CS 443/525 Homework 2

Sakib Jalal

November 27, 2017

1 XOR

1.1 Temporal Encoding

I would represent the inputs with a temporal encoding by employing a time-to-first-spike encoding with equal scale for input and output. An input of 1 could translate to the corresponding input neuron spiking at $t = 0ms$ and an input of 0 could translate to the corresponding input neuron spiking at $t = 6ms$. Once a neuron fires, it would not be able to fire again for the same round of inputs. The output would then also be compared on a time-to-first-spike basis, with 0 being distinct from 1.

1.2 Firing Rate Encoding

I would represent the inputs with a firing rate encoding by employing a simple spike-count rate with similar scales for input and output. An input of 1 could translate to a high spike-count rate and an input of 0 could translate to a low spike-count rate on intervals of $t = 100ms$. The outputs would then output at either high or low spike-count rates which yield 1 or 0 outputs, respectively.

1.3 SNN Implementation

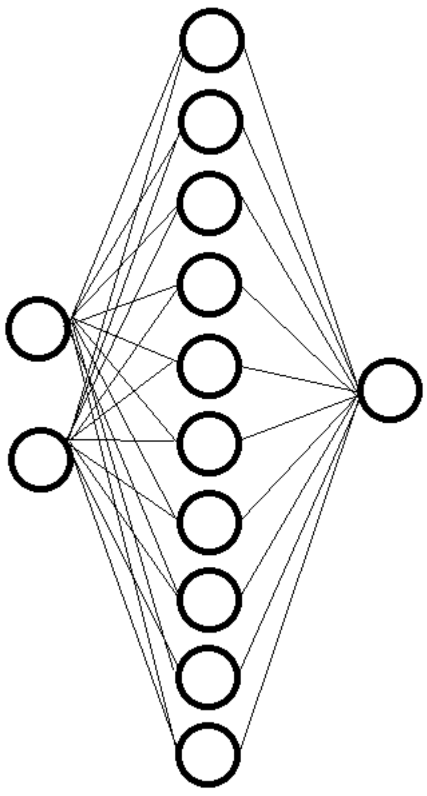
My Spiking Neural Net had three layers using LIF (Leaky-Integrate-and-Fire) neurons. The first layer had two neurons, one for each input to the XOR function, and the last layer had one neuron corresponding to the output of the XOR function.

I gave the hidden layer 10 LIF neurons. Depending on the type of learning I was implementing, the neuron types and input/output encodings were different. For Hebbian Learning, the neurons were all excitatory and the input/output encoding was based on firing rates of neurons. for STDP Learning, 8 of the neurons were excitatory and 2 were inhibitory and the input/output encoding was based on spike timing.

I initialized all of my LIF neurons with the following values: capacitance: 2.5, resistance: 2, threshold voltage: 75, rest voltage: 10, type: excitatory (by default).

I implemented the SNN in Python by keeping a list of layers, which themselves are lists of aforementioned LIF neurons. I also kept a list of synapse matrices where rows were indexed by the index of the pre-synaptic neuron in its respective layer and columns were indexed by the index of the post-synaptic neuron in its respective layer. I initialized these in the range [0, 1].

To have neurons communicate spikes to each other, I kept a constant input current feeding the input neurons (depending on the input encodings) and on every timestep, I would iterate over all neurons in each layer from left to right and check the corresponding previous layers, if they existed, for spiking pre-synaptic neurons. If a pre-synaptic neuron spiked, I induced an excitatory post-synaptic potential in the post-synaptic neuron by scaling an exponential function by the synaptic weight and the neuron's voltage threshold. I would sum these exponentials together (which decayed as time went on) and feed that as a current into the post-synaptic neuron.



1.4 Hebbian Learning

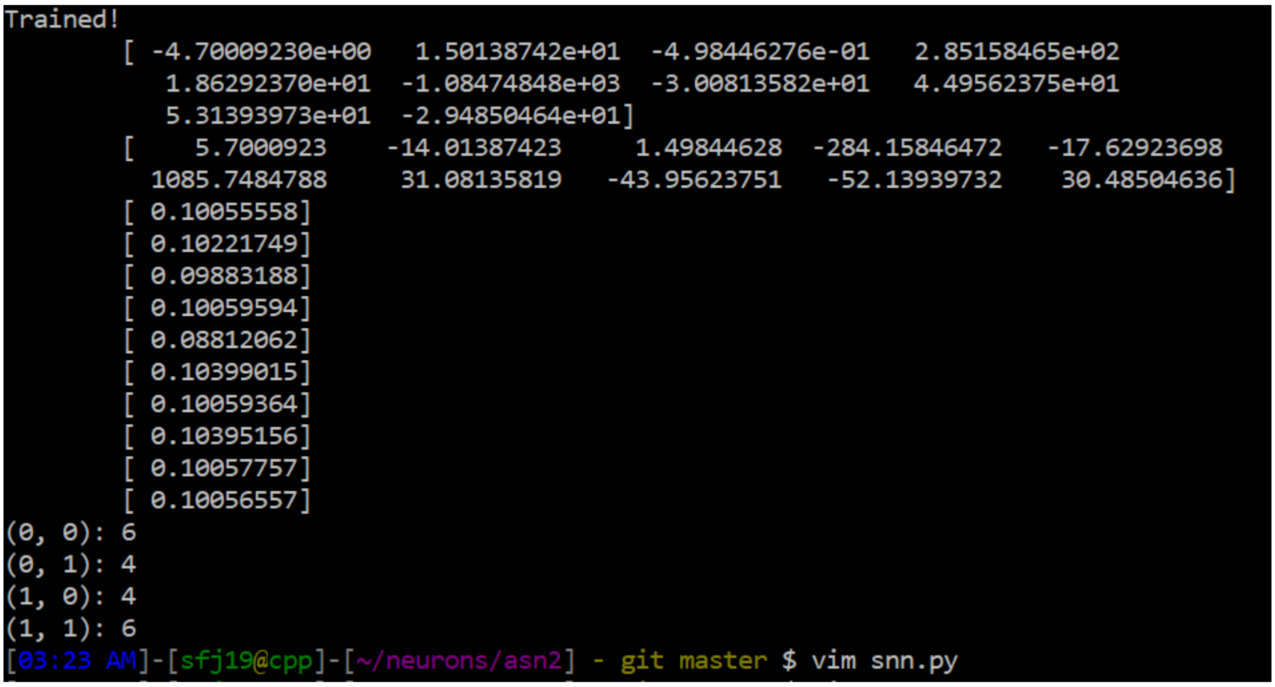
To implement Hebbian Learning, I defined a time window in the SNN class and encoded the inputs with neuronal firing rates, or number of spikes per time window. For example, to encode a 0, I would have an input neuron fire 12 times in a time window of 50 timesteps (milliseconds), and to encode a 1, I would have an input neuron fire 16 times in the same time window of 50 timesteps (milliseconds). I noted, interestingly, that some firing rates could not be encoded into the network with currents inputted to LIF neurons - for example, there was a jump from a firing rate of 25 (for input current 5.2) to a firing rate of 50 (for input current 5.3).

I trained the network on the set of inputs $[0, 0], [0, 1], [1, 0], [1, 1]$ and, as I described above, would keep the SNN running for a number of timesteps, only pausing every *timewindow* milliseconds to update the weights of all synapses according to Oja's rule (Oja, 1982):

$$\frac{dw_{ij}}{dt} = \gamma(v_i v_j - w_{ij} v_i^2)$$

where v_i was the firing rate activity of the i^{th} pre-synaptic neuron, v_j was the firing rate activity of the j^{th} post-synaptic neuron, and w_{ij} was the weight from the two neurons. I chose this rule because it introduces competition between pre-synaptic neurons by converging to yield synaptic weights that are normalized to:

$$\sum_j w_{ij}^2 = 1$$



```
Trained!
        [ -4.70009230e+00   1.50138742e+01  -4.98446276e-01   2.85158465e+02
           1.86292370e+01  -1.08474848e+03  -3.00813582e+01   4.49562375e+01
           5.31393973e+01  -2.94850464e+01]
        [    5.7000923    -14.01387423     1.49844628  -284.15846472   -17.62923698
          1085.7484788     31.08135819   -43.95623751   -52.13939732    30.48504636]
        [ 0.10055558]
        [ 0.10221749]
        [ 0.09883188]
        [ 0.10059594]
        [ 0.08812062]
        [ 0.10399015]
        [ 0.10059364]
        [ 0.10395156]
        [ 0.10057757]
        [ 0.10056557]
(0, 0): 6
(0, 1): 4
(1, 0): 4
(1, 1): 6
[03:23 AM]-[sfj19@cpp]-[~/neurons/asn2] - git master $ vim snn.py
```

This shows an example of the results of the training: the outputs of the network are displayed along with the initial weight matrices that yielded them. The first two arrays correspond to the weights between the input layer and the hidden layer, and the final ten arrays correspond to the weights between the neurons of the hidden layer and the output neuron. Evidently, some neurons grew to be more significant than others, especially since weights were not normalized. The fact that input pairs (0,0) and (1,1) yield the same rate of 6 output spikes per time window and input pairs (0,1) and (1,0) yield the same rate of 4 output spikes per time window demonstrates the nonlinearity of the network's approximation of the XOR function - if the output 6 is decoded as a 0 and the output 4 is decoded as 1, the XOR function exactly is yielded. I noted that adding more neurons to the hidden layer made this result pattern more stable.

1.5 STDP Learning

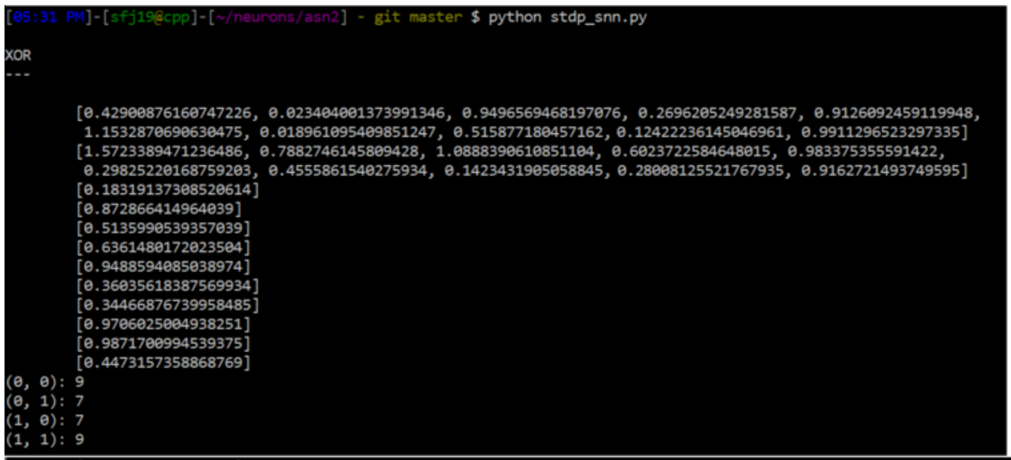
To implement Spike-Timing Dependent Plasticity, I kept 5 excitatory LIF neurons and 5 inhibitory LIF neurons in the hidden layer. My time constant was 3, my learning rate was 7, my tau values were 4, and my A values were $+/-weight/4$. I implemented pair-based STDP learning by treating synapses as objects which were able to update their own weights on every time step by first updating their pre-synaptic and post-synaptic traces according to these three equations:

$$\frac{dx_j}{dt} = -\frac{x_j}{\tau_+} + \sum_f \delta(t - t_j^f)$$

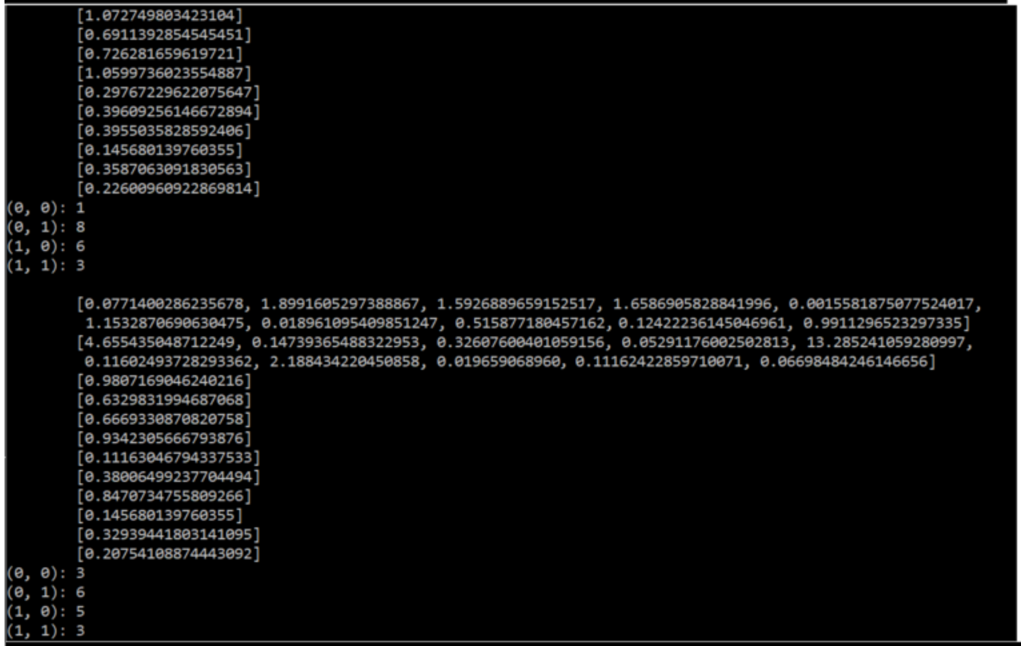
$$\frac{dy_i}{dt} = -\frac{y_i}{\tau_-} + \sum_f \delta(t - t_i^f)$$

$$\frac{dw_{ij}}{dt} = A_-(w_{ij})y_i(t)\sum_f \delta(t - t_j^f) + A_+(w_{ij})x_j(t)\sum_f \delta(t - t_i^f)$$

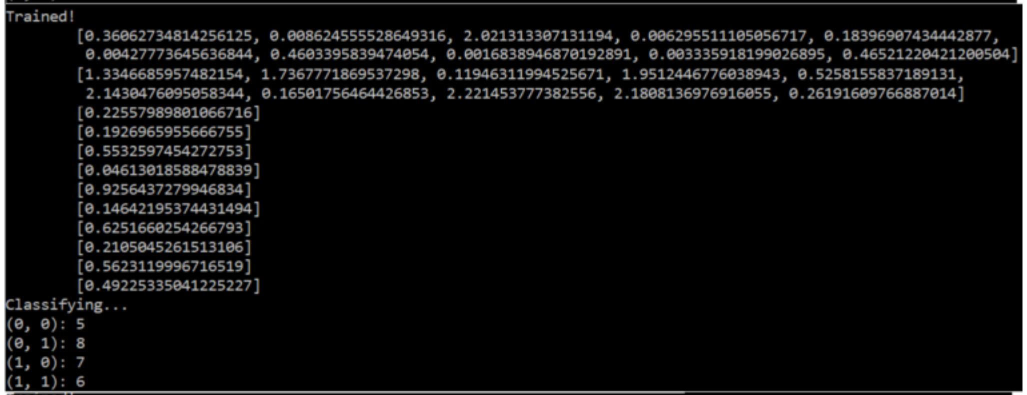
I encoded the inputs with a neuronal time-to-first-spike encodings. For example, to encode a 0, I would have an input neuron spike for the first time at t=6ms, and to encode a 1, I would have an input neuron spike for the first time at t=3ms. I kept the network running for a variable number of timesteps and would update the weights of all synapses after processing the spikes and updating the potentials for all neurons.



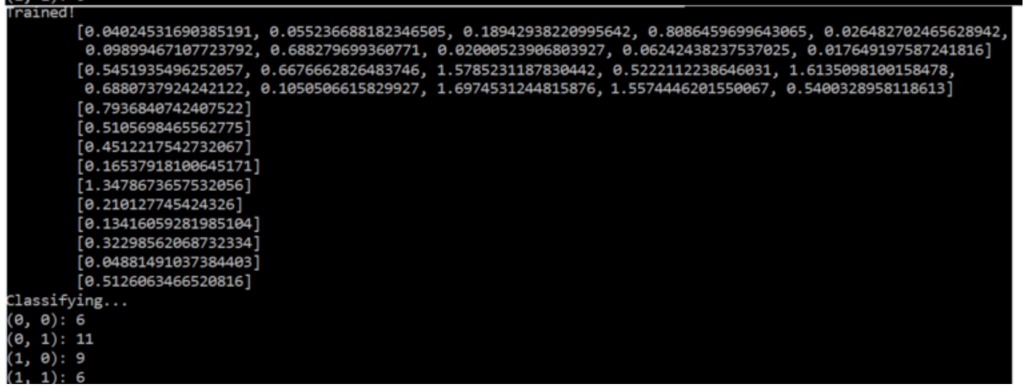
```
[05:31 PM]-[sfj19@cpp]-[~/neurons/asn2] - git master $ python stdp_snn.py

XOR
---

        [0.42900876160747226, 0.023404001373991346, 0.9496569468197076, 0.2696205249281587, 0.9126092459119948,
         1.1532870690630475, 0.018961095409851247, 0.515877180457162, 0.12422236145046961, 0.9911296523297335]
        [1.5723389471236486, 0.7882746145809428, 1.0888390610851104, 0.6023722584648015, 0.983375355591422,
         0.29825220168759203, 0.4555861540275934, 0.1423431905058845, 0.28008125521767935, 0.9162721493749595]
        [0.18319137308520614]
        [0.872866414964039]
        [0.5135990539357039]
        [0.6361480172023504]
        [0.9488594085038974]
        [0.36035618387569934]
        [0.34466876739958485]
        [0.9706025004938251]
        [0.9871700994539375]
        [0.4473157358868769]
(0, 0): 9
(0, 1): 7
(1, 0): 7
(1, 1): 9
```



```
        [1.072749803423104]
        [0.6911392854545451]
        [0.726281659619721]
        [1.0599736023554887]
        [0.29767229622075647]
        [0.39609256146672894]
        [0.3955035828592406]
        [0.145680139760355]
        [0.3587063091830563]
        [0.22600960922869814]
(0, 0): 1
(0, 1): 8
(1, 0): 6
(1, 1): 3

        [0.0771400286235678, 1.8991605297388867, 1.5926889659152517, 1.6586905828841996, 0.0015581875077524017,
         1.1532870690630475, 0.018961095409851247, 0.515877180457162, 0.12422236145046961, 0.9911296523297335]
        [4.655435048712249, 0.14739365488322953, 0.32607600401059156, 0.05291176002502813, 13.285241059280997,
         0.11602493728293362, 2.188434220450858, 0.019659068960, 0.11162422859710071, 0.06698484246146656]
        [0.9807169046240216]
        [0.6329831994687068]
        [0.6669330870820758]
        [0.9342305666793876]
        [0.11163046794337533]
        [0.38006499237704494]
        [0.8470734755809266]
        [0.145680139760355]
        [0.32939441803141095]
        [0.20754108874443092]
(0, 0): 3
(0, 1): 6
(1, 0): 5
(1, 1): 3
```



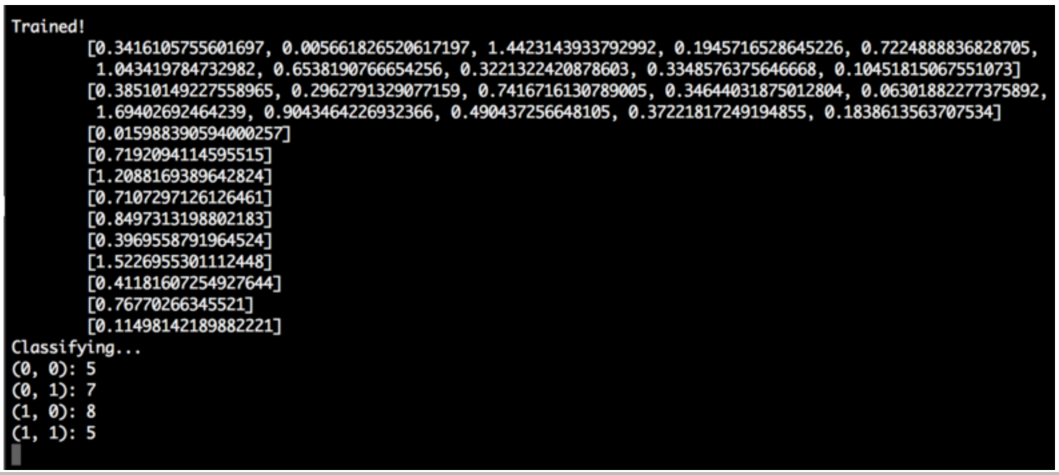
```
Trained!
        [0.36062734814256125, 0.008624555528649316, 2.021313307131194, 0.006295511105056717, 0.18396907434442877,
         0.00427773645636844, 0.4603395839474054, 0.0016838946870192891, 0.003335918199026895, 0.46521220421200504]
        [1.3346685957482154, 1.7367771869537298, 0.11946311994525671, 1.9512446776038943, 0.5258155837189131,
         2.1430476095058344, 0.16501756464426853, 2.221453777382556, 2.1808136976916055, 0.26191609766887014]
        [0.22557989801066716]
        [0.1926965955666755]
        [0.5532597454272753]
        [0.04613018588478839]
        [0.9256437279946834]
        [0.14642195374431494]
        [0.6251660254266793]
        [0.2105045261513106]
        [0.5623119996716519]
        [0.49225335041225227]
Classifying...
(0, 0): 5
(0, 1): 8
(1, 0): 7
(1, 1): 6
```



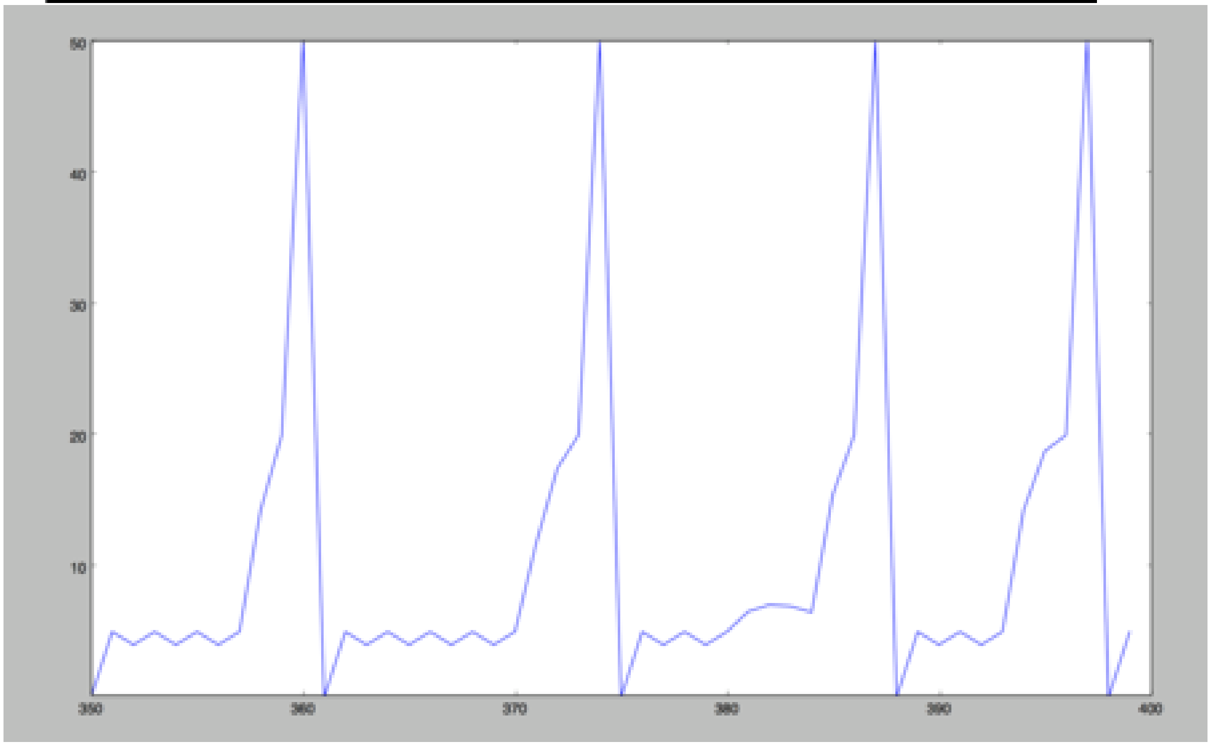
```
Trained!
        [0.04024531690385191, 0.055236688182346505, 0.18942938220995642, 0.8086459699643065, 0.026482702465628942,
         0.09899467107723792, 0.688279699360771, 0.02000523906803927, 0.06242438237537025, 0.017649197587241816]
        [0.5451935496252057, 0.6676662826483746, 1.5785231187830442, 0.5222112238646031, 1.6135098100158478,
         0.6880737924242122, 0.1050506615829927, 1.6974531244815876, 1.5574446201550067, 0.5400328958118613]
        [0.7936840742407522]
        [0.5105698465562775]
        [0.4512217542732067]
        [0.16537918100645171]
        [1.3478673657532056]
        [0.210127745424326]
        [0.13416059281985104]
        [0.32298562068732334]
        [0.04881491037384403]
        [0.5126063466520816]
Classifying...
(0, 0): 6
(0, 1): 11
(1, 0): 9
(1, 1): 6
```

Here are some of the instances where my network successfully approximated the non-linear XOR function. Each shows the output neuron's time-to-first-spike, which yielded similar values in the case that the inputs were the same and similar values in the case that the inputs were different, along with the synapse weights that enabled this to occur. The weights of the first five synapses from the hidden layer to the output layer are considered negative because they protrude from inhibitory LIF neurons.

The following plot is a graph of the voltage of my output neuron during the classification phase when it spiked near t=5ms for inputs (0,0) and (1,1) and near t=7.5ms for inputs (0,1) and (1,0). It is evident that the output neuron spiked when two excitatory post-synaptic potentials were propagated to the output neuron in the same time frame, and the distances between spikes correspond to the aforementioned time-to-first-spike values of 5ms, 7ms, 8ms, 5ms.



```
Trained!
        [0.3416105755601697, 0.005661826520617197, 1.4423143933792992, 0.1945716528645226, 0.7224888836828705,
         1.043419784732982, 0.6538190766654256, 0.3221322420878603, 0.3348576375646668, 0.10451815067551073]
        [0.38510149227558965, 0.2962791329077159, 0.7416716130789005, 0.34644031875012804, 0.06301882277375892,
         1.69402692464239, 0.9043464226932366, 0.490437256648105, 0.37221817249194855, 0.1838613563707534]
        [0.015988390594000257]
        [0.7192094114595515]
        [1.2088169389642824]
        [0.7107297126126461]
        [0.8497313198802183]
        [0.3969558791964524]
        [1.5226955301112448]
        [0.41181607254927644]
        [0.76770266345521]
        [0.11498142189882221]
Classifying...
(0, 0): 5
(0, 1): 7
(1, 0): 8
(1, 1): 5
```



2 Line Detector: Tempotron

2.1 Theory

My line detector network responds selectively to lines with a horizontal orientation. To do this, I implemented a tempotron with $N = 9$ input neurons feeding post-synaptic potentials to one output neuron. My images were $5x5$ grids of characters with each input representing a line oriented differently, incremented by 20 degrees each. To classify an image, I would break it up into $N = 9$ $3x3$ perceptive windows, and use "switch functions" to map these windows to switches, which are booleans that indicate whether an input neuron should be on or off. For a input neuron to be "on", I would pass in a constant current that would make the input neuron spike in pre-determined increments. For each timestep of my classification, I would sum up all post-synaptic potentials (PSP's) induced by each pre-synaptic spike and track the output voltage until it crossed a maximum threshold, which would indicate 'True' in a binary classification. My voltage and PSP's followed these dynamics:

$$V(t) = \sum_i w_i \sum_{t_i} K(t - t_i) + V_{rest}$$

$$K(t - t_i) = \begin{cases} V_0[exp(-(t - t_i)/\tau) - exp(-(t - t_i)/\tau_s)] & t \geq t_i \\ 0 & t < t_i \end{cases}$$

where t_i indicated the spike time of the i^{th} input neuron with weight w_i, V_{rest} is the resting voltage, $K(t - t_i)$ is the increase in voltage induced in the output neuron at time t by the i^{th} input neuron, τ and τ_s are decay time constants, V_0 normalizes each individual PSP kernel to the range $[0, 1]$.

In order to train the tempotron, I would iterate over the input images and for a given image, I would check what the tempotron currently classifies it as. If the image was classified as a valid horizontal line but the truth is it isn't, I would subtract Δw_i from each input neuron's synaptic weight, and if the image was not classified as a valid horizontal line but the truth is it is, I would add Δw_i to each input neuron's synaptic weight. The equation is as follows:

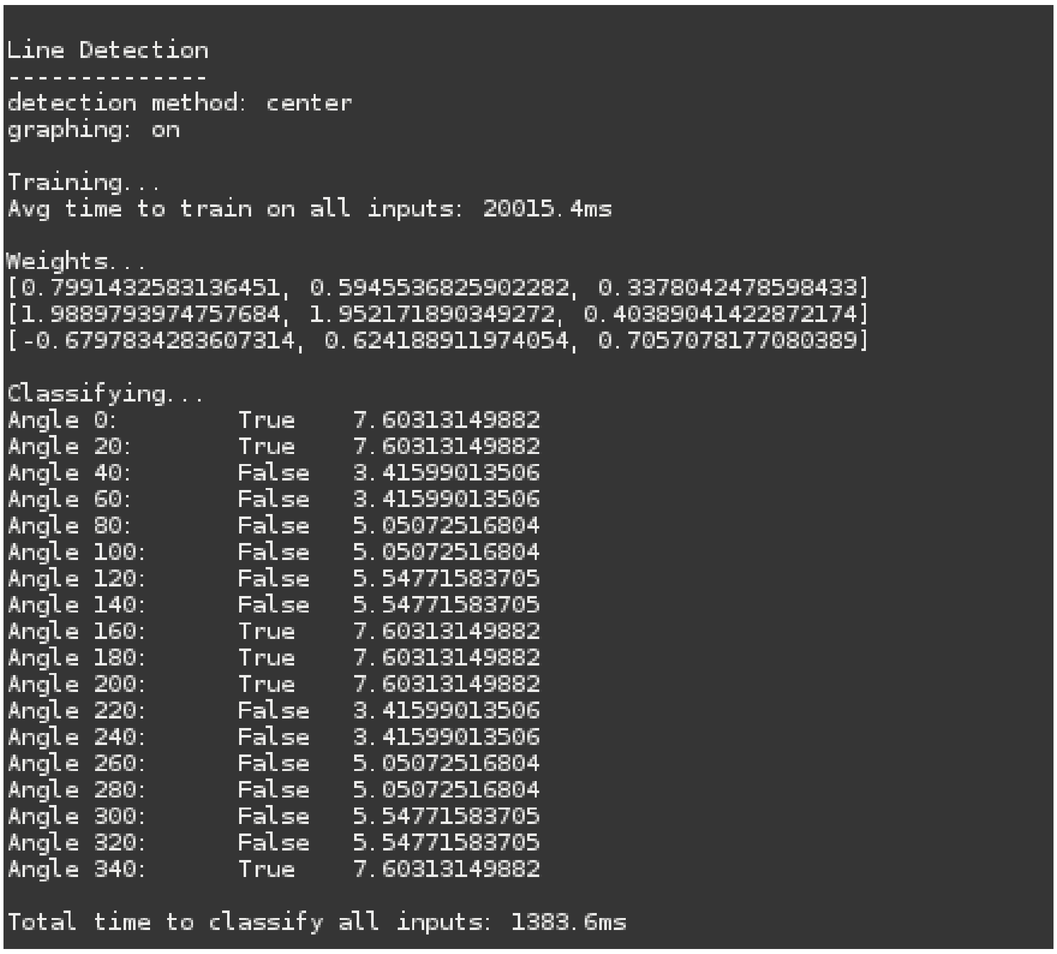
$$\Delta w_i = \lambda \sum_{t_i < t_{max}} K(t_{max} - t_i)$$

where t_{max} is the time at which the output neuron reaches its maximum voltage over the simulation (bounded by a time threshold). This update rule follows the gradient-based approach of minimizing the cost function $V(t) - V_{th}$ where V_{th} is the maximum threshold to classify an image to True or False.

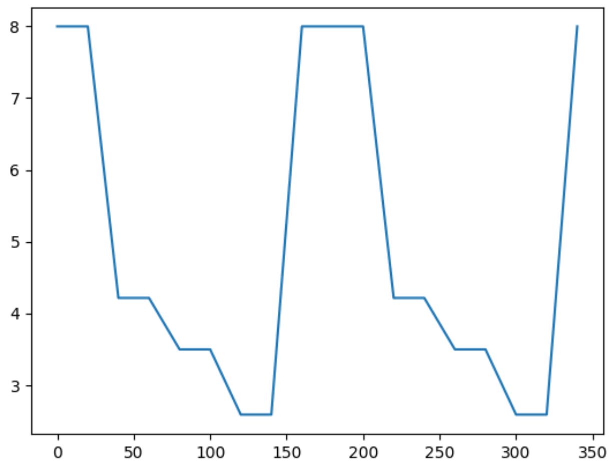
2.2 Setup

I initialized my weights to random numbers uniformly in the range $[-1, 1]$. I set λ, which indicates how much weights should change on updates, to $\frac{1}{4}$. I made "on" neurons spike for the first time at $t = 4$, I set my time threshold to 25 timesteps, my τ to 4, and τ_s to 1. I varied the training and classifications on two different switch functions: one which turned an input neuron "on" if the center neuron had an asterisk, and another which turned an input neuron "on" if the 3 center horizontal neurons had asterisks. With this setup, I was able to get consistent and correct classification behavior.

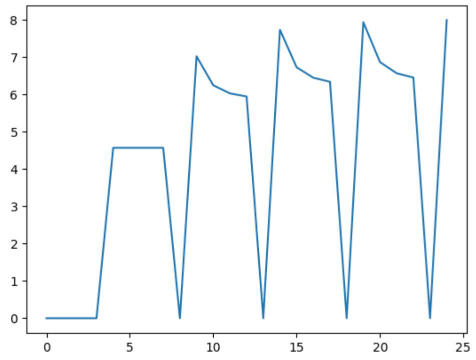
2.3 Results: Center Perception



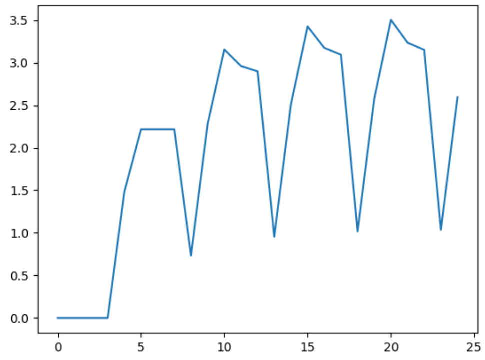
```
Line Detection
--------------
detection method: center
graphing: on

Training...
Avg time to train on all inputs: 20015.4ms

Weights...
[0.7991432583136451, 0.5945536825902282, 0.3378042478598433]
[1.9889793974757684, 1.952171890349272, 0.40389041422872174]
[-0.6797834283607314, 0.624188911974054, 0.7057078177080389]

Classifying...
Angle 0:        True    7.60313149882
Angle 20:       True    7.60313149882
Angle 40:       False   3.41599013506
Angle 60:       False   3.41599013506
Angle 80:       False   5.05072516804
Angle 100:      False   5.05072516804
Angle 120:      False   5.54771583705
Angle 140:      False   5.54771583705
Angle 160:      True    7.60313149882
Angle 180:      True    7.60313149882
Angle 200:      True    7.60313149882
Angle 220:      False   3.41599013506
Angle 240:      False   3.41599013506
Angle 260:      False   5.05072516804
Angle 280:      False   5.05072516804
Angle 300:      False   5.54771583705
Angle 320:      False   5.54771583705
Angle 340:      True    7.60313149882

Total time to classify all inputs: 1383.6ms
```



With the simplistic detection method of only checking center pixels, we see the weight matrix that yields the following classifications, which are formatted as Angle, Classification, and Maximum Output Voltage induced. The lines angled at degrees of 0, 20, 160, 180, 200, and 340 all yield the same maximum output voltage and are therefore classified as horizontal lines, which makes sense because the 3x3 windows for each of these lines check the same center values and therefore induce the same currents in the same input neurons. Here are the output voltage graphs for lines at orientations of 0 degrees and 100 degrees (the troughs are indications of spikes, for which the PSP kernel function yields 0).



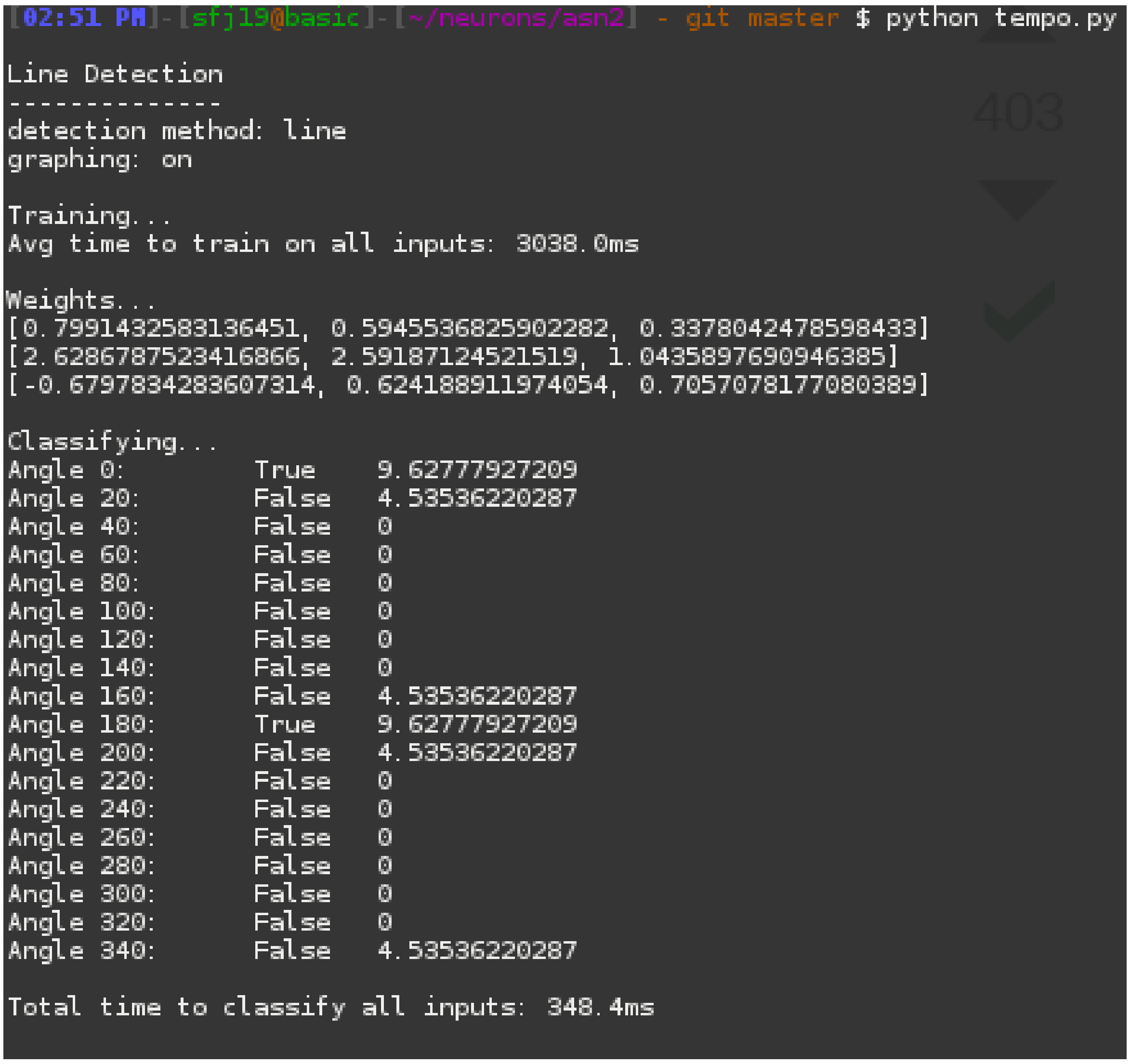
(a) Output Voltage for Line at 0 Degrees



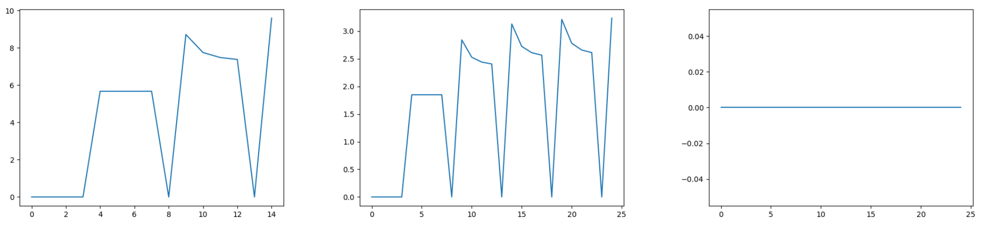
(b) Output Voltage for Line at 100 Degrees

Figure 1: Detection Method: Center Pixel

2.4 Results: Line Perception

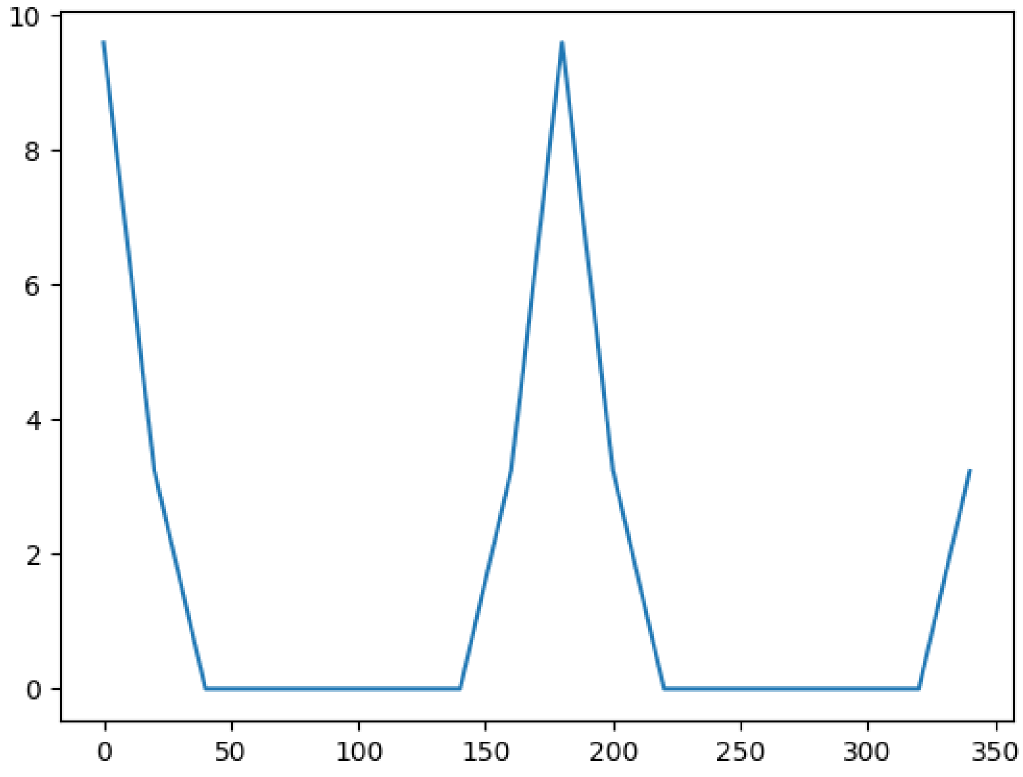


```
[02:51 PM]-[sfj19@basic]-[~/neurons/asn2] - git master $ python tempo.py

Line Detection
--------------
detection method: line
graphing: on

Training...
Avg time to train on all inputs: 3038.0ms

Weights...
[0.7991432583136451, 0.5945536825902282, 0.3378042478598433]
[2.6286787523416866, 2.59187124521519, 1.0435897690946385]
[-0.6797834283607314, 0.624188911974054, 0.7057078177080389]

Classifying...
Angle 0:         True    9.62777927209
Angle 20:        False   4.53536220287
Angle 40:        False   0
Angle 60:        False   0
Angle 80:        False   0
Angle 100:       False   0
Angle 120:       False   0
Angle 140:       False   0
Angle 160:       False   4.53536220287
Angle 180:       True    9.62777927209
Angle 200:       False   4.53536220287
Angle 220:       False   0
Angle 240:       False   0
Angle 260:       False   0
Angle 280:       False   0
Angle 300:       False   0
Angle 320:       False   0
Angle 340:       False   4.53536220287

Total time to classify all inputs: 348.4ms
```



(a) Voltage: 0 Degrees (b) Voltage: 20 Degrees (c) Voltage: 100 Degrees

Figure 2: Detection Method: 3 Horizontal Pixels



With the more apt detection method of checking horizontal lines within the perception windows, we see the weight matrix that yields the following classifications, which we note are more accurate than that of the previous method - the network only returns True for the actual horizontal lines of 0 and 180. This is reflected in that the graph has narrower curves around the true horizontal line image inputs than those of its counterpart graph for the other detection method. Also, lines without a horizontal component yield maximum voltages of 0 because no input neurons spike because no perceptive window has a horizontal line of length 3. Here are the output voltage graphs for lines at orientations of 0 degrees, 20 degrees, and 100 degrees (the troughs are indications of spikes, for which the PSP kernel function yields 0).