Major in Computer Science (HB)

# Using a Spiking Neural Network for Spoken Stop Consonant Classification

**Hunter Betz**
5/9/17
443 Integration of Computer and Brain Sciences
Prof. Konstantinos Michmizos

## Abstract

With the emergence of intelligent personal assistants that primarily rely on voice input, such as Amazon's Alexa and Google Assistant, there is increased demand for speech recognition systems. Being able to quickly and efficiently recognize spoken letters is the first step in developing such a system. Our goal is to use a spiking neural network (SNN) to accurately classify spoken stop consonants. The network utilized *Izhikevich* regularly spiking neurons and spike-timing dependent plasticity to learn from a database of training data. Our SNN generated distinct spiking signatures for each of the spoken stop consonants, but ultimately failed to accurately classify the consonants based on these signatures. This result was not due to the network architecture, but was likely caused by the equation used to calculate the injected current. We assume that further investigation would yield better results, however our study was a step in the right direction.

## 1. Introduction

Speech recognition has become increasingly popular; as a result, the need for accurate and fast speech recognition systems is rising in demand. Using neural networks for speech recognition is not new and past studies have used various neural networks for their models. Using a Spiking Neural Network (SNN), it was determined that the English stop consonants (b, d, g, k, t, p) produce distinct spiking patterns. These distinct patterns allow the spoken consonants to be identified and classified, which can be useful in many speech recognition systems such as Amazon's Alexa, Google Assistant, or Microsoft's Cortana.

Inspiration for this study came from an article published in Neurocomputing by Amirhossein Tavanaei and Anthony Maida from the Center for Advanced Computer Studies at the University of Louisiana [1]. In this study, the authors used an SNN to generate distinct spike signatures for the spoken digits 0 – 9 and to classify the spoken digit based off these signatures. We decided to take a similar approach but apply it to the English stop consonants which had been done in a separate study using a Time Delay Neural Network (TDNN). By successfully achieving our goal in this project, we hope to expand the list of letters that can be recognized to include the entire English alphabet.

## 2. Background

Previous studies have been conducted on neural networks and their efficiency in classifying distinct sounds. One study looked at the classification of English stop consonants using a Time Delay Neural Network (TDNN) achieving good results [2]. We hope to achieve similar results using the same feature extraction method, different dataset, and an SNN. Another study used a two layer SNN to extract distinct spike signatures from spoken digits [1]. The study concluded that each digit, when spoken, produced a distinct spike signature that distinguished it from the other digits. Using the same neuron model, network architecture, and learning model we hope to create an SNN that can produce distinct spike signatures for each of the consonants that will enable us to effectively classify each signature to the correct sound.

One of the most crucial steps in speech classification is choosing an adequate feature extraction method. Popular methods include obtaining the frequency spectrum of the signal and obtaining the Mel Frequency Cepstral Coefficients (MFCC) of the signal. When extracting the features from our signal, it is important that we extract a fixed number of features regardless of the duration of the signal. We achieve this by splitting our signal into a fixed number of frames with a 50% overlap. The audio in our dataset ranges from 300ms to 1000ms in duration so for our purposes we will split our signal into 40 frames resulting in each frame ranging from 15ms – 50ms in length. We calculate the frame length using equation 1 below:

$$\text{frame length} = \frac{L}{N(1-y)+y} \qquad (1)$$

*L* is the length of the audio signal in milliseconds, *N* is the number of frames we want to split our signal into, and *y* is the overlap with a value between 0 and 1. For each of these frames we will perform our feature extraction which will return a 13-element vector containing the MFCCs of that frame.

Since many of the feature values returned from the MFCC method are less than or equal to 0, we need to develop a function that will scale these values such that we can obtain a corresponding input current that will cause our input neurons to fire. Equation 2, below, is the function we came up with to scale our feature values accordingly:

$$I_{inj} = \begin{cases} 8.5 & x = 0 \\ .05309x + 8.5 & x < 0 \\ .11806x + 8.5 & x > 0 \end{cases} \quad (2)$$

When our feature value is 0, the input neuron will fire at a rate of 20Hz. When our feature value is less than 0, our input neuron's firing rate will be below 20Hz, but greater than 2Hz. Finally, when our feature value is greater than 0, our input neuron's firing rate will be greater than 20Hz, but not exceed 38Hz.

Each of the input neurons will be connected to each of the output neurons via synapses that will undergo spike-timing dependent plasticity (STDP) learning. Each synapse has a function that models the conductance time-course of it receiving one spike at time t:

$$I(t) = w * t * e^{-t/\tau} \quad (3)$$

*w* is the synapse weight which is the value that is adjusted during learning. $\tau$ is a time constant which represents the time at which the synapse reaches its maximum conductance. This models the conductance of only a single synapse during the entire duration of the experiment. To generate the total synaptic conductance of 520 input synapses we sum over all of the synapses and each of their input spikes according to the equation below:

$$I_{total} = \sum_{k=1}^{520} \sum_{j=1}^{N_k} w * \left(t - t_{kj}\right) * e^{-\frac{t-t_{kj}}{\tau}} \quad (4)$$

$t_{kj}$ is the time where synapse $k$ receives a spike $j$ and $N_k$ denotes the number of spikes received by synapse $k$. This value then gets propagated to the output neuron as the input current.

There are two types of STDP learning that we are implementing; Hebbian STDP and anti-Hebbian STDP. Only the synapses connected to our desired output neuron will undergo Hebbian STDP. All of the other synapses will undergo anti-Hebbian STDP. Equation 5, below, covers Hebbian STDP:

$$\Delta w_{ij} = \begin{cases} 0.1e^{\frac{-(|t_j - t_i|)}{\tau}} & t_j - t_i \geq 0 \\ 0.1e^{\frac{-(|t_j - t_i|)}{\tau}} & t_j - t_i < 0 \end{cases} \quad (5)$$

For both Hebbian and anti-Hebbian there are two cases; long-term potentiation (LTP) and long-term (LTD). In Hebbian STDP, if the postsynaptic spike occurs after the presynaptic spike, then it can be determined that the presynaptic spike has a direct effect on the output neuron's firing rate. As a result, the synaptic weight between these two neurons will increase (LTP). On the other hand, if an output neuron fires before it receives the presynaptic spike, then the synaptic weight between the two neurons decreases (LTD). For anti-Hebbian STDP the cases are swapped, so that the synapse undergoes LTD when the difference in time between the pre-and postsynaptic spikes is positive and undergoes LTP otherwise.

After the network is trained, we will feed the network one of each of the stop consonants and generate a base spike signature which our testing spike signatures will be compared against. For example, we will feed the network a 'b' sound and obtain the spike signature returned from the 'b' output neuron and this will be our prototype spike signature for a 'b' sound. This will be performed for each of the stop consonants until we have a prototype spike signature for each. A small portion of the dataset not used for training or prototype generation will be used for testing. To determine classification, we will compare the resulting spike signatures with the corresponding prototype signatures using Spike Synchronization calculated using equations 6 – 9 below. We generate coincidence indicators for each pair of spike trains (n, m) using equation 6:

$$C_i^{(n,m)} \begin{cases} 1 & if\ min_j \left( \left| t_i^{(n)} - t_j^{(m)} \right| \right) < \tau_{ij}^{(n,m)} \\ 0 & otherwise \end{cases} \quad (6)$$

where $\tau_{ij}$ is the coincidence detection and is defined as:

$$\tau_{ij} = \frac{\min\left\{t_{i+1}^x - t_i^x, t_i^x - t_{i-1}^x, t_{j+1}^y - t_j^y, t_j^y - t_{j-1}^y\right\}}{2} \quad (7)$$

where $x$ and $y$ correspond to the two spike trains being compared and $t_i$ and $t_j$ correspond to the spike times of spike $i$ and spike $j$ respectively. Then we generate a normalized coincidence counter for each spike of each spike train:

$$C_i^{(n)} = \frac{1}{N-1} \sum_{m \neq n} C_i^{(n,m)} \quad (8)$$

where $N$ is the number of coincidence indicators involving the spike train $n$.

We then average the coincidence counters of the total spikes from the total number of spike trains using the equation below:

$$S_C = \frac{1}{M}\sum_{k=1}^{M} C(t_k) \quad (9)$$

where M is the total number of spikes when combining all of the spike trains. The value returned from this equation will be a measure of how similar two spike trains are with a higher value corresponding to greater similarity and a low value corresponding to a greater dissimilarity.

## 3. Experimental/Modeling Design

### *Data Preparation*

The data we used to both train and test our model was obtained from the University of Pennsylvania's Linguistic Data Consortium. The database we obtained our data from is the ISOLET Spoken Letter Database which contained 7800 spoken letters sampled from 150 speakers, half male and half female, recorded as a .wav file. To convert our raw audio signal into a usable format for our model, we extracted the Mel-Frequency Cepstral Coefficients (MFCCs) and used these values as our features.

Initially we trained our network on 300 samples, 50 samples for each stop consonant. We found that this resulted in some output neurons, particularly the neuron associated with the 'g' sound to not generate any spikes regardless of the input signal. So, we reduced our training sample to 120 samples with 20 samples for each stop consonant.

### *Spike Generation*

To generate the input spikes, we used the *Izhikevich* regular spiking (RS) neuron to generate a spike train given an input current per the following equations. The parameters we used for our neuron model is given in table 1.

$$\frac{dV}{dt} = 0.04V^2 + 5V + 140 - u + I_{inj}$$

$$\frac{du}{dt} = a(bV - u)$$

Where $I_{inj}$ is the injected current and is calculated using equation 2.

The reset equation:

$$V > V_{peak}: \begin{cases} V = c \\ u = u + d \end{cases}$$

**Table 1:**

| Parameter | Value |
|---|---|
| a | 0.02 |
| b | 0.2 |
| c | -65 |

| | |
|---|---|
| d | 8 |
| u | -13 |
| $V_{peak}$ | 30 |

Each of our output neurons receives input from each of the 520 input neurons via a synapse. As a result, the injected current for the output neurons is determined differently than it is for the input neurons. Equations 3 and 4 are used to determine the input current for.

### *Network Architecture*

Our network consists of two layers of *Izhikevich* regular spiking (RS) neurons connected by trainable synapses as shown in figure 1. The first layer of our network is our input layer consisting of 520 input neurons with each neuron receiving one feature value as its injected current. The output layer consists of 6 neurons corresponding to each of the stop consonants. The output neurons contain the same parameter configuration as the input layer neurons. Each input neuron is connected to each of the output neurons via a trainable synapse resulting in each output neuron having 520 input synapses. The synapses were initialized with weights equal to 0.5. The training for these synapses is described by equation 5. Each of the output neurons generate a unique spike train based on the signal features given to the input layer. The spike trains are analyzed and used to classify the consonant that was fed into the network according to equations 6 - 9. Finally, there is a "teacher" signal that determines which neurons and their synapses undergo Hebbian and anti-Hebbian STDP. For a given signal, the respective output neuron undergoes Hebbain STDP while the others undergo anti-Hebbian STDP.

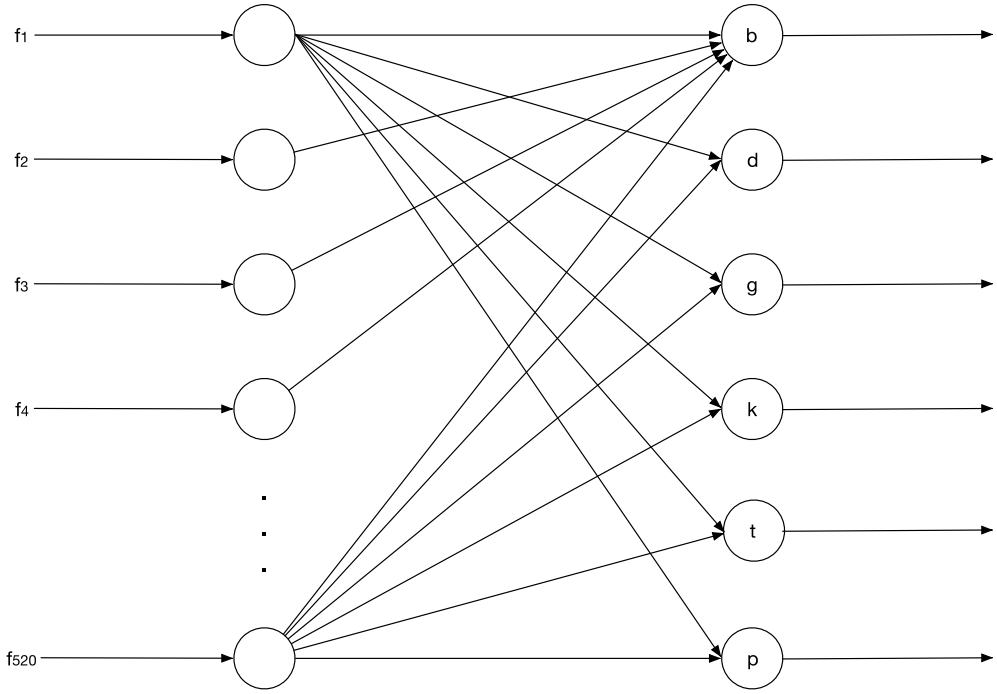


**Figure 1:** Network Diagram

### *Prototyping and Testing*

After training, we fed our network a signal for each of the stop consonants and obtained the corresponding spike trains for each. These spike trains were what our test results would be compared against. After each test, each of the output neurons generated spike trains based on the given input signal. Each spike train was compared against its respective prototype spike train using spike synchronization as described in equations 6 – 9. We describe a "hit" as the test spike signature having the greatest similarity to its respective prototype signature.

## 4. Results and Discussion

We used a portion of our dataset not used for training or prototype generation to test our network and generate results. Each consonant was tested a total of 10 times and the average synchronization between the test spike signatures and the prototype spike signatures is found in table 2. Both prototyping and testing were done for 500ms.

**Table 2:**

| Average Spike Synchronization compared to Prototypes | | | | | | |
|---|---|---|---|---|---|---|
| | Test Consonant | | | | | |
| **Output Neuron** | **B** | **D** | **G** | **K** | **P** | **T** |
| **B** | 0.91 | 0.74 | 0.95 | 0.78 | 0.84 | 0.84 |
| **D** | 0.86 | 0.76 | 0.93 | 0.81 | 0.87 | 0.83 |
| **G** | 0.86 | 0.8 | 0.93 | 0.73 | 0.82 | 0.73 |
| **K** | 0.93 | 0.86 | 0.92 | 0.82 | 0.93 | 0.87 |
| **P** | 0.77 | 0.79 | 0.90 | 0.76 | 0.76 | 0.97 |
| **T** | 0.82 | 0.86 | 0.94 | 0.81 | 0.81 | 0.71 |
| **Classification Ratio** | **.46** | **.2** | **.13** | **.49** | **0** | **.04** |

Comparison between the prototype spike signatures with the corresponding output signatures of a given sound. For example, when there is a 'B' consonant as the input, the 'D' output neuron has, on average, an 86% similarity with the prototype spike signature of a 'D' consonant.

**Table 3:** Results from a single test for each stop consonant.

| Single Test | | | | | | |
|---|---|---|---|---|---|---|
| | Randomly Selected Test | | | | | |
| **Output Neuron** | **B** | **D** | **G** | **K** | **P** | **T** |
| **B** | .99 | 0.73 | 0.94 | 0.75 | .99 | 0.67 |
| **D** | 0.91 | .98 | 0.94 | 0.86 | 0.67 | .99 |
| **G** | .99 | 0.8 | .98 | 0.75 | 0.75 | 0.4 |
| **K** | 0.91 | 0.8 | 0.94 | .98 | .99 | 0.67 |
| **P** | 0.83 | 0.8 | 0.88 | 0.86 | 0.89 | .99 |
| **T** | 0.91 | 0.89 | 0.94 | 0.86 | .99 | 0.8 |

Our model managed to produce distinct spike signatures for each of the spoken stop consonants as is evident in figures 2 and 3. However, the results displayed in tables 2 suggest that on average there was a significant amount of confusion in our network in that it produced spike signatures similar to the prototype spikes for consonants that were not the input.

The stop consonant ‘K’ was the only consonant that our network, on average, managed to produce a distinct pattern that allowed it to be distinguished from the other stop consonants. The difference however was by only 1%. The stop consonant ‘B’ managed to have an average synchronization of 91% but the spike signature generated from the ‘K’ output given a ‘B’ sound had, on average, a greater similarity with it the ‘K’ prototype spike signature. The ‘G’ stop consonant produced spike signatures for each of the output neurons that were very similar to their respective prototype signatures.

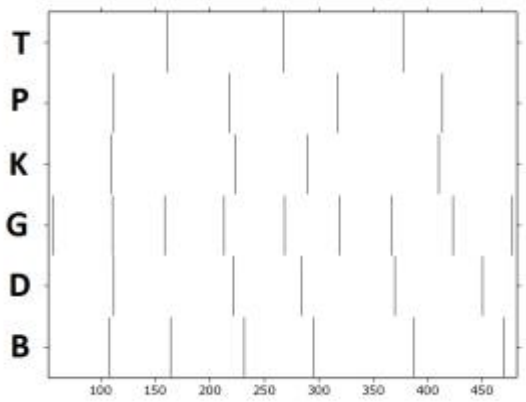


**Figure 2:** (left). Prototype spike train for each of the spoken stop consonants in a 500ms window.

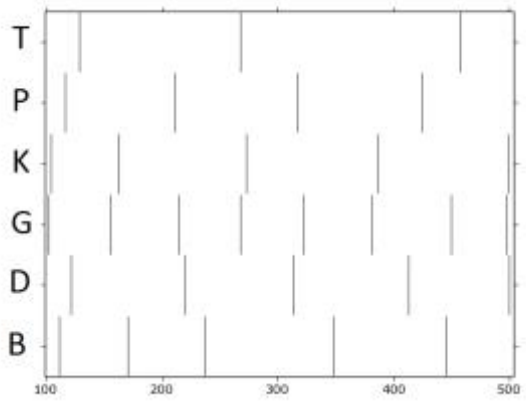


**Figure 3:** (right). A test spike train for each of the spoken stop consonants in a 500ms window

It is interesting to note that our model classified the consonants ‘B’ and ‘K’ with almost 50% accuracy whereas the other consonants performed very poorly by comparison (0 – 20% accuracy). It may be natural to assume that the ‘K’ consonant should be easy to distinguish, but our model still managed to get confuse and misclassify the ‘K’ consonant or classify other consonants as ‘K’. Furthermore, the test signatures of each stop consonant, on average, had a relatively low synchronization with their prototype spike signature. Ideally our model, given a certain input, should have generated spiking signatures similar to the corresponding prototype signature and signatures very different for each of the other outputs relative to their prototype signatures. One possible cause for this is that our equation to calculate the injected current wasn’t generating values large enough to generate a high enough spiking frequency in our output layer. By having a greater number of spikes and high firing rate, we would have a greater pool of samples to find similarities and differences. We rule out the possibility that this was a training issue since we had trained two instances of our model with two different training sample sizes,

one with 50 samples for each stop consonant (group A) and the other with 20 samples per consonant (group B). What we found that group A had similar performance to group B in that there wasn't enough variability in spiking patterns for the outputs. Additionally, regardless of the time window, the 'G' output neuron refused to fire despite some synaptic weights having a high value. It is important to note that our model still produced distinct firing patterns for each of the stop consonants, they just happened to firing a similar pattern regardless of what the input was.

## 5. Conclusion

Our model succeeds in generating distinct spike signatures for the spoken stop consonants but fails to accurately classify the output. The 'B' and 'K' stop consonants had an average success rate of about 50% while the other stop consonants (D, G, P, T) performed significantly worse with classification success ranging from 0 to 20%. We determined that the failings of our model were due to equations used for input current and not in the network architecture or the method of learning implemented. The findings of this study show that it is indeed very possible to have a minimal SNN that can generate distinct spike signatures that allow them to be correctly classified. Further tweaking and adjustment of our model can lead to greater success in recognizing spoken stop consonants and eventually spoken words or letters.

## Acknowledgments

These guidelines have grown out of an outline prepared by Prof. Konstantinos Michmizos for 443/674. The authors gratefully acknowledge helpful discussions with Prof. Konstantinos Michmizos of the Department of Computer Science at Rutgers University. We would also like to thank and acknowledge the helpful discussions with Amirhossein Tavanaei, Ph.D. student of computer science at the University of Louisiana at Lafayette.

## References

[1] Tavanaei, A., Maida, A. S., 2017, "A Spiking Network that Learns to Extract Spike Signatures from Speech Signals", Neurocomputing, 240, 191-199

[2] Esposito, A., Ezin, C. E., Ceccarelli, M., 1996, "Preprocessing and Neural Classification of English Stop Consonants", ICSLP 96. Proceedings, vol. 2, 1249-1252

## Appendix

### *Neuron.py*

```
import numpy as np
import random
import Synapse

global Pref, Pmin, Pth, D, Prest, pre_times, post_times, synapses
global out_synapses
global in_synapses
Pref = 0
Prest = 0
Pmin = -1
Pth = 5.5
D = 0.5


class Neuron:
    def __init__(self):
        self.Pth = Pth
        self.t_ref = 4
        self.t_rest = -1
        self.P = Prest
        self.D = D
        self.Pmin = Pmin
        self.Prest = Prest
        self.post_times = []
        self.pre_times = []
        self.synapses = []
        self.in_synapses = []
        self.out_synapses = []

    def append_pre_times(self, times):
        self.pre_times = times

    def append_post_times(self, times):
        self.post_times = times

    def append_synapse(self, synapse):
        self.synapses.append(synapse)

    def append_in_synapse(self, synapse):
        self.in_synapses.append(synapse)

    def append_out_synapse(self, synapse):
        self.out_synapses.append(synapse)

    def output_izh_simulation(self, a, b, c, d, time_ita, current,
v_init):
        # a,b,c,d parameters for Izhikevich model
        # time_ita time iterations for euler method
        # current list of current for each time step
        # v_init initial voltage
        spike_times = []
```

```
        v = v_init
        u = v * b
        v_plt = np.zeros(time_ita)
        u_plt = np.zeros(time_ita)
        spike = np.zeros(time_ita)
        num_spikes = 0
        tstep = 0.1  # ms
        ita = 0
        while ita < time_ita:
            if ita < 200:
                v_plt[ita] = v_init
            else:
                v_plt[ita] = v
            u_plt[ita] = u
            v += tstep * (0.04 * (v ** 2) + 5 * v + 140 - u +
current[ita])
            u += tstep * a * (b * v - u)
            if v > 30.:
                if ita > 200:
                    spike[ita] = 1
                    num_spikes += 1
                v = c
                u += d

                # spike_times.append(ita)

            ita += 1
        time = np.arange(time_ita) * tstep
        i = 0
        for t in time:
            if spike[i] == 1:
                spike_times.append(t)
            i += 1
        return time, v_plt, spike, num_spikes, spike_times

    def izh_simulation(self, a, b, c, d, time_ita, current, v_init):
        # a,b,c,d parameters for Izhikevich model
        # time_ita time iterations for euler method
        # current list of current for each time step
        # v_init initial voltage
        spike_times = []
        v = v_init
        u = v * b
        v_plt = np.zeros(time_ita)
        u_plt = np.zeros(time_ita)
        spike = np.zeros(time_ita)
        num_spikes = 0
        tstep = 0.1  # ms
        ita = 0
        while ita < time_ita:
            v_plt[ita] = v
            u_plt[ita] = u
            v += tstep * (0.04 * (v ** 2) + 5 * v + 140 - u +
current[ita])
```

```
            u += tstep * a * (b * v - u)
            if v > 30.:
                spike[ita] = 1
                v = c
                u += d
                num_spikes += 1
                #spike_times.append(ita)

            ita += 1
        time = np.arange(time_ita) * tstep
        i = 0
        for t in time:
            if spike[i] == 1:
                spike_times.append(t)
            i += 1
        return time, v_plt, spike, num_spikes, spike_times
```

### *Synapse.py*

```
import numpy as np
import random as random


class Synapse:

    global spike, time, conductance_amplitude, w, spikes_received,
pre_spikes, post_spikes, input_neuron, out_neuron, voltage

    def __init__(self):
        self.post_spikes = []
        self.pre_spikes = []
        self.w = .5

    def set_weight(self, w):
        self.w = float(w)

    def set_voltage(self, voltage):
        self.voltage = voltage

    def set_input_neuron(self, neuron):
        self.input_neuron = neuron

    def set_out_neuron(self, neuron):
        self.out_neuron = neuron

    def set_time(self, time):
        self.time = time

    def set_spike(self, spike):
        self.spike = spike

    def set_pre_spikes(self, pre_spikes):
        self.pre_spikes = pre_spikes

    def set_post_spikes(self, post_spikes):
```

```
        self.post_spikes = post_spikes

    # Our W function for Hebbian STDP
    def synaptic_weight_func(self, delta_t):
        tau_pre = 20
        tau_post = 20
        Apre = .10
        Apost = -Apre
        if delta_t >= 0:
            return Apre*np.exp(-np.abs(delta_t)/tau_pre)
        if delta_t < 0:
            return Apost*np.exp(-np.abs(delta_t)/tau_post)

    # Our W function for anti-Hebbian STDP
    def anti_heb(self, delta_t):
        tau_pre = 20
        tau_post = 20
        Apre = .10
        Apost = -Apre
        if delta_t < 0:
            return Apre*np.exp(-np.abs(delta_t)/tau_pre)
        if delta_t >= 0:
            return Apost*np.exp(-np.abs(delta_t)/tau_post)

    # Same as Hebbian STDP except the cases are reversed
    def Anti_Heb_STDP(self):
        delta_w = 0
        for t_pre in self.pre_spikes:
            for t_post in self.post_spikes:
                delta_w += self.anti_heb(t_post - t_pre)
        self.w += delta_w
        if self.w < 0:
            self. w = 0


    # Change in synaptic weight is the sum over all presynaptic spike
times (t_pre) and postsynaptic spike times (t_post)
    # of some function W of the difference in these spike times
    def Heb_STDP(self):
        delta_w = 0
        for t_pre in self.pre_spikes:
            for t_post in self.post_spikes:
                delta_w += self.synaptic_weight_func(t_post - t_pre)
        self.w += delta_w
        if self.w < 0:
            self.w = 0


    # Calculates synaptic output
    def synapse(self, tau):
        synapse_output = np.zeros(len(self.time))
        for t in range(len(self.time)):
            tmp_time = self.time[t] - self.time[0:t]
```

```
            synapse_output[t] = np.sum(((tmp_time * self.spike[0:t]) /
tau) * np.exp(-(tmp_time * self.spike[0:t]) / tau))
        return self.w * synapse_output

    def synapse_func(self, tau):
        time = np.arange(10000) * 0.1
        func = time / tau * np.exp(-time / tau)
        return time, func
```

### *Network.py*

```
import numpy as np
import Synapse
import Neuron
import Utils
from matplotlib import pyplot as plt
from neuronpy.graphics import spikeplot

global input_layer
global output_layer
global hidden_layer
global synapses
global a, b, c, d, time_ita

class Network:

    def __init__(self, weights):
        self.a = 0.02
        self.b = 0.2
        self.c = -65.
        self.d = 8.
        self.time_ita = 5000  # 100ms

        # Build a layer of 120 input neurons (20 frames, 6 features for
each frame)
        self.input_layer = []
        for i in range(520):
            n = Neuron.Neuron()
            self.input_layer.append(n)

        # Build output layer, one neuron for each letter of the alphabet
        self.output_layer = []
        for i in range(6):
            o = Neuron.Neuron()
            self.output_layer.append(o)

        i = 0
        # For each input neuron, append one synapse to each output neuron
        for n in self.input_layer:
            for out in self.output_layer:
                synapse = Synapse.Synapse()
                n.append_synapse(synapse)
                out.append_synapse(synapse)

        if weights is not None:
```

```
            i = 0
            n = 0
            s = 0
            with open(weights) as f:
                for line in f:
                    # For every 520 synapses, go to the next neuron
                    if i % 520 == 0 and n < len(self.output_layer):
                        s = 0
                        out = self.output_layer[n]
                        n += 1
                        out.synapses[s].set_weight(line)
                        s += 1
                    elif s < 520:
                        out.synapses[s].set_weight(line)
                        s += 1
                    i += 1

    # Calculates the current from the given MFCC value
    def get_current(self, x):
        if x == 0:
            return 8.5
        elif x < 0:
            return (.0530914398 * x) + 8.5
        else:
            return (.11805555555 * x) + 8.5


    # Get the total synaptic output for this neuron
    def total_synaptic_value(self, neuron):
        conductance = 0
        for syn_k in neuron.synapses:
            output = syn_k.synapse(2)
            conductance += output
        return conductance

    # if result == 0, then our target neuron is the first neuron in the
output layer
    # result == 1 --> 2nd output neuron, result == 3 --> 3rd output neuron
and so on
    def conduct_training(self, result):
        i = 0
        for out in self.output_layer:
            if i == result:
                # Undergo Hebbian STDP
                for syn in out.synapses:
                    syn.Heb_STDP()
            else:
                # Undergo anti-Hebbian STDP for non-target synapses
                for syn in out.synapses:
                    syn.Anti_Heb_STDP()
            i += 1

    def conduct_specific_training(self, result):
        i = 0
```

```
        for out in self.output_layer:
            if i == result:
                for syn in out.synapses:
                    syn.Heb_STDP
            i += 1

    # Perform analysis on the given filename using mel_Freq command
    def start(self, fname):

        features = Utils.get_mel(fname)
        features = features[:520]

        #Feed features into our network and get spike information (number
of spikes, time of largest spike)
        i = 0

        # Use for mel_freq. 520 input neurons
        for feature in features:
            n = self.input_layer[i]
            current = np.ones(self.time_ita) * self.get_current(feature)
            time, v_plt, spike, num_spikes, spike_times =
n.izh_simulation(self.a,self.b,self.c,self.d,self.time_ita, current,
self.c)
            # Set pre spikes for each synapse connected to this neuron
            for synapse in n.synapses:
                synapse.set_pre_spikes(spike_times)
                synapse.set_time(time)
                synapse.set_spike(spike)
            i += 1

        # Create a 3 neuron output vector
        outputs = [0] * 6
        spikes = []
        v_plts = []
        currents = []
        i = 0
        for out in self.output_layer:
            current = np.ones(self.time_ita) *
self.total_synaptic_value(out)
            time, v_plt, spike, num_spikes, spike_times =
out.output_izh_simulation(self.a,self.b,self.c,self.d,self.time_ita,
current, self.c)
            spikes.append(spike_times)
            v_plts.append(v_plt)
            currents.append(current)
            for syn in out.synapses:
                syn.set_post_spikes(spike_times)

            outputs[i] = num_spikes
            i += 1

        return outputs, currents, time, v_plts, spikes
```

***SNN.py***

```
from random import shuffle
import os
import sys
import Utils
import Network
import pyspike as spk
from pyspike import SpikeTrain
from datetime import datetime
from matplotlib import pyplot as plt
from neuronpy.graphics import spikeplot

prototype_trains = [None] * 6
test_trains = [None] * 6

def write_weights(network):
    i = 0
    with open("lessWeight.txt", "a") as f:
        for out in network.output_layer:
            if i == 0:
                f.write("B\n")
            elif i == 1:
                f.write("D\n")
            elif i == 2:
                f.write("G\n")
            elif i == 3:
                f.write("K\n")
            elif i == 4:
                f.write("P\n")
            elif i == 5:
                f.write("T\n")
            i += 1

def print_result(results):
    print('\tB: ' + str(results[0]))
    print('\tD: ' + str(results[1]))
    print('\tG: ' + str(results[2]))
    print('\tK: ' + str(results[3]))
    print('\tP: ' + str(results[4]))
    print('\tT: ' + str(results[5]))

# Generate a spike train from the given spike
def generate_prototypes(spike, key):
    global prototype_trains
    spike_train = SpikeTrain(spike, [0.0, 500.0])
    if key == 'B':
        prototype_trains[0] = spike_train
    elif key == 'D':
        prototype_trains[1] = spike_train
    elif key == 'G':
        prototype_trains[2] = spike_train
    elif key == 'K':
        prototype_trains[3] = spike_train
    elif key == 'P':
```

```
        prototype_trains[4] = spike_train
    elif key == 'T':
        prototype_trains[5] = spike_train


def generate_test_signatures(spike, key):
    global test_trains
    spike_train = SpikeTrain(spike, [0.0, 500.0])
    if key == 'B':
        test_trains[0] = spike_train
    elif key == 'D':
        test_trains[1] = spike_train
    elif key == 'G':
        test_trains[2] = spike_train
    elif key == 'K':
        test_trains[3] = spike_train
    elif key == 'P':
        test_trains[4] = spike_train
    elif key == 'T':
        test_trains[5] = spike_train


def spike_analysis(spikes):
    distances = []
    letters = ['B', 'D', 'G', 'K', 'P', 'T']
    i = 0
    for spike in spikes:
        spike_train = SpikeTrain(spike, [0.0, 500.0])
        isi_profile = spk.spike_sync(prototype_trains[i], spike_train)
        print("\t%s: " % letters[i] + str(isi_profile))
        distances.append(isi_profile)
        i += 1

    val, idx = max((val, idx) for (idx, val) in enumerate(distances))
    print("Closest Distance: %.8f" % val)
    print("Index: %s" % letters[idx])


def show_plots(time, v_plts, currents, spikes):
    plt.figure('B')
    plt.plot(time, v_plts[0], 'g-')
    plt.figure('D')
    plt.plot(time, v_plts[1], 'b-')
    plt.figure('G')
    plt.plot(time, v_plts[2], 'k-')
    plt.figure('K')
    plt.plot(time, v_plts[3], 'r-')
    plt.figure('P')
    plt.plot(time, v_plts[4], 'm-')
    plt.figure('T')
    plt.plot(time, v_plts[5], 'c-')
    sp = spikeplot.SpikePlot()
    sp.plot_spikes(spikes)
    plt.show()
```

```
# Test our network
def test():
    global prototype_trains
    global test_trains
    prototype_trains = [None] * 6
    test_trains = [None] * 6
    mapping = dict()

    weights = "lessWeight.txt"

    # Build our trained network
    network = Network.Network(weights=weights)

    # Build our prototype audio
    audio_path = "letter_audio/speech/isolet5"
    audio = [os.path.join(root, name)
                for root, dirs, files in os.walk(audio_path)
                for name in files
                if name.endswith((".wav"))]

    for fname in audio:
        mapping[fname] = Utils.get_label(fname)

    # Build our test audio
    audio_path = "letter_audio/speech/isolet4"
    audio = [os.path.join(root, name)
                for root, dirs, files in os.walk(audio_path)
                for name in files
                if name.endswith((".wav"))]

    test_dict = dict()
    for fname in audio:
        test_dict[fname] = Utils.get_label(fname)

    b_count = 1
    d_count = 1
    g_count = 1
    k_count = 1
    p_count = 1
    t_count = 1

    count = 6

    # Generate our prototype spike trains
    for key in mapping:
        if mapping[key] == 'B' and b_count != 0:
            print(key)
            results, currents, time, v_plts, spikes = network.start(key)
            print_result(results)
            b_count -= 1
            count -= 1
            if b_count == 0:
                # Generate a spike train for the 'B' sound
```

```
            generate_prototypes(spikes[0], 'B')
        elif mapping[key] == 'D' and d_count != 0:
            print(key)
            results, currents, time, v_plts, spikes = network.start(key)
            print_result(results)
            d_count -= 1
            count -= 1
            if d_count == 0:
                # Generate a spike train for the 'D' sound
                generate_prototypes(spikes[1], 'D')
        elif mapping[key] == 'G' and g_count != 0:
            print(key)
            results, currents, time, v_plts, spikes = network.start(key)
            print_result(results)
            g_count -= 1
            count -= 1
            if g_count == 0:
                # Generate a spike train for the 'G' sound
                generate_prototypes(spikes[2], 'G')
        elif mapping[key] == 'K' and k_count != 0:
            print(key)
            results, currents, time, v_plts, spikes = network.start(key)
            print_result(results)
            k_count -= 1
            count -= 1
            if k_count == 0:
                # Generate a spike train for the 'K' sound
                generate_prototypes(spikes[3], 'K')
        elif mapping[key] == 'P' and p_count != 0:
            print(key)
            results, currents, time, v_plts, spikes = network.start(key)
            print_result(results)
            p_count -= 1
            count -= 1
            if p_count == 0:
                # Generate a spike train for the 'P' sound
                generate_prototypes(spikes[4], 'P')
        elif mapping[key] == 'T' and t_count != 0:
            print(key)
            results, currents, time, v_plts, spikes = network.start(key)
            print_result(results)
            t_count -= 1
            count -= 1
            if t_count == 0:
                # Generate a spike train for the 'T' sound
                generate_prototypes(spikes[5], 'T')


    # Display our prototype spike trains
    sp = spikeplot.SpikePlot()
    sp.plot_spikes(prototype_trains, label="Prototypes")

    b_count = 10
    d_count = 10
```

```
    g_count = 10
    k_count = 10
    p_count = 10
    t_count = 10

    for key in test_dict:
        if test_dict[key] == 'B' and b_count != 0:
            print("Testing %s" % key)
            results, currents, time, v_plts, spikes = network.start(key)
            print_result(results)
            b_count -= 1
            spike_analysis(spikes)
            if b_count == 0:
                # Generate a spike train for the 'B' sound and find a
match
                generate_test_signatures(spikes[0], 'P')
        elif test_dict[key] == 'D' and d_count != 0:
            print("Testing %s" % key)
            results, currents, time, v_plts, spikes = network.start(key)
            print_result(results)
            d_count -= 1
            spike_analysis(spikes)
            if d_count == 0:
                # Generate a spike train for the 'D' sound and find a
match
                generate_test_signatures(spikes[1], 'D')
        elif test_dict[key] == 'G' and g_count != 0:
            print("Testing %s" % key)
            results, currents, time, v_plts, spikes = network.start(key)
            print_result(results)
            g_count -= 1
            spike_analysis(spikes)
            if g_count == 0:
                # Generate a spike train for the 'G' sound and find a
match
                generate_test_signatures(spikes[2], 'G')
        elif test_dict[key] == 'K' and k_count != 0:
            print("Testing %s" % key)
            results, currents, time, v_plts, spikes = network.start(key)
            print_result(results)
            k_count -= 1
            spike_analysis(spikes)
            if k_count == 0:
                # Generate a spike train for the 'K' sound and find a
match
                generate_test_signatures(spikes[3], 'K')
        elif test_dict[key] == 'P' and p_count != 0:
            print("Testing %s" % key)
            results, currents, time, v_plts, spikes = network.start(key)
            print_result(results)
            p_count -= 1
            spike_analysis(spikes)
            if p_count == 0:
```

```
                # Generate a spike train for the 'P' sound and find a
match
                generate_test_signatures(spikes[4], 'P')
        elif test_dict[key] == 'T' and t_count != 0:
            print("Testing %s" % key)
            results, currents, time, v_plts, spikes = network.start(key)
            print_result(results)
            t_count -= 1
            spike_analysis(spikes)
            if t_count == 0:
                # Generate a spike train for the 'T' sound and find a
match
                generate_test_signatures(spikes[5], 'T')

    # Display our last round of test spikes
    sp = spikeplot.SpikePlot()
    sp.plot_spikes(test_trains, label="Test")


# Train the network
def train():
    network = Network.Network(weights=None)

    mapping = dict()

    audio_path = "letter_audio/speech/isolet1"

    # Gets list of all audio files in the directory
    audio = [os.path.join(root, name)
             for root, dirs, files in os.walk(audio_path)
             for name in files
             if name.endswith((".wav"))]

    audio_path = "letter_audio/speech/isolet2"
    audio2 = [os.path.join(root, name)
              for root, dirs, files in os.walk(audio_path)
              for name in files
              if name.endswith((".wav"))]

    audio.extend(audio2)

    shuffle(audio)

    # Get a mapping of labels to audio
    for fname in audio:
        mapping[fname] = Utils.get_label(fname)

    print(datetime.now())

    b_count = 20
    d_count = 20
    g_count = 20
    k_count = 20
    p_count = 20
```

```
    t_count = 20

    for key in mapping:
        if mapping[key] == 'B' and b_count > 0:
            print(key)
            results, currents, time, v_plts, spikes = network.start(key)
            print_result(results)
            network.conduct_training(0)
            b_count -= 1
        elif mapping[key] == 'D' and d_count > 0:
            print(key)
            results, currents, time, v_plts, spikes = network.start(key)
            print_result(results)
            network.conduct_training(1)
            d_count -= 1
        elif mapping[key] == 'G' and g_count > 0:
            print(key)
            results, currents, time, v_plts, spikes = network.start(key)
            print_result(results)
            network.conduct_training(2)
            g_count -= 1
        elif mapping[key] == 'K' and k_count > 0:
            print(key)
            results, currents, time, v_plts, spikes = network.start(key)
            print_result(results)
            network.conduct_training(3)
            k_count -= 1
        elif mapping[key] == 'P' and p_count > 0:
            print(key)
            results, currents, time, v_plts, spikes = network.start(key)
            print_result(results)
            network.conduct_training(4)
            p_count -= 1
        elif mapping[key] == 'T' and t_count > 0:
            print(key)
            results, currents, time, v_plts, spikes = network.start(key)
            print_result(results)
            network.conduct_training(5)
            t_count -= 1

    write_weights(network)
    print(datetime.now())


if __name__ == "__main__":
    if len(sys.argv) > 1:
        if sys.argv[1] == 'train':
            print('Training...')
            train()
        else:
            print('Testing')
            test()
```