[1] = abs_err((phi_2/degree), (phi_est/degree))
        fake_2.restore()

        fake_3.run(duration)
        nspikes = fake_spikes_3.count
        phi_est = imag(log(sum(nspikes * exp(gamma * 1j))))
        errors[2] = abs_err((phi_3/degree), (phi_est/degree))
        fake_3.restore()

        fake_4.run(duration)
        nspikes = fake_spikes_4.count
        phi_est = imag(log(sum(nspikes * exp(gamma * 1j))))
        errors[3] = abs_err((phi_4/degree), (phi_est/degree))
        fake_4.restore()

        me_err = me(errors)

        # attempt adding learning rate

        fake_synapses_1.weight[p] += l_rate
        fake_synapses_2.weight[p] += l_rate
        fake_synapses_3.weight[p] += l_rate
        fake_synapses_4.weight[p] += l_rate

        fake_1.run(duration)
        nspikes = fake_spikes_1.count
        phi_est = imag(log(sum(nspikes * exp(gamma * 1j))))
        errors[0] = abs_err((phi_1/degree), (phi_est/degree))
        fake_1.restore()

        fake_2.run(duration)
        nspikes = fake_spikes_2.count
        phi_est = imag(log(sum(nspikes * exp(gamma * 1j))))
        errors[1] = abs_err((phi_2/degree), (phi_est/degree))
        fake_2.restore()

```

```
        fake_3.run(duration)
        nspikes = fake_spikes_3.count
        phi_est = imag(log(sum(nspikes * exp(gamma * 1j))))
        errors[2] = abs_err((phi_3/degree), (phi_est/degree))
        fake_3.restore()

        fake_4.run(duration)
        nspikes = fake_spikes_4.count
        phi_est = imag(log(sum(nspikes * exp(gamma * 1j))))
        errors[3] = abs_err((phi_4/degree), (phi_est/degree))
        fake_4.restore()

        add_me_err = me(errors)

        # attempt subtracting learning rate

        fake_synapses_1.weight[p] -= l_rate
        fake_synapses_2.weight[p] -= l_rate
        fake_synapses_3.weight[p] -= l_rate
        fake_synapses_4.weight[p] -= l_rate

        fake_1.run(duration)
        nspikes = fake_spikes_1.count
        phi_est = imag(log(sum(nspikes * exp(gamma * 1j))))
        errors[0] = abs_err((phi_1/degree), (phi_est/degree))
        fake_1.restore()

        fake_2.run(duration)
        nspikes = fake_spikes_2.count
        phi_est = imag(log(sum(nspikes * exp(gamma * 1j))))
        errors[1] = abs_err((phi_2/degree), (phi_est/degree))
        fake_2.restore()

        fake_3.run(duration)
        nspikes = fake_spikes_3.count
        phi_est = imag(log(sum(nspikes * exp(gamma * 1j))))
        errors[2] = abs_err((phi_3/degree), (phi_est/degree))
        fake_3.restore()

        fake_4.run(duration)
```

```
        nspikes = fake_spikes_4.count
        phi_est = imag(log(sum(nspikes * exp(gamma * 1j))))
        errors[3] = abs_err((phi_4/degree), (phi_est/degree))
        fake_4.restore()

        sub_me_err = me(errors)

        min_err = min(set([me_err, add_me_err, sub_me_err]))

        if(min_err == add_me_err):
            fake_synapses_1.weight[p] += l_rate
            fake_synapses_2.weight[p] += l_rate
            fake_synapses_3.weight[p] += l_rate
            fake_synapses_4.weight[p] += l_rate
        if(min_err == sub_me_err):
            fake_synapses_1.weight[p] -= l_rate
            fake_synapses_2.weight[p] -= l_rate
            fake_synapses_3.weight[p] -= l_rate
            fake_synapses_4.weight[p] -= l_rate

        # storing new initial state
        fake_1.store()
        fake_2.store()
        fake_3.store()
        fake_4.store()

        p = p + 1

    print('epoch %i:' % e)
    fake_1.run(duration, report='text')
    nspikes = fake_spikes_1.count
    phi_est = imag(log(sum(nspikes * exp(gamma * 1j))))
    print("True angle (deg): %.2f" % (phi_1/degree))
    print("Estimated angle (deg): %.2f" % (phi_est/degree))
    errors[0] = abs_err((phi_1/degree), (phi_est/degree))
    fake_1.restore()

    fake_2.run(duration, report='text')
    nspikes = fake_spikes_2.count
    phi_est = imag(log(sum(nspikes * exp(gamma * 1j))))
```

```
    print("True angle (deg): %.2f" % (phi_2/degree))
    print("Estimated angle (deg): %.2f" % (phi_est/degree))
    errors[1] = abs_err((phi_2/degree), (phi_est/degree))
    fake_2.restore()

    fake_3.run(duration, report='text')
    nspikes = fake_spikes_3.count
    phi_est = imag(log(sum(nspikes * exp(gamma * 1j))))
    print("True angle (deg): %.2f" % (phi_3/degree))
    print("Estimated angle (deg): %.2f" % (phi_est/degree))
    errors[2] = abs_err((phi_3/degree), (phi_est/degree))
    fake_3.restore()

    fake_4.run(duration, report='text')
    nspikes = fake_spikes_4.count
    phi_est = imag(log(sum(nspikes * exp(gamma * 1j))))
    print("True angle (deg): %.2f" % (phi_4/degree))
    print("Estimated angle (deg): %.2f" % (phi_est/degree))
    errors[3] = abs_err((phi_4/degree), (phi_est/degree))
    fake_4.restore()

    me_err = me(errors)

    print("Mean error (deg): %.2f" % me_err)
    print()

    e = e + 1

# testing network on new angle
print("TESTING NETWORK ON NEW ANGLE:")

phi = 180*degree  # angle of prey
phi_5 = phi

# Wave (vector w)
time = arange(int(duration / defaultclock.dt) + 1) * defaultclock.dt
Dtot = 0.
w = 0.
for f in arange(150, 451)*Hz:
    D = exp(-(f/Hz - 300) ** 2 / (2 * (50 ** 2)))
```

```
    rand_angle = 2 * pi * rand()
    w += 100 * D * cos(2 * pi * f * time + rand_angle)
    Dtot += D
w = .01 * w / Dtot

# Rates from the wave
rates = TimedArray(w, dt=defaultclock.dt)

leg_net.store()
leg_net.run(duration, report ='text')
# And keep a copy of those spikes
spikes_i = fake_spikes_legs.i
spikes_t = fake_spikes_legs.t
# Now construct the network that we run each time
# SpikeGeneratorGroup gets the spikes that we created before
SGG_5 = SpikeGeneratorGroup(8, spikes_i, spikes_t)
leg_net.restore()

fake_neurons_5 = NeuronGroup(8, model=eqs_model, threshold='v>1', reset='v=0',
↪   method='exact')
fake_synapses_5 = Synapses(SGG_5, fake_neurons_5, 'weight : 1',  on_pre='v_post
↪   += weight')
fake_synapses_5.connect()
fake_synapses_5.weight = fake_synapses_4.weight
fake_spikes_5 = SpikeMonitor(fake_neurons_5)

fake_5 = Network(SGG_5, fake_neurons_5, fake_synapses_5, fake_spikes_5)

fake_5.run(duration, report='text')
nspikes = fake_spikes_5.count
phi_est = imag(log(sum(nspikes * exp(gamma * 1j))))
print("True angle (deg): %.2f" % (phi_5/degree))
print("Estimated angle (deg): %.2f" % (phi_est/degree))
error = abs_err((phi_5/degree), (phi_est/degree))
print("Error (deg): %.2f" % error)

rmax = amax(nspikes)/duration/Hz
polar(concatenate((gamma, [gamma[0] + 2 * pi])),
      concatenate((nspikes, [nspikes[0]])) / duration / Hz,
      c='k')
```

```
axvline(phi_5, ls='-', c='g')
axvline(phi_est, ls='-', c='b')
show()
```

Source Code 2: Simulating Hebbian Plasticity

```
from brian2 import *
%matplotlib inline

start_scope()

taupre = taupost = 20*ms
Apre = 0.01
Apost = -Apre*taupre/taupost*1.05
tmax = 50*ms
N = 100

# Presynaptic neurons G spike at times from 0 to tmax
# Postsynaptic neurons G spike at times from tmax to 0
# So difference in spike times will vary from -tmax to +tmax
G = NeuronGroup(N, 'tspike:second', threshold='t>tspike', refractory=100*ms)
H = NeuronGroup(N, 'tspike:second', threshold='t>tspike', refractory=100*ms)
G.tspike = 'i*tmax/(N-1)'
H.tspike = '(N-1-i)*tmax/(N-1)'

S = Synapses(G, H,
             '''
             w : 1
             dapre/dt = -apre/taupre : 1 (event-driven)
             dapost/dt = -apost/taupost : 1 (event-driven)
             ''',
             on_pre='''
             apre += Apre
             w = w+apost
             ''',
             on_post='''
             apost += Apost
             w = w+apre
```

```
                ''')
S.connect(j='i')

run(tmax+1*ms)

plot((H.tspike-G.tspike)/ms, S.w)
xlabel(r'$\Delta t$ (ms)')
ylabel(r'$\Delta w$')
axhline(0, ls='-', c='k');
```

Source Code 3: Feature Selection

```
from __future__ import division
import numpy as np
import matplotlib.pyplot as plt

import pickle
import os
import sys

def compute_sta(stim, rho, num_timesteps):
    """Compute the spike-triggered average from a stimulus and spike-train.

    Args:
        stim: stimulus time-series
        rho: spike-train time-series
        num_timesteps: how many timesteps to use in STA

    Returns:
        spike-triggered average for specified number of timesteps before
    ↪   spike"""

    sta = np.zeros((num_timesteps,))

    # This command finds the indices of all of the spikes that occur
    # after 300 ms into the recording.
    spike_times = rho[num_timesteps:].nonzero()[0] + num_timesteps

```

```
    # Fill in this value. Note that you should not count spikes that occur
    # before 300 ms into the recording.
    num_spikes = len(spike_times)

    # Compute the spike-triggered average of the spikes found.
    # To do this, compute the average of all of the vectors
    # starting 300 ms (exclusive) before a spike and ending at the time of
    # the event (inclusive). Each of these vectors defines a list of
    # samples that is contained within a window of 300 ms before each
    # spike. The average of these vectors should be completed in an
    # element-wise manner.
    #
    # Your code goes here.

    for i in spike_times:
        for j in range(num_timesteps):
            sta[j] = sta[j] + stim[i-num_timesteps+j]

    sta = sta/num_spikes

    return sta

sys.path.append(os.path.realpath('datasets/c1p8.pickle'))
FILENAME = 'datasets/c1p8.pickle'

with open(FILENAME, 'rb') as f:
    data = pickle.load(f)

stim = data['stim']
rho = data['rho']


# Fill in these values
# number in ms
sampling_period = 2
num_timesteps = 150

sta = compute_sta(stim, rho, num_timesteps)

time = (np.arange(-num_timesteps, 0) + 1) * sampling_period
```

```python

plt.plot(time, sta)
plt.xlabel('Time (ms)')
plt.ylabel('Stimulus')
plt.title('Spike-Triggered Average')

plt.show()
```

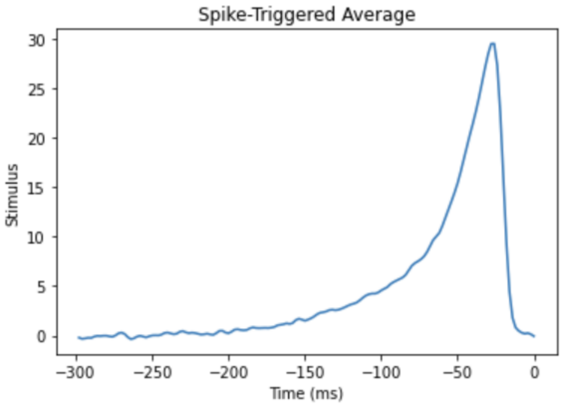

Figure 7: Spike triggered average for feature selection

Source Code 4: Linear Regression using PyTorch

```python
import numpy as np
import torch
import torch.nn as nn
import torch.nn.functional as F
from torch.utils.data import TensorDataset
from torch.utils.data import DataLoader

# Input (temp, rainfall, humidity)
inputs = np.array([[73, 67, 43], [91, 88, 64], [87, 134, 58],
                   [102, 43, 37], [69, 96, 70], [73, 67, 43],
                   [91, 88, 64], [87, 134, 58], [102, 43, 37],
                   [69, 96, 70], [73, 67, 43], [91, 88, 64],
                   [87, 134, 58], [102, 43, 37], [69, 96, 70]],
                  dtype='float32')

```

```
# Targets (apples, oranges)
targets = np.array([[56, 70], [81, 101], [119, 133],
                    [22, 37], [103, 119], [56, 70],
                    [81, 101], [119, 133], [22, 37],
                    [103, 119], [56, 70], [81, 101],
                    [119, 133], [22, 37], [103, 119]],
                   dtype='float32')

inputs = torch.from_numpy(inputs)
targets = torch.from_numpy(targets)

# Define dataset
train_ds = TensorDataset(inputs, targets)

# Define data loader
batch_size = 5
train_dl = DataLoader(train_ds, batch_size, shuffle=True)

model = nn.Linear(3, 2)
loss_fn = F.mse_loss
# Random sampling approach
#def opt(parameters, l_rate, xb, yb):
    # Checking the loss around w

    # Checking the loss around b
opt = torch.optim.SGD(model.parameters(), lr=1e-5)

# Utility function to train the model
def fit(num_epochs, model, loss_fn, opt, train_dl):

    # Repeat for given number of epochs
    for epoch in range(num_epochs):

        # Train with batches of data
        for xb,yb in train_dl:

            # 1. Generate predictions
            pred = model(xb)

            # 2. Calculate loss
```

```
            loss = loss_fn(pred, yb)

            # 3. Compute gradients
            loss.backward()

            # 4. Update parameters using gradients
            opt.step()

            # 5. Reset the gradients to zero
            opt.zero_grad()

        # Print the progress
        if (epoch+1) % 10 == 0:
            print('Epoch [{}/{}], Loss: {:.4f}'.format(epoch+1, num_epochs,
            ↪ loss.item()))

fit(100, model, loss_fn, opt, train_dl)
```