Speech Recognition using Spiking Neural Networks on GPU

Swapnil Ahlawat
ID No. 2018A7PS0178G
Department of Computer Science and Information Systems
f20180178@goa.bits-pilani.ac.in

Neelay Shah
ID No. 2018A8PS0400G
Department of Electrical and Electronics Engineering
f20180400@goa.bits-pilani.ac.in

Under the supervision of

Dr. Basabdatta Sen Bhattacharya
Department of Computer Science and Information Systems

November 26, 2020

Abstract

Presently, artificial neural networks(ANNs) are the mainstream modelling technique for automatic speech recognition [1]. A conventional ANN features a deep multi-layer architecture which makes use of multiple matrix computations for processing data. The rapid progress in the integration of voice interfaces has been viable on account of the remarkable performance of the ASR systems using ANNs for acoustic modeling

The performance gains however, come with immense computational requirements often due to the time-synchronous processing of input audio signals. These models are computationally intensive and memory inefficient to operate as compared to the biological brains. Alternatively, event-driven models such as spiking neural networks (SNNs) inspired by the human brain have attracted ever-growing attention in recent years. Unlike ANNs, asynchronous and event-driven information processing of SNNs resembles the computing paradigm that observed in the human brains, whereby the energy consumption matches the activity levels of sensory stimuli.

The aim of this project is to implement and simulate biologically plausible SNNs on GPU compute for a speech recognition task. We attempt to train an SNN capable of classifying speech signals into one of 10 digits (0-9). We use a supervised Spike-timing-dependent plasticity (STDP) learning rule to train network to differentiate between speech signals belonging to different categories of digits.

Acknowledgement

We would like to express our sincere gratitude to Professor Basabdatta Sen Bhattacharya for her constant guidance and support during the course of the project.

We are grateful to Mr. Sounak Dey from TCS Research who has overseen our work during this project and has provided his invaluable inputs during the process.

We would also like to thank Aiswarya Subramanian, an alumni of BITS Pilani Goa for her help and guidance.

We extend our gratitude to Dr. James Knight from the University of Sussex, co-creator of PyGeNN (a Python framework for neuronal network simulation), for his valuable advice on how to use the framework.

We are also grateful to Mr. Shreenivas A Naik, member of techincal staff at BITS Pilani Goa, who helped us in setting up and accessing GPU on a remote server.

Contents

1 Introduction

Spiking neural networks are computational models that simulate neural behavior in biological systems. Like biological networks, they are driven by discrete spike trains whose instantaneous frequency encodes data. These spikes, by either polarisation or depolarisation, will alter the membrane potential of neurons. Dynamics of the neuron are decided by a set of differential equations that relate membrane voltage with conductance and synaptic current. If a neuron's membrane potential exceeds a certain threshold value, it spikes, passing the integrated information to the next neuron. The influence of a synapse on the membrane voltage of the next neuron can be of two types: excitatory and inhibitory. Excitatory synapses depolarise the neuron, pushing it towards the threshold voltage while inhibitory synapses polarise, thereby reducing the likelihood of a spike. Inhibitory neurons are generally useful to regulate spiking frequency and increase more information contained per spike of the postsynaptic neuron.

There are various neuron models that mimic fairly accurately,the dynamics of biological neurons. The Hodgkin-Huxley model is a conductance-based model that uses a set of non-linear differential equations to explain the electrical characteristics of neurons. While it is a powerful model which is biophysically meaningful, it is very complex and computationally expensive to simulate. Hence, simple spiking neuron models are favoured for applications. A simple model which is widely used is Integrate-and-Fire model. Integrate-and-Fire model considers the neuron like an R-C circuit with a current *I(t)* flowing through the circuit. This current can be simulated either by external current or presynaptic input spikes. option recently has been the Izhikevich neuron model since it is both accurate and computationally efficient for simulation purposes We focus on the Izhikevich neuron model in the next section.

2 Izhikevich neuron model

The Izhikevich's Neuron model is a simple neuron model which combines the biological plausi- bility, accuracy, scalability and flexibility of the Hodgkin-Huxley model and the computational inexpensiveness of the Integrate and Fire Neuron model.

$\frac{dv(t)}{dt} = 0.04\mathrm{v}^2(t) + 5v(t) + 140u(t) + Ipsc(t) + Idc$

$\frac{du(t)}{dt} = \mathrm{a(bv(t) - u(t))}$

If $v(t) > Vpeak$, then v(t) $\leftarrow$ c; u(t) $\leftarrow$ u(t) + d

where a, b, c, d are parameters that affect the dynamics of the model and can be chosen to obtain different kinds of spiking behavior. u(t) is the recovery variable v(t) is the membrane voltage. Ipsc is the post-synaptic current and Idc is the DC bias current stimulus

3 Spike Time Dependent Plasticity

Spike timing dependent plasticity (STDP) is a biological process that adjusts the strength of connections between neurons in a network. The process adjusts the connection strengths based on the relative timing of a particular neuron's output and input action potentials (or spikes). An increase in the strength of the connection is called as Long-Term Potentiation(LTP) whereas a decrease is called Long-Term Depression(LTD)

3.1 Hebbian and Anti-hebbian STDP

In the case of Hebbian STDP, if the postsynaptic spike is generated immediately after receiving the presynaptic spike, the presynaptic spike has a causal role in the output neuron firing. The synaptic weight is thus increased (LTP). Conversely, if a postsynaptic spike occurs before the presynaptic spike, the strength is reduced (LTD), as seen in the equation below

$$\Delta w_{ji} = \begin{cases} Ae^{\frac{(-|t_j - t_i|)}{\tau+}}, & t_j - t_i > 0, A > 0. \\ Be^{\frac{(-|t_j - t_i|)}{\tau-}}, & t_j - t_i < 0, B < 0\ . \end{cases} \tag{1}$$

In the above, the first case (Case 1) covers LTP and the second case (Case 2) covers LTD. Both cases are decaying exponentials that decay with the distance between and pre and postsynaptic spikes. $A > 0$ and $B < 0$ scale the amplitude of the exponential, and τ+ and τ are the respective time constants

Anti-hebbian STDP is simply the reverse of Hebbian STDP

$$\Delta w_{ji} = \begin{cases} Be^{\frac{(-|t_j - t_i|)}{\tau-}}, & t_j - t_i > 0, B < 0\ . \\ Ae^{\frac{(-|t_j - t_i|)}{\tau+}}, & t_j - t_i < 0, A > 0.. \end{cases} \tag{2}$$

4 PyGeNN - A neuronal network simulation framework

GPU-enhanced Neuronal Networks (GeNN) is a software package to enable neuronal network simulations on NVIDIA GPUs by code generation. It is a cross-platform C++ library for generating optimized CUDA code for GPU accelerated spiking neural network simulations.

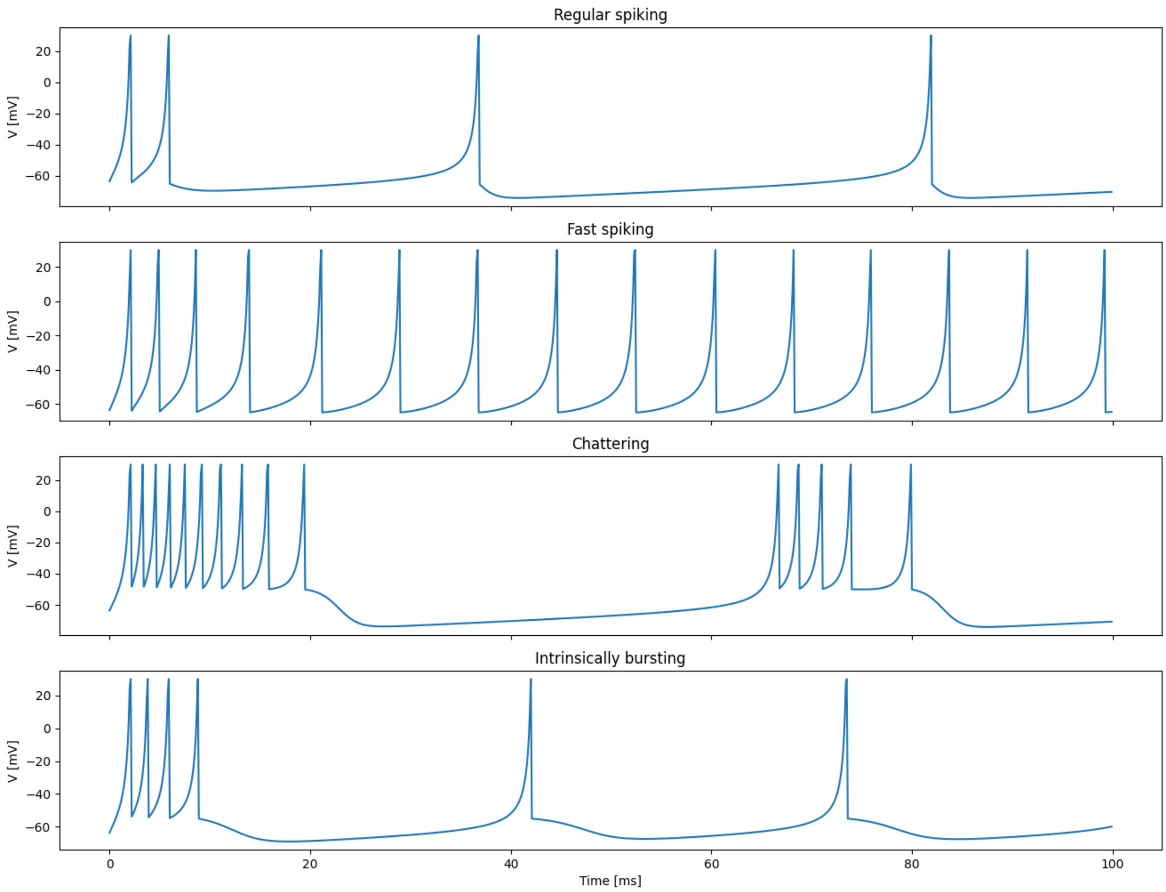
Python interface to GeNN (PyGeNN) is an abstraction on top of GeNN that allows users to access all GeNN features from Python

Given below is an example code snippet highlighting how an Izhikevich neuron can be simulated in 4 regimes for 100 ms using PyGeNN -

```
import numpy as np
import matplotlib.pyplot as plt
from pygenn.genn_model import GeNNModel

# Create a single-precision GeNN model
model = GeNNModel("float", "pygenn")

# Set simulation timestep to 0.1ms
model.dT = 0.1

# Initialise IzhikevichVariable parameters
izk_init = {"V": -65.0,
            "U": -20.0,
            "a": [0.02,     0.1,    0.02,    0.02],
            "b": [0.2,      0.2,    0.2,     0.2],
            "c": [-65.0,    -65.0,  -50.0,   -55.0],
            "d": [8.0,      2.0,    2.0,     4.0]}

# Add neuron populations and current source to model
pop = model.add_neuron_population("Neurons", 4, "IzhikevichVariable",
                                  {}, izk_init)
model.add_current_source("CurrentSource", "DC", "Neurons",
                         {"amp": 10.0}, {})

# Build and load model
model.build()
model.load()

# Create a numpy view to efficiently access the membrane
voltage from Python
voltage_view = pop.vars["V"].view

# Simulate
v = None
while model.t < 100.0:
    model.step_time()
    model.pull_state_from_device("Neurons")
    v = np.copy(voltage_view) if v is None
        else np.vstack((v, voltage_view))
```

```python

# Create plot
figure, axes = plt.subplots(4, sharex=True)

# Plot voltages
for i, t in enumerate(["RS", "FS", "CH", "IB"]):
    axes[i].set_title(t)
    axes[i].set_ylabel("V [mV]")
    axes[i].plot(np.arange(0.0, 200.0, 0.1), v[:,i])
axes[-1].set_xlabel("Time [ms]")
```

Figure 1: Membrane potential vs Simulation Time



5 Current Work

We aim to replicate the results of *A spiking network that learns to extract spike signatures from speech signals* [2]. The method described in the paper uses a compact, non-recurrent SNN consisting of Izhikevich neurons equipped with STDP for a spoken digit recognition task.

We are making use of the Free Spoken Digit Dataset (FSDD) - an open speech dataset consisting of recordings of spoken digits in English by 6 native English speakers.

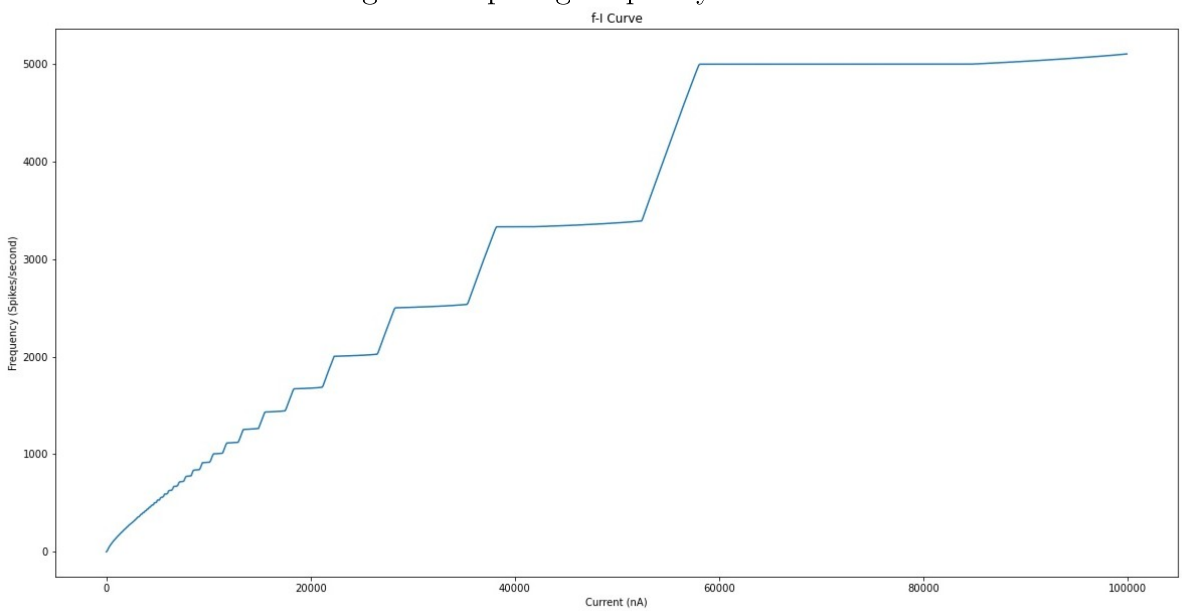
Described hereon are the different steps we have completed towards trying to replicate the previous speech recognition work -

5.1 Observing behaviour of Izhikevich neurons with custom parameters

PyGeNN allows users the flexibility of simulating Izhikevich neurons with custom parameters. We use the values of parameters mentioned in [2] in order to have a faithful comparison when trying to replicate results.

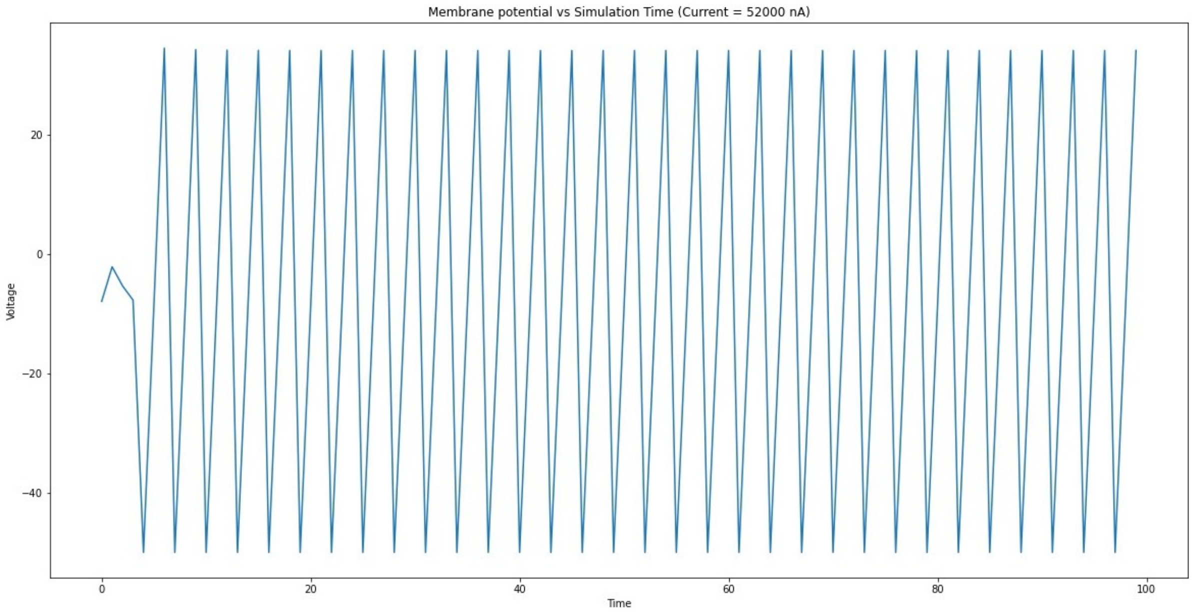
We simulated a single Izhikevich neuron for a range of current values and for a time duration of 1000 ms with a timestep of 1 ms to determine the current sweep for which the neurons display desired spiking behaviour (Figure 2). We observed the frequency of spikes for the neuron for the range of current values. Simultaneously, we also recorded the membrane potential of the neuron to keep track of the model dynamics.

Figure 2: Spiking frequency vs Current



We observed that the neuron starts spiking at an input current of 52 nA, and the spiking frequency saturates at a current value of 59000 nA. We checked the membrane potential over time for the neuron to confirm whether 59000 nA was a suitable upper bound of input current. However, We noticed irregularties in the action potentials of the neuron for a current value of greater than 52000 nA. Hence, we decided to use 52000 nA as the upper bound of input current to be fed to the Izhikevich neurons during the speech recognition task, with 52 nA being the lower bound.

Figure 3: Membrane potential vs Simulation time



5.2 Feature extraction from speech dataset

We follow the same procedure as [2] for extracting representative feature vectors to be used as input to the SNN from raw speech signals. The speech signal is divided into small overlapping time sections called speech frames. A fixed number of frames, N, is used for for each spoken digit. The length, L, of a spoken digit varies from 500 to 1000 ms, and was divided into N=40 frames with 50% overlap to support a frame length of 10-50 ms. The 50% overlap captures the temporal characteristics of the changing spectrum of the speech signal.

After framing, a small feature vector based on the frame's frequency spectrum is extracted. The spectrum values are calculated for all the frames temporally to represent the speech signal spectrogram. A frame encompassing an R Hz frequency range can be divided into M frequency bands. We use R = 4000 Hz and M = 5 to produce a feature vector of length N*M = 200 for each spoken digit. The number of filter banks (M = 5) is small enough to create a minimal SNN.

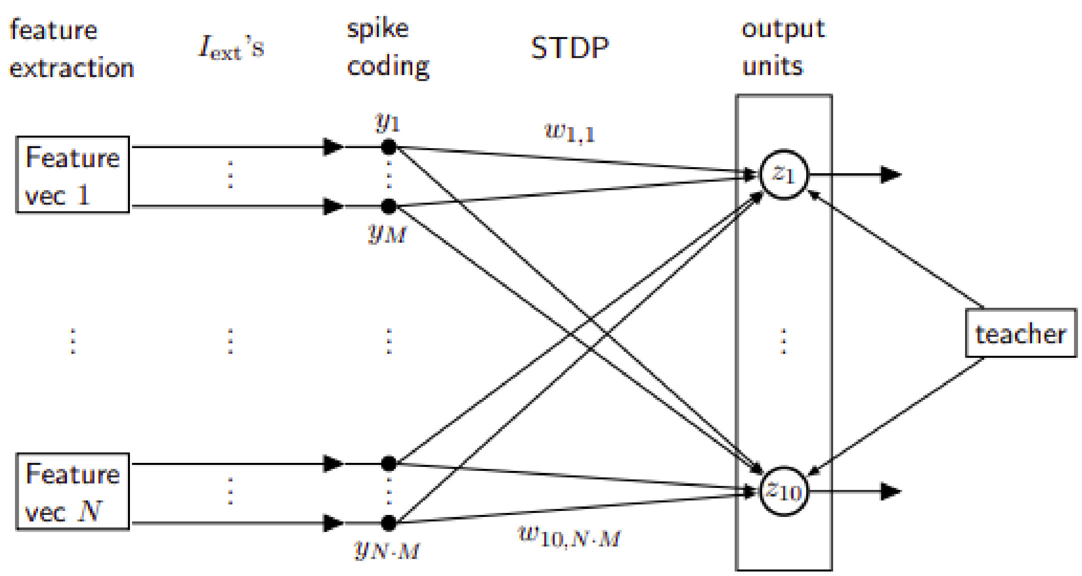
5.3 Network Architecture

We use Izhikevich neurons in pur spiking network. The speech signal is divided into 40 input vectors and each input vector has 5 features. Each of these are connected to one neuron. Therefore, the input layer has 200 neurons (denoted by y unit layer).

The output layer has 10 neurons (denoted by z unit layer). Each unit corresponds to one of the ten spoken digit categories (class labels). The y units are fully connected to the z units, i.e., there is no hidden layer.

Finally, there is a supervision signal (called a teacher) that monitors the z units in order to determine the form of the STDP used in training. The teacher determines which z units undergo Hebbian versus anti-Hebbian STDP. During training, whenever a z unit emits a spike, it undergoes some form of STDP. If the z unit represents the target category, then it undergoes Hebbian STDP. Otherwise, it undergoes anti-Hebbian STDP The teaching signal is only used for the training phase.

Figure 4: Network Architecture



5.4 Feeding data into input neurons

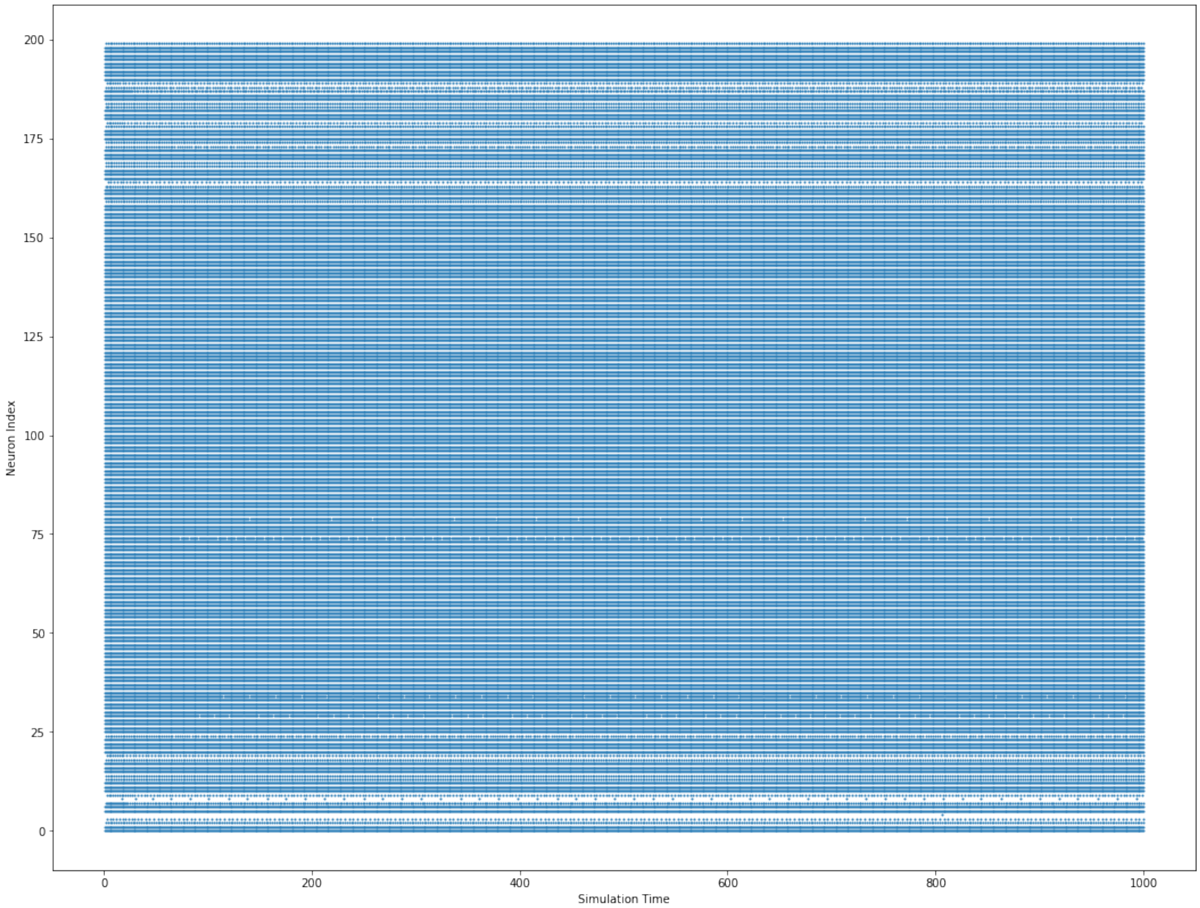
We employ an input layer of 200 Izhikevich neurons in our network; one neuron for each element in the feature vector of a speech signal. The values of the features are used as constant DC current sources for the network simulation. We normalize the feature values between the lower and upper bounds determined previously.

We supply one sample speech signal as input and simulate the network for a period of 100 ms. We observe the behaviour of the 200 input neurons in the form of a raster plot.(Figure 5)

5.5 Implementing STDP in PyGeNN

We implemented both hebbian and anti-hebbian forms of STDP in PyGeNN for a spiking neural network of 2 neuron layers. We use the utility of defining custom learning rules

Figure 5: Raster plot of the input (y) layer neurons



PyGeNN provides to implement both the STDP forms. We initialize the synaptic weights between neurons uniformly between 0.1 and 1.0 and limit them to the same upper and lower bounds during the training process as well. We employ the same STDP parameters (in equations (1) and (2)) as [2]. We track the synaptic weights, spike times and membrane potentials of all the neurons in the network using state variables.

5.6 Teacher supervision in PyGeNN

To facilitate the supervision signal, we make use of an 'index' parameter in the network. Every z unit (output neuron) has the aforementioned 'index' parameter which corresponds to what digit that particular z unit represents. For example, the first z unit would have an index of 0 because it corresponds tot the digit 0. We also employ a variable called 'label' which denotes which digit category a speech signal belongs to. The 'label' variable is updated every time a new speech signal is being processed.

Every time a z unit spikes, its index is compared to the label of the speech signal currently being processed. If the index matches the label, the 'correct' neuron for that particular signal has spiked and it undergoes hebbian SDTP. If the index of a neuron doesn't match the label, it undergoes anti-hebbian STDP.

5.7 Network training

We simulated the network for 1 speech sample using teacher supervision as described above. We plotted and observed the behaviour of the neurons in the output (z) layer. Figure 6 shows the membrane potential over time of one z unit. Regular spiking behaviour is observed.

Figure 6: Membrane potential of neuron 1 in the output layer (1st z unit)

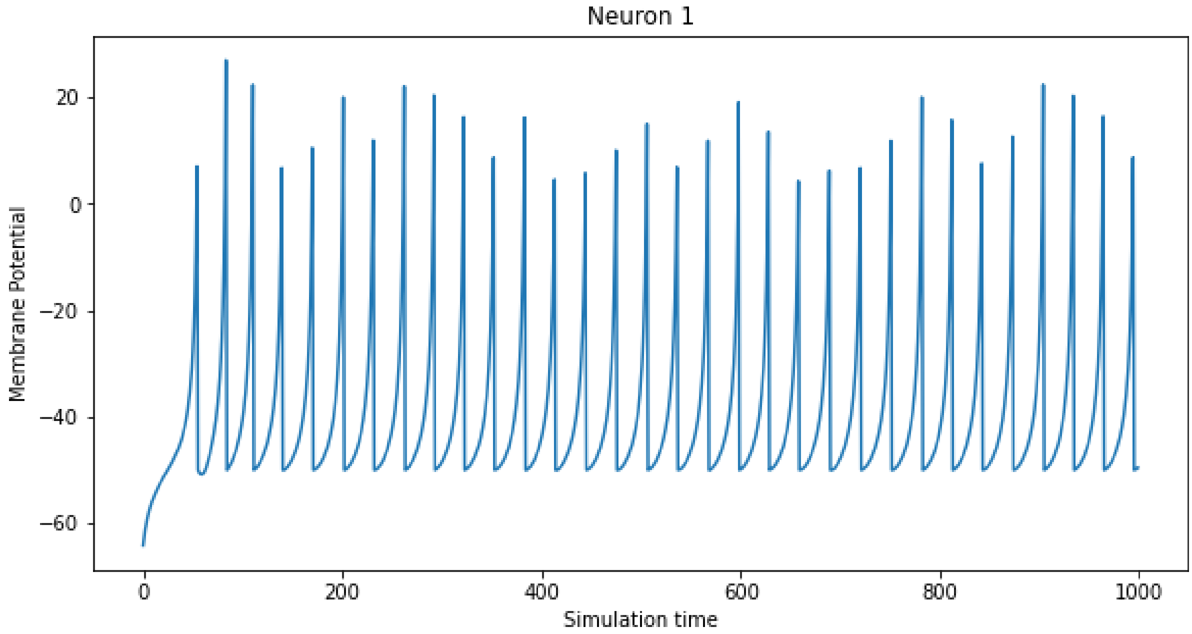


Figure 7: Raster plot of the output (z) layer neurons

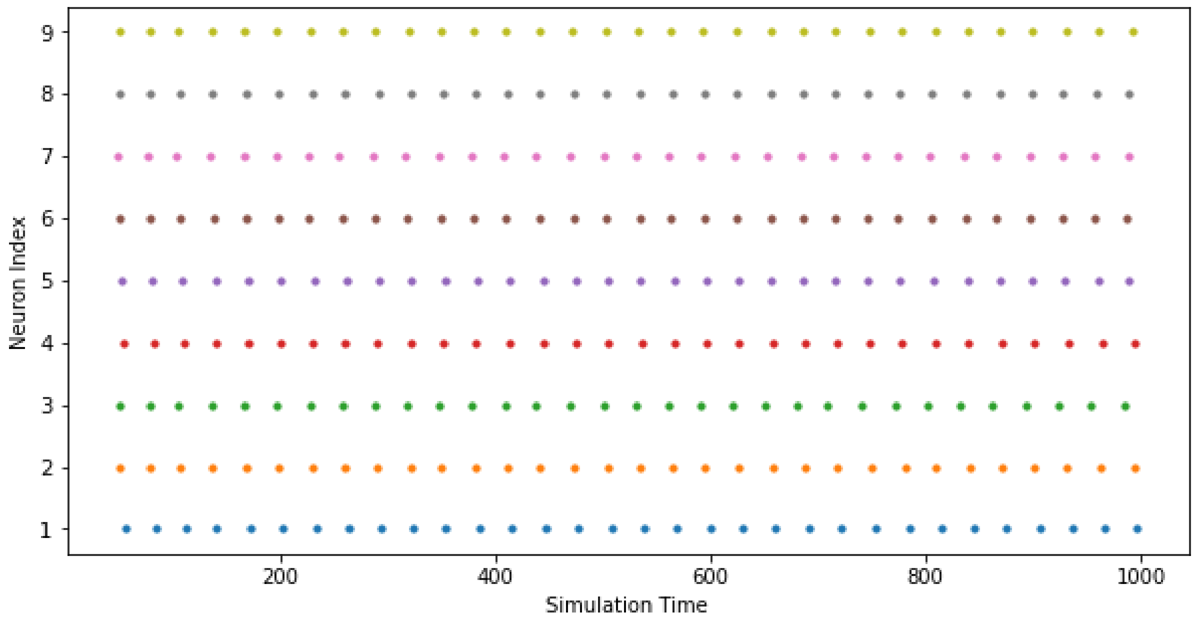


Figure 7 displays a raster plot of the output layer neurons. Since the network hasn't been trained yet with enough training samples to distinguish between speech signals belonging to different categories, all the output layer neurons display similar spiking behaviour. After training, we expect the spiking behaviour of the neurons to be representative of the category of the speech signal being processed.

6 Conclusion and Future Work

During the course of the project, we were able to learn how to simulate neuronal networks on GPU compute using PyGeNN. We understood how spiking neurons process information, how learning takes place in spiking neural networks and what are the ways by which we can decode what an SNN has learnt.

We were able to develop a neuronal network for speech recognition in PyGeNN with an aim of replicating the results in [2]. We tested the working of the network we developed on 1 training sample from a spoken digit speech dataset.

We plan to carry out the following steps here onwards -

- Train the network for the entire training portion of the speech dataset.
- Obtain spike signatures for a test portion of speech signals in the dataset and calculate evaluation metrics to judge performance.

Appendix

PyGeNN code for training the network using supervised STDP -

```
# Imports

import pandas as pd
import numpy as np
from pygenn import genn_model, genn_wrapper
from pygenn.genn_model import (
    create_custom_neuron_class,
    create_custom_current_source_class,
    create_custom_weight_update_class,
    GeNNModel,
)
import argparse
from os.path import exists, join
import os


# Command line arguments

parser = argparse.ArgumentParser("Script to train model the SNN using
    supervised STDP")

parser.add_argument(
    "--datafile",
    type=str,
    required=True,
    help="Path to the .npy file containing the speech data",
)
parser.add_argument(
    "--outdir",
    type=str,
    default="output",
    help="Name of folder where all the ouput files (membrane potentials
     etc) should be stored. Folder doesn't need to exist beforehand. (
    default = output)",
)
parser.add_argument(
    "--n_samples",
    type=int,
    default=1,
    help="Number of samples in the dataset for which the network should
     be simulated. (default=1)",
)
args = parser.parse_args()


# Define custom neuron class

izk_neuron = create_custom_neuron_class(
    "izk_neuron",
    param_names=["a", "b", "c", "d", "C", "k"],
    var_name_types=[("V", "scalar"), ("U", "scalar")],
```

```
    sim_code="""
        $(V)+= (0.5/$(C))*($(k)*($(V) + 60.0)*($(V) + 40.0)-$(U)+$(Isyn
    )); //at two times for numerical stability
        $(V)+= (0.5/$(C))*($(k)*($(V) +  60.0)*($(V) + 40.0)-$(U)+$(
    Isyn));
        $(U)+=$(a)*($(b)*($(V) + 60.0)-$(U))*DT;
        """,
    reset_code="""
        $(V)=$(c);
        $(U)+=$(d);
        """,
    threshold_condition_code="$(V) > 30",
)


# Set the parameters and initial values of variables for the IZK neuron
     (whose class has been defined above)

izk_params = {"a": 0.03, "b": -2.0, "c": -50.0, "d": 100.0, "k": 0.7, "
    C": 100.0}

izk_var_init = {
    "V": -60.0,
    "U": 0.0,
}


# Define the weight update (supervised STDP learning) rule

# An additional called parameter called 'index' is created. Each neuron
     in the ouput layer corresponds to a digit from 0-9. This digit a
    neuron in the output layer corresponds to is stored in 'index'.
    Therefore, each post neuron has an index

# An additional variable called 'label' is created. 'label' is the
    category (digit:0-9) that the speech signal currently processed
    belongs to. 'label' is updated every time a new speech signal is
    being processed

# 'sim_code' is called when a pre-synaptic spike occurs

# 'learn_post_code' is called whena post-synaptic spike occurs

# Whether the correct post neuron corresponding to a particular speech
    signal being processed has spiked is determined by comparing the '
    label' of the speech signal to the 'index' of the post neruon

# Logic in 'sim_code': when a pre-synaptic spike occurs -
#                                                                    if
    the the correct post neuron for the speech signal currently being
    processed fires , the synaptic strength is reduced     (Hebbian
    learning case 2)
#                                                                    if
    the the incorrect post neuron for the speech signal currently being
```

```
processed fires , the synaptic strength is reduced  (Anti-hebbian learning case 2)

# Logic in 'learn_post_code': when a post-synaptic spike occurs -
#                                                  if the the correct post neuron for the speech signal currently being processed fires , the synaptic strength is increased  (Hebbian learning case 1)
#                                                  if the the incorrect post neuron for the speech signal currently being processed fires , the synaptic strength is increased (Anti-hebbian learning case 1)

supervised_stdp = create_custom_weight_update_class(
    "supervised_stdp",
    param_names=["tauMinus", "tauPlus", "A", "B", "gMax", "gMin", "index"],
    var_name_types=[("g", "scalar"), ("label", "scalar")],
    sim_code="""
        $(addToInSyn, $(g));
        scalar dt = $(t) - $(sT_post);
        if(dt > 0) {
            if($(index) == $(label)){
                scalar timing = exp(-dt / $(tauMinus));
                scalar newWeight = $(g) - ($(B) * timing);
                $(g) = fmax($(gMin), newWeight);
                            }

            else{
                scalar timing = exp(-dt / $(tauPlus));
                scalar newWeight = $(g) + ($(A) * timing);
                $(g) = fmin($(gMax), newWeight);
            }
        }

        """,
    learn_post_code="""
        scalar dt = $(t) - $(sT_pre);
        if (dt > 0) {
            if($(index) == $(label)){
                scalar timing = exp(-dt / $(tauPlus));
                scalar newWeight = $(g) + ($(A) * timing);
                $(g) = fmin($(gMax), newWeight);
                            }

            else{
                scalar timing = exp(-dt / $(tauMinus));
                scalar newWeight = $(g) - ($(B) * timing);
                $(g) = fmax($(gMin), newWeight);
            }
        }
        """,
    is_pre_spike_time_required=True,
    is_post_spike_time_required=True,
)
```

```python


# Set the initial values of variables of the weight update rule

# Initialize weights uniformly between 0 and 1

stdp_var_init = {
    "g": genn_model.init_var("Uniform", {"min": 0.1, "max": 1.0}),
    "label": 0.0,
}  # Initialize label to 0


# Define GeNN model

model = genn_model.GeNNModel("float", "speech_recognition")


# Add neuron populations

num_inp_neurons = 200
num_output_neurons = 10

inp_layer = model.add_neuron_population(
    "input_layer", num_inp_neurons, izk_neuron, izk_params,
   izk_var_init
)

neuron_layers = [inp_layer]

for i in range(10):
    neuron_layers.append(
        model.add_neuron_population(
            "output_neuron_" + str(i), 1, izk_neuron, izk_params,
   izk_var_init
        )
    )


# Create synaptic connections

# Each synapse group contains synapse connections from all 200 input
   neurons to 1 particular neuron in the output layer. 10 such synapse
   groups are created, one for every neuron in the output layer

# Every synapse group contains (belonging to specific output neuron)
   contains the 'index' of that neuron

syn_io = []
for i in range(num_output_neurons):
    syn_io.append(
        model.add_synapse_population(
            "synapse_input_output_" + str(i),
            "DENSE_INDIVIDUALG",
            genn_wrapper.NO_DELAY,
            inp_layer,
```

```
            neuron_layers[i + 1],
            supervised_stdp,
            {
                "tauMinus": 20.0,
                "tauPlus": 20.0,
                "A": 0.1,
                "B": 0.1,
                "gMax": 1.0,
                "gMin": 0.1,
                "index": float(i),
            },
            stdp_var_init,
            {},
            {},
            "DeltaCurr",
            {},
            {},
        )
    )


# Define current source

current_source = create_custom_current_source_class(
    "current_source",
    var_name_types=[("magnitude", "scalar")],
    injection_code="$(injectCurrent, $(magnitude));",
)


# Create current input

current_input = model.add_current_source(
    "input_current", current_source, inp_layer, {}, {"magnitude": 0.0}
)


# Set simulation parameters

timesteps_per_sample = (
    1000.0  # No. of timesteps one speech signal in the dataset is presented for
)
resolution = 1


# Build and load model

model.dT = resolution
model.build()
model.load()


# Load data

```

```python
dataset = np.load(args.datafile, allow_pickle=True).item()
data = dataset["data"]
labels = dataset["labels"]


# Initialize data structures for variables and parameters we want to track

layer_spikes = [(np.empty(0), np.empty(0)) for _ in enumerate(neuron_layers)]
layer_voltages = [l.vars["V"].view for l in neuron_layers]
current_input_magnitude = current_input.vars["magnitude"].view[:]
neuron_labels = [
    syn_io[neuron].vars["label"].view for neuron in range(num_output_neurons)
]

input_voltage_view = inp_layer.vars["V"].view[:]
input_voltage = None
output_voltage = {}
output_voltage_view = {}

for index, output_neuron in enumerate(neuron_layers[1:]):
    output_voltage[index] = None
    output_voltage_view[index] = output_neuron.vars["V"].view

synaptic_weights = {}
synaptic_weight_views = {}
for index, synapse in enumerate(syn_io):
    synaptic_weights[index] = None
    synaptic_weight_views[index] = synapse.vars["g"].view[:]


# Simulate

num_simulation_samples = args.n_samples

while model.t < timesteps_per_sample * num_simulation_samples:

    timestep_in_example = model.t % timesteps_per_sample
    sample = int(model.t // timesteps_per_sample)

    if timestep_in_example == 0:  # If a new sample is starting to be processed

        label = labels[sample]
        print(f"Processing sample {sample}: {label}")

        current_input_magnitude[:] = data[
            sample
        ]  # Update the current input for the new sample
        model.push_var_to_device("input_current", "magnitude")

        for l, v in zip(neuron_layers, layer_voltages):

```

```python
            v[:] = -65.0  # Manually 'reset' voltage
            l.push_var_to_device("V")

        for index, synapse in enumerate(syn_io):
            neuron_labels[index] = float(label)
            synapse.push_var_to_device("label")  # Update the 'label' for the new sample

    model.step_time()  # Simulate a timestep

    # record input neurons membrane potential

    model.pull_state_from_device("input_layer")
    input_voltage = (
        np.copy(input_voltage_view)
        if input_voltage is None
        else np.vstack((input_voltage, input_voltage_view))
    )

    # record output neurons membrane potential

    for index, output_neuron in enumerate(neuron_layers[1:]):
        model.pull_state_from_device("output_neuron_" + str(index))
        output_voltage[index] = (
            np.copy(output_voltage_view[index])
            if output_voltage[index] is None
            else np.hstack((output_voltage[index], output_voltage_view[index]))
        )

    # record synaptic weights

    for synapse_index, synapse in enumerate(syn_io):
        synapse.get_var_values("g")
        synaptic_weights[synapse_index] = (
            np.copy(synaptic_weight_views[synapse_index])
            if synaptic_weights[synapse_index] is None
            else np.vstack(
                (synaptic_weights[synapse_index], synaptic_weight_views[synapse_index])
            )
        )

    # record spikes

    for i, l in enumerate(neuron_layers):
        model.pull_current_spikes_from_device(l.name)
        spike_times = np.ones_like(l.current_spikes) * model.t
        layer_spikes[i] = (
            np.hstack((layer_spikes[i][0], l.current_spikes)),
            np.hstack((layer_spikes[i][1], spike_times)),
        )


# Create ouput directory
```

```python

if not exists(args.outdir):
    os.mkdir(args.outdir)


# Save all output files

pd.DataFrame(input_voltage).to_csv(
    join(args.outdir, "input_neurons_membrane_potential.csv"), header=None, index=None
)
np.save(join(args.outdir, "output_neurons_membrane_potential"), output_voltage)
np.save(join(args.outdir, "synaptic_weights"), synaptic_weights)
np.save(join(args.outdir, "spikes_data"), layer_spikes)
```

References

[1] J. Wu, E. Yılmaz, M. Zhang, H. Li, and K. C. Tan, "Deep spiking neural networks for large vocabulary automatic speech recognition," *Frontiers in Neuroscience*, vol. 14, p. 199, 2020.

[2] A. Tavanaei and A. S. Maida, "A spiking network that learns to extract spike signatures from speech signals," *Neurocomputing*, vol. 240, p. 191–199, May 2017.