Assignment 2 - Basic SNN

Jihun Joo Haolun Cheng Jonathan Li
Michael Ziegler Ray Sy Joseph Shamma

December 2020

Contributions

- **Ray Sy: Worked on problem 2**
- **Michael Ziegler: Designed SNN for AND function and worked on pt 2 SNN**
- **Joseph Shamma: Worked on exercise 1.2**
- **Haolun Cheng: Worked on exercise 2.2**
- **Jihun Joo: Worked on problem 2**
- **Jonathan Li: Worked on problem 1**

Problem 1:

Exercises 1.2

1. Inputs x and y could be represented by an SNN with 2 input neurons - 1 for x and 1 for y - that encode spikes using a mathematical model done on a millisecond time scale. The output would then depend upon which of the 2 neurons would record a spike in a given millisecond. For example, say at $t = 0$ ms, neurons x and y would record no spikes, meaning in a network trained to learn AND the output neuron would not spike either. However, if both neurons fired at $t = 10$ ms, then the output neuron would spike soon after as well.
2. The same approach as above could be utilized, only instead of encoding by time, the firing rate over a given epoch for x and y could be recorded and compared. For example, if neurons x and y each exhibit a positive firing rate from 0-100 ms to represent their respective values of 1, then the given output neuron could be trained to spike given this input in order to represent the AND function.
3. **Single-Layer SNN to represent AND function**

   The single layer SNN we designed to solve this problem involves 2 input neurons, representing x and y, exhibiting different firing rates meant to represent a 0 or 1 in order to generate a firing rate from the output neuron corresponding to the correct evaluation of the AND function.

Figure 1: Graphic representation of our single layer SNN

In lieu of the cited architecture from Wall and Reljan-Delaney, our network encodes a 0 input to a neuron as a firing rate of 25 hz and a 1 as a firing rate of 51 hz. This was done by inputting a certain amount of input current into our existing LIF model so that for an input of 0, that neuron would be receiving enough current to fire at a speed of 25 hz. The same also goes for an input of 1 and its corresponding firing rate.

```
def LIF(I, Cm, Rm):
    """
    Runs a LIF simulation on neuron and returns outputted voltage

            Parameters:
                    I (double[]): A list of input voltages in mV
                    Cm (double): The membrane capacitance
                    Rm (double): The membrane resistance

            Returns:
                    V (double[]): A list of the output voltages in mV
    """
    V_thresh = 30
    V_rest = -65
    V_spike = 80
    dT = 0.02  # time step in ms
    total_time = (I.size) * dT

    # an array of time
    time = np.arange(0, total_time, dT)

    # default voltage list set to resting volatage of -65mV
    V = (-65) * np.ones(len(time))
    #V_bin = 0*np.ones(len(time))

    did_spike = False

    # function member variable to track spikes
    LIF.spikes = 0.0

    for t in range(len(time)):
        # using "I - V(t)/Rm = Cm * dV/dT"
        dV = (I[t] - (V[t - 1] - V_rest) / Rm) / Cm

        # reset membrane potential if neuron spiked last tick
        if did_spike:
            V[t] = V_rest + dV * dT
        else:
            V[t] = V[t - 1] + dV * dT

        # check if membrane voltage exceeded threshold (spike)
        if V[t] > V_thresh:
            did_spike = True
            #V_bin[t] = 80
            # set the last step to spike value
            V[t] = V_spike
            LIF.spikes += 1
        else:
            did_spike = False

    return V, LIF.spikes
```

Figure 2: Code representation of our existing LIF model

Encoding Input

```
def generateSpikeRates(input):

    """
    Encodes input into resulting spike rates for all timesteps in a time frame given input current injected
    Parameters:
        input: Array of inputs at timesteps t
    Returns:
        spike_rate: Array of spiking rates at timesteps t
    """
    t = 100
    dT = .01
    time = np.arange(0, t + dT, dT)

    spike_rate = np.empty(input.size).astype(float)
    for i in range(input.size):
        """calculate spike rate for each iteration"""

        I = np.zeros(len(time))
        I[0:len(time)] = input[i]

        # run LIF simulation
        temp, spikes = LIF(I=I, Cm=4, Rm=5)

        # calculate the spike rate during the period
        spike_rate[i] = spikes / (time[len(time)-1] - time[0] ) * 100

    return spike_rate

def encodeInputs(x,y):

    """
    Encodes neuron activity from x and y into rates generated from input current injected at each dT
    Parameters:
        x: Array of binary values representing x
        y: Array of binary values representing y
    Returns:
        x_out: Inputs of x encoded by their corresponding firing rates
        y_out: Inputs of y encoded by their corresponding firing rates
    """

    one_curr = 107
    zero_curr = 58

    inputToCurr = lambda t: zero_curr if t == 0 else (one_curr if t == 1 else -1)

    one_rt = 51
    zero_rt = 25

    x_in = np.array(np.array([inputToCurr(xi) for xi in x]).astype(float))
    y_in = np.array(np.array([inputToCurr(yi) for yi in y]).astype(float))

    x_out = generateSpikeRates(x_in)
    y_out = generateSpikeRates(y_in)

    return x out, y out
```

Figure 3: Implementation of encoding methodology

The learning rule we employed was the standard Hebbian learning model, in which our pre-synaptic variable was represented by the appropriate encoding of our input neuron's spiking behavior, and our post-synaptic variable was taken from a teacher set used to train our model to exhibit the right behavior. This teacher set consists of the appropriate firing rates for the output we are trying to achieve (so in this case, if $x = [0, 0, 1, 1]$ & $y = [0, 1, 0, 1]$, the teacher set will be equal to [25,25,25,51].)

Our training then proceeds as follows:

- Initialize our weights to 0 and pass in our inputs corresponding to our *x,y,*& teacher neurons.
- Encode *x* and *y*.

$$\frac{d}{dt} w_{ij} = a_2^{corr} v_j^{pre} v_i^{post}$$

Figure 4: Standard Hebbian learning model

- Compute our total adjustments to the weights by finding the dot products between our encoded inputs and our teacher outputs, then multiplying them by a_2^{corr}.
- Add the adjustments to their corresponding weights and repeat the process for the given number of epochs.

Training

```
In [7]: def train(x,y,targ,init,l_rt,epochs):
            print("---------------training---------------")
            print("input layer:\n", np.array([x,y]).T)

            x_out, y_out = encodeInputs(x,y)
            print("inputs encoded as rates:\n", np.array([x_out,y_out]).T)

            weights = init

            for epoch in range(epochs):
                print("iter: ", epoch)
                w_adj_x = l_rt*np.dot(x_out,targ) #standard Hebb model modified for teacher neuron => Δw=a*v_pre*v_post_targ
                w_adj_y = l_rt*np.dot(y_out,targ)
                weights[0] += w_adj_x
                weights[1] += w_adj_y
                print(weights)

            print("--------------------------------------")

            return weights
```

Figure 5: Training algorithm

Upon examining the spiking behavior of the output neuron, if its firing rate was within a certain threshold from 51 hz (in our code, this threshold was equal to ±3 hz), then the output was decoded as 1. Otherwise if its firing rate was within the same threshold from 25 hz - or was equal to 0 hz - the output was decoded to 0.

Predicting

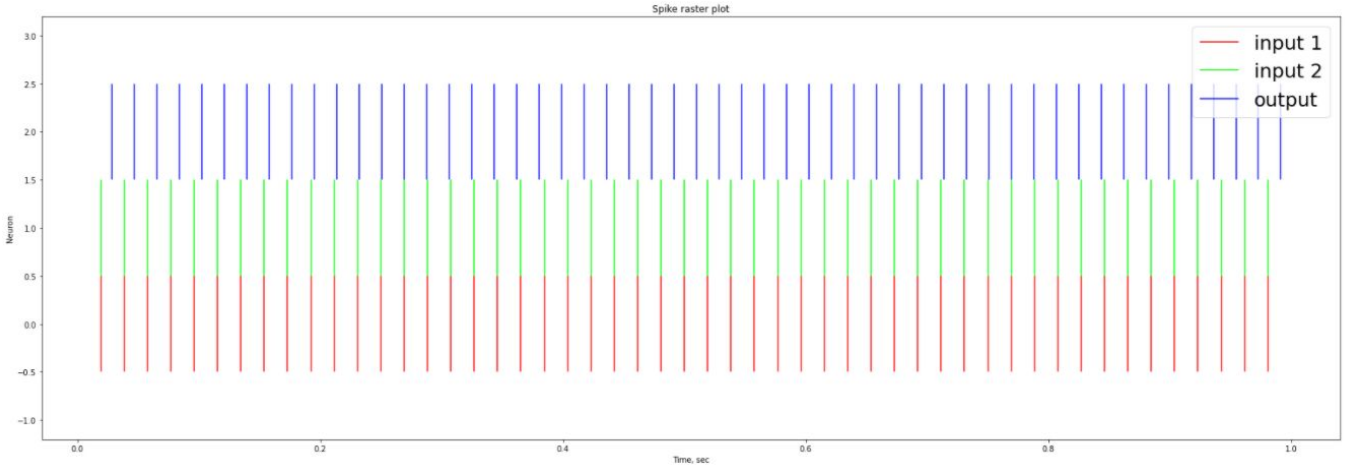
```
def predict(input, weights, thresh):

    print("input: ", input)

    #p_in, temp = encodeInputs(input, np.array([]))
    p_sum = np.dot(input, weights)
    p_rt = generateSpikeRates(p_sum)

    zero_lo = zero_rt - thresh
    zero_hi = zero_rt + thresh

    one_lo = one_rt - thresh
    one_hi = one_rt + thresh

    if (zero_lo <= p_rt <= zero_hi) or (p_rt == 0):
        prediction = 0
    elif one_lo <= p_rt <= one_hi:
        prediction = 1

    print("your predicted rt: ", prediction)
```

Figure 6: Prediction algorithm

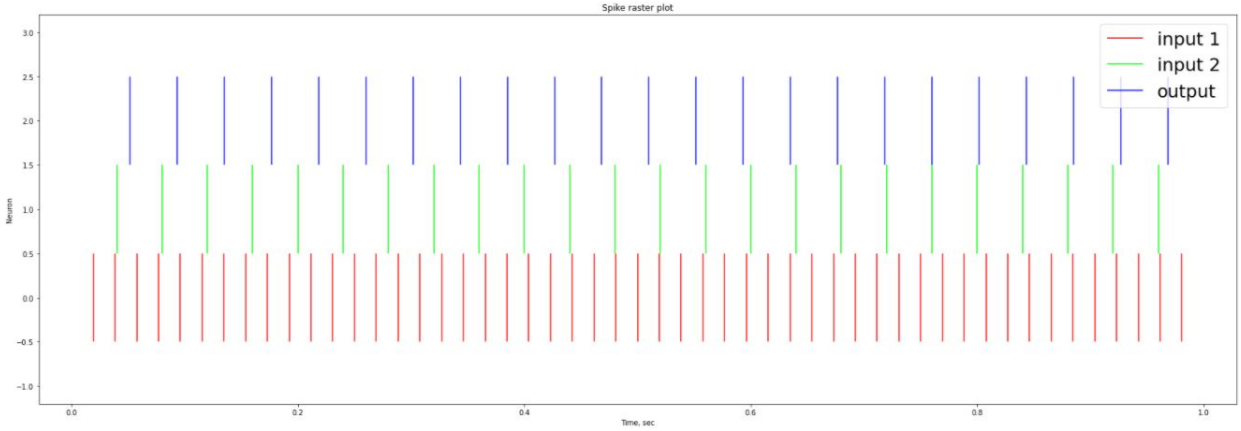
With $a_2^{corr} = 0.00055$, epochs = 20, the following output raster plots are generated for each possible set of inputs. Note that for $x = 1$, $y = 1$, the firing rates for the output is the same as that of inputs 1 and 2, implying

they are encoding the same value of 1. However, note that for inputs $x = 1,\ y = 0,$ the firing rate of the output neuron matches that of the input neuron exhibiting a firing rate representing 0. Finally, for $x = 1,\ y = 0,$ the output neuron generates no spiking behavior.

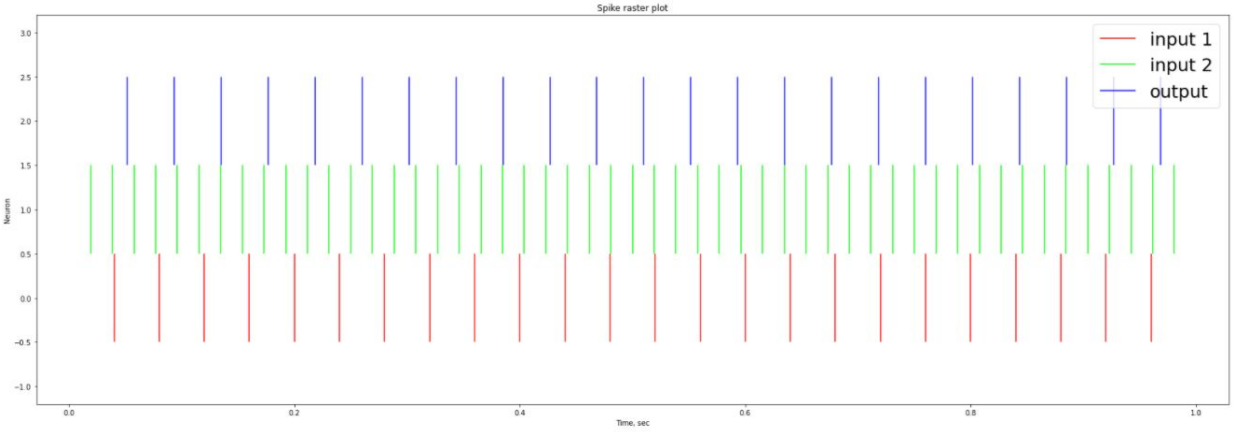
1 AND 1 = 1



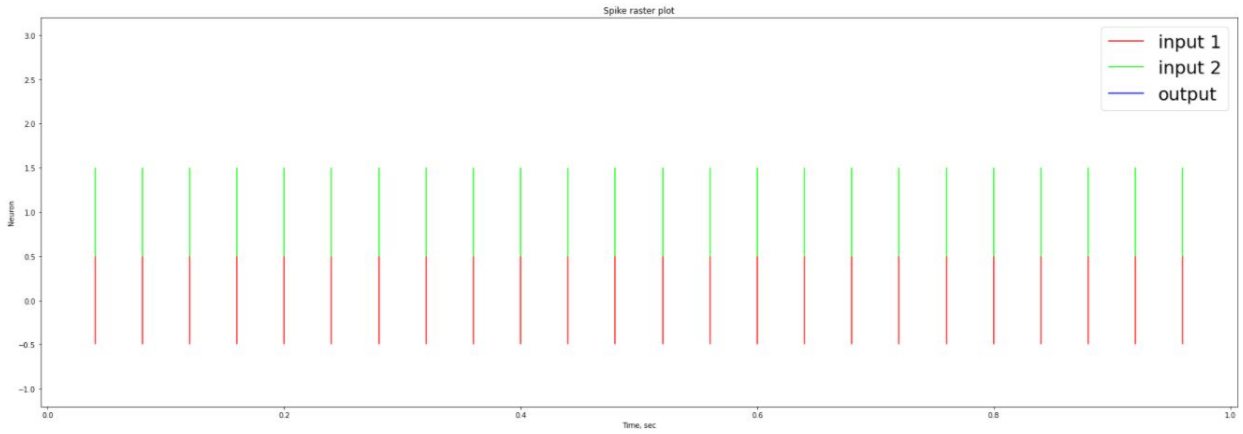
0 AND 1 = 0



1 AND 0 = 0



0 AND 0 = 0



OR Function

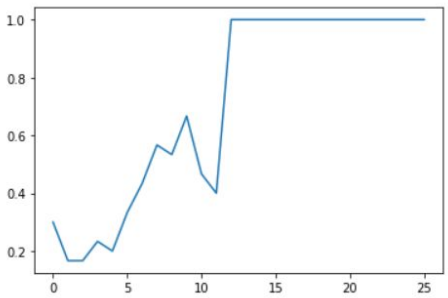
Result:

Our Single Neuron Model can indeed learn the OR gate function. Since for OR, whenever one input neuron fires, the output would also fire, compared to our AND gate weights, the OR gates would learn to have a higher weight. From our training accuracy graph we can see that after 26 epochs, our Spiking Neural Network has achieved a 100% accuracy. From our raster plots, we can see that the neural network is indeed firing correctly, where the output neuron would fire at a rate above 50 spikes/second as long as there is one input that is 1, and that when both input 1 and input 2 is firing at the rate we encoded for 0, there are 0 spikes in the output.

Training accuracy:

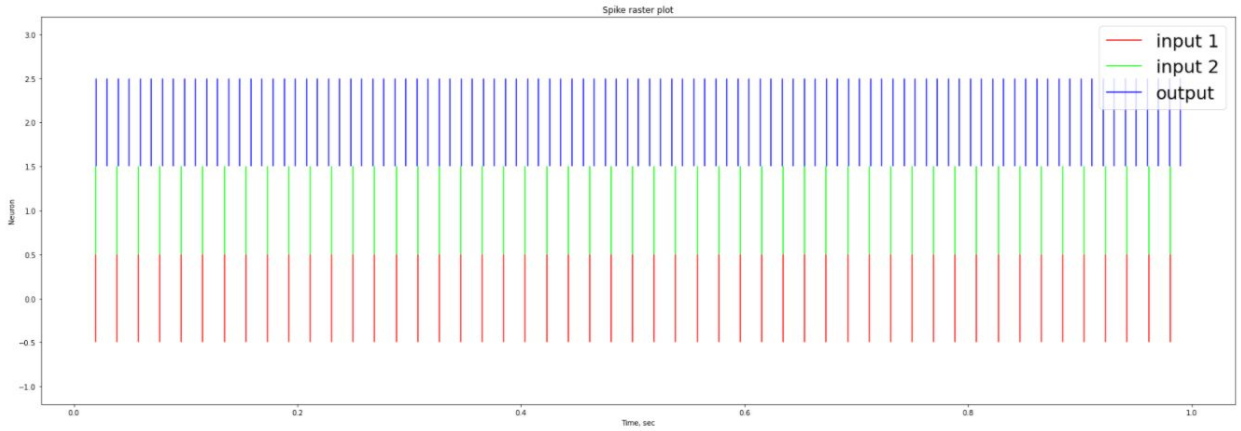
Acuracy after each epoch:

```
In [26]: a=np.stack(train.accuracy, axis=0)
         b=np.reshape(a, (26))
         plt.plot(b)
Out[26]: [<matplotlib.lines.Line2D at 0x6d5e164220>]
```

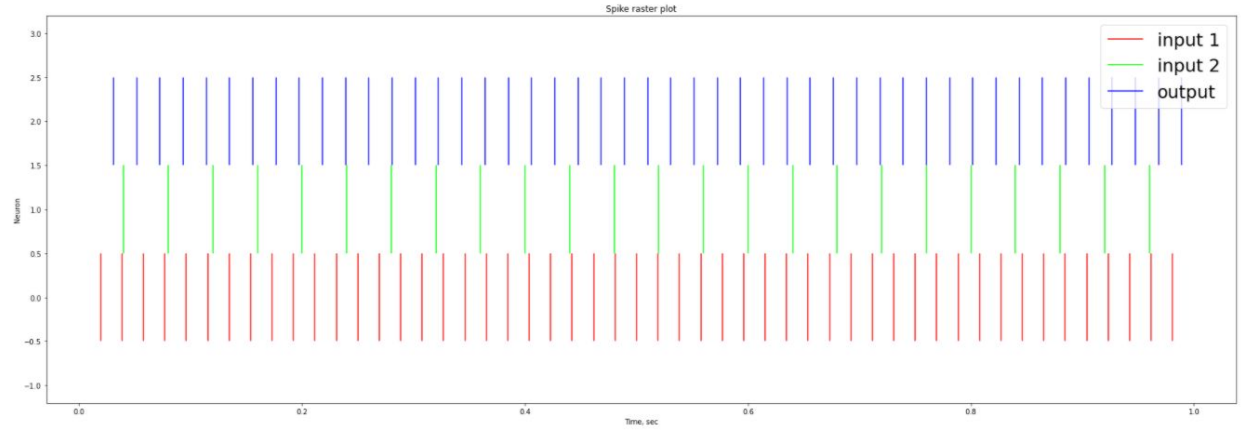


Raster plots:

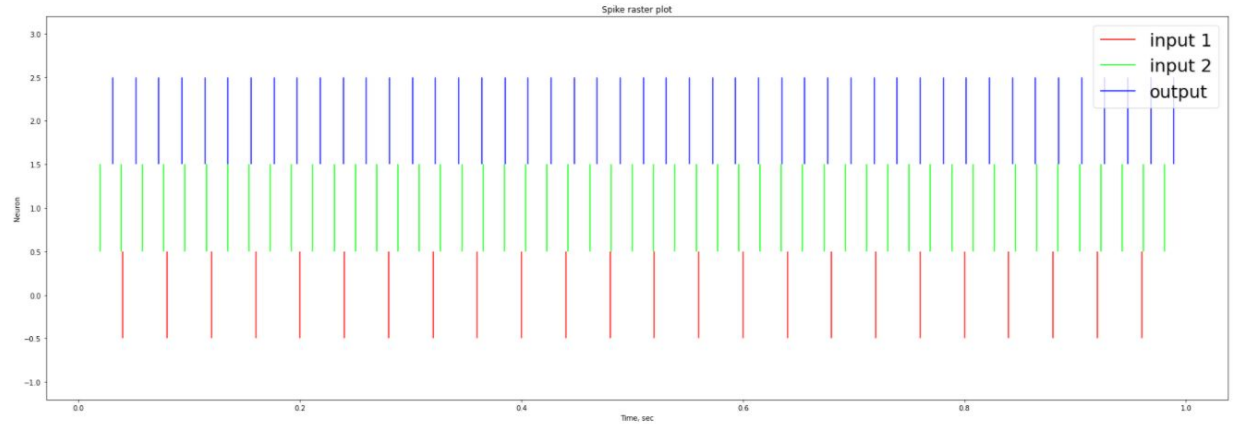
1 OR 1 = 1



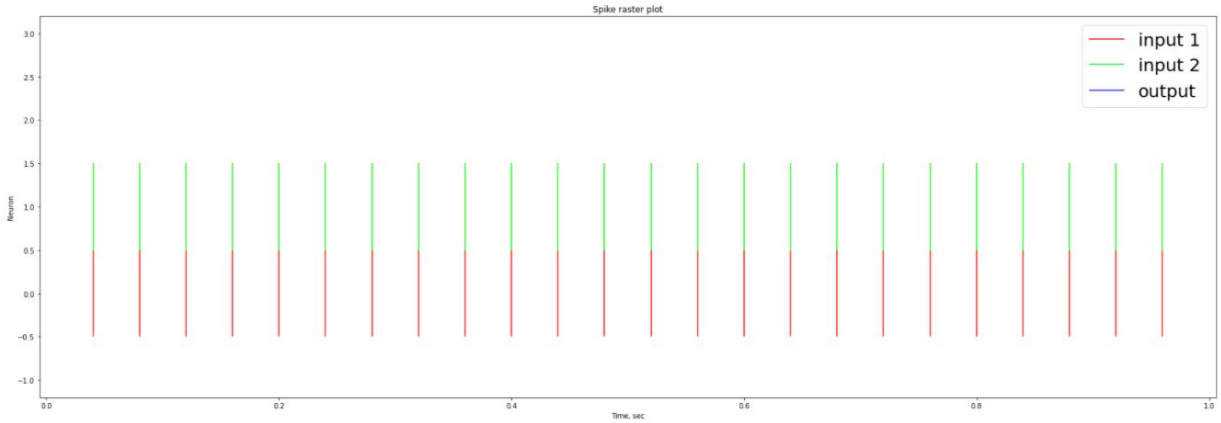
1 OR 0 = 1



0 OR 1 = 1



0 OR 0 = 0



XOR function

You cannot simulate XOR with a single layer. According to Reljan-Delaney and Wall, “In particular, it was shown that a single layer perceptron is not capable of solving linearly inseparable problems, such as the XOR problem. A linearly inseparable outcome is the set of results, which when plotted on a 2D graph cannot be delignated by a single line. A classic example of a linearly inseparable problem is the XOR function...”

Problem 2:

Exercises 2.2

1.

a. We now have 64 neurons since there are 8*8=64 total pixels, each of them will be an input.
b. In this case, “chance level” would be the probability of picking the correct output randomly out of N possible outputs, meaning the probability of doing so would be 1/N. So if we are attempting to classify out of 10 possible images, chance level would be 1/10=10%.
c. Three output neurons would be needed - one for each possible digit - in order to assign weights that will correlate each pixel out of 64 to each possible output neuron. In the case of 10 digits, this would mean you need 10 output neurons.
d. For this modified SNN, we decided to employ a Poisson Encoding scheme in order to generate a spike train representation for each of our images. Citing the provided paper on spike based MNIST encoding (Fatahi, Ahmadi, Shahsavari, Ahmadi, & Devienne, 2016)
e. Our algorithm for doing so was as follows:
    - normalize all r rate (pixel intensity) values to be between 0 and 1
    - generate M uniform random numbers between 0 and 1 where M is equal to the number of bins of time we're examining spiking activity (since dT = 1, M will be equal to the size of the array)
    - for all i in M random numbers in rand, if r[i] > rand[i], generate 1 spike

After executing this process, 3 arrays will be returned of size M: one containing the normalized pixel intensities of each bin, one containing their binary spike values, and one consisting of the maximum amount of intensity at each index where a spike occurs for plotting purposes.

```python
def poissonEncoding(rates):
    """
    Transforms a 1D array of inputs representing an image into spike train representation utilizing Poisson method of
    encoding

    Params:
        rates: 1D array of inputs (in this case, pixel intensitiies corresponding to our image)
    Returns:
        n_rates: Normalized set of input pixel values (represents probabilities of a spike occurring at each dT)
        bin_spikes: 1D array of binary values representing timesteps in which spikes occur
        p_spikes: 1D array representing pixel intensities of spike train representation for plotting/computation purposes

    """
    dT = 1

    max = np.amax(rates)
    n_rates = (rates*dT)/max

    rand = np.random.uniform(0,1, len(rates))
    bin_spikes = np.zeros(len(rates))
    pix_spikes = np.zeros(len(rates))

    for i in range(len(rates)):
        if n_rates[i] > rand[i]:
            bin_spikes[i] = 1
            pix_spikes[i] = max

    return n_rates, bin_spikes, pix_spikes
```

Figure 7: Poisson encoding algorithm

Beyond this, we also made modifications to our training process in order to accommodate the dataset. Using training set *X* and teacher set *Y,* both populated with images from the MNIST dataset where *X[i]* and *Y[i]* represent the same number, our new methodology computes adjustments to all weights by utilizing the same standard Hebbian model from Figure 5. Our new process also includes a method to test the accuracy of our data using training vs. validation data, showing how both improve over time over the course of the first epoch, and then again with each recurrent epoch. Our full algorithm can be found in Figure 8.

2. Binary Digit Classification

Digit Classification - 2 digits

```
test_bin = []
test_ind = []
for i in range(len(X_test)):
    if (digits.target[len(X_train)+i] == 1) or (digits.target[len(X_train)+i] == 3):
        #print("adding digit")
        test_bin.append(X_test[i])
        test_ind.append(digits.target[len(X_train)+i])

count = 0
for i in range (len(test_bin)):
    if test_ind[i] == _predict(test_bin[i], weights):
        count+=1
print("accuracy for classifying 1 and 3: ", count/len(test_bin))

test_bin = []
test_ind = []
for i in range(len(X_test)):
    if (digits.target[len(X_train)+i] == 8) or (digits.target[len(X_train)+i] == 3):
        #print("adding digit")
        test_bin.append(X_test[i])
        test_ind.append(digits.target[len(X_train)+i])

count = 0
for i in range (len(test_bin)):
    if test_ind[i] == _predict(test_bin[i], weights):
        count+=1
print("accuracy for classifying 8 and 3: ", count/len(test_bin))

#print(test_bin)
```

```
accuracy for classifying 1 and 3:  0.7222222222222222
accuracy for classifying 8 and 3:  0.45714285714285713
```

The code above is used to classify between 2 digits in our existing network. We used our existing training algorithm to train our network to classify all 10 images. Then to test its ability to distinguish between 2 digits, we fed in a dataset containing only images of 1 and 3, and then another dataset consisting of only images of 8 and 3. We then compared our network's accuracy when making predictions between these 2 datasets.

Note that the accuracy for predicting between images of 1 and 3 is much higher than that of the accuracy of predicting between images of 8 and 3. This is likely due to the fact that images of 8 and 3 would likely be sharing similar neurons (pixels) that are spiking, leading our network to be much less accurate when distinguishing between the two.

3. Single-Layer SNN for 10 Digit Classification

```
def train(X, Y, val, l_rt, init, epochs):

    weights = init

    for i in range(epochs):

        print("epoch: ", i)
        print("training......")
        for x in range(len(X)):

            true_index = digits.target[x]
            #print("index: ", x, " true index: ", true_index)
            in_neurons = X[x]
            out_neurons = Y[x]

            X_n_rts, X_bins, X_pixs = poissonEncoding(in_neurons)

            weight_adjs = np.zeros((len(in_neurons),10))

            y_out  = np.zeros((10,len(in_neurons)))
            y_out[true_index] = out_neurons.flatten()

            for pix in range(len(in_neurons)):
                for digit in range(10):
                   weight_adjs[pix][digit] = l_rt*X_bins[pix]*y_out[digit][pix]

            weights = weights + weight_adjs

            if (i == 0) and (x % 250 == 0):
                print("********************")
                print("validating after ",x,"th image:")
                validate(X,val,weights.T,0,len(X_train)+len(X_test))

        print("weights after epoch ", i,": ", weights, "\n")
        validate(X, val, weights.T, 0, len(X_train)+len(X_test))
        print("------------------------------")


    return weights.T
```

Figure 8: Digit classification training algorithm

Upon adjusting our network to operate with 64 input neurons and 10 output neurons, we use the above algorithm to train our data. Note that in the above algorithm, during the first epoch validation testing is performed for every 250 images trained, and then again after each subsequent epoch is completed. Running our training algorithm with $a_2^{corr} = 0.0008$ and a validation set of 180 images gives us the following sample metrics in Figure 9 for our first epoch, and the metrics in Figure 10 for our other epochs and final weights. From these metrics, we can see that our validation accuracy increases with training accuracy until peaking in the 64-70% range, whereas training accuracy peaks in the 72-75% range.

Training

```
init_weights = np.zeros((len(X_train[0]),10))

#w_i,j = w_out,pix
weights = train(X_train, y_train, X_val, 0.0008, init_weights, 3)
```

```
epoch:  0
training......
********************
validating after  0 th image:
training accuracy:  0.09951287404314545
validation accuracy:  0.08888888888888889
total accuracy:  0.09833024118738404
********************
validating after  250 th image:
training accuracy:  0.6214335421016005
validation accuracy:  0.6055555555555555
total accuracy:  0.6196660482374768
********************
validating after  500 th image:
training accuracy:  0.6339596381350034
validation accuracy:  0.5722222222222222
total accuracy:  0.62708719851577
********************
validating after  750 th image:
training accuracy:  0.6102992345163535
validation accuracy:  0.5333333333333333
total accuracy:  0.6017316017316018
********************
validating after  1000 th image:
training accuracy:  0.7007654836464857
validation accuracy:  0.5833333333333334
total accuracy:  0.6876932591218305
********************
validating after  1250 th image:
training accuracy:  0.6889352818371608
validation accuracy:  0.6055555555555555
total accuracy:  0.6796536796536796
```

Figure 9: Validation metrics for 1st epoch

```
training accuracy:  0.7362560890744607
validation accuracy:  0.7166666666666667
total accuracy:  0.7340754483611627
------------------------------
training accuracy:  0.7383437717466945
validation accuracy:  0.6388888888888888
total accuracy:  0.7272727272727273
------------------------------
training accuracy:  0.7390396659707724
validation accuracy:  0.65
total accuracy:  0.7291280148423006
------------------------------
```

Validation Testing

```
validate(X_train, X_val, weights, 0, len(X_train)+len(X_test))
```

```
training accuracy:  0.7251217814892137
validation accuracy:  0.7
total accuracy:  0.7223252937538652
```

Figure 10: Validation for other epochs & final weights

Finally, we run one last batch of testing data on a set of 180 images to evaluate our performance. The sample data indicates a testing accuracy of 66.1% for this particular run, but based upon my own personal observation testing accuracy following training typically falls between the range of 62-72%. Regardless, the accuracy achieved here on test data indicates our SNN classifies images better than simple random chance.

Final Testing

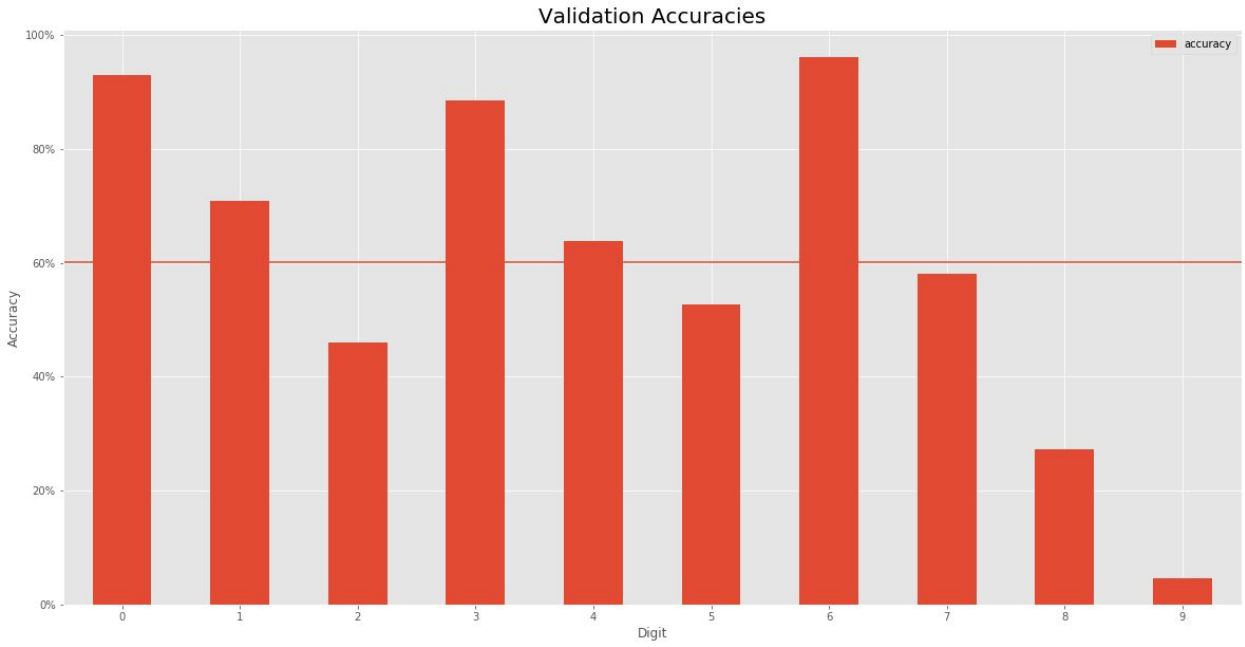
```
accuracy = sim(X_test, weights, len(X_train))
print("accuracy: ", accuracy)
```

```
image value:  2
predicted value:  2
prediction correct:  True
------------------------------
image value:  3
predicted value:  3
prediction correct:  True
------------------------------
image value:  4
predicted value:  4
prediction correct:  True
------------------------------
image value:  5
predicted value:  6
prediction correct:  False
------------------------------
image value:  6
predicted value:  6
prediction correct:  True
------------------------------
image value:  1
predicted value:  1
prediction correct:  True
------------------------------
image value:  5
predicted value:  5
prediction correct:  True
------------------------------
image value:  0
predicted value:  4
prediction correct:  False
------------------------------
image value:  9
predicted value:  9
prediction correct:  True
------------------------------
accuracy:  0.6611111111111111
```

Alternate Implementation with Rate Encoding Single Layer LIF SNN

Link to jupyter notebook with documentation in markdown:

https://github.com/fireteam99/basic-snn/blob/main/rate_encoding_single_layer_LIF_SNN.ipynb



Digit Accuracies

Digit 0: 92.96875%

Digit 1: 70.99236641221374%

Digit 2: 46.09375%

Digit 3: 88.63636363636364%

Digit 4: 63.84615384615384%

Digit 5: 52.67175572519084%

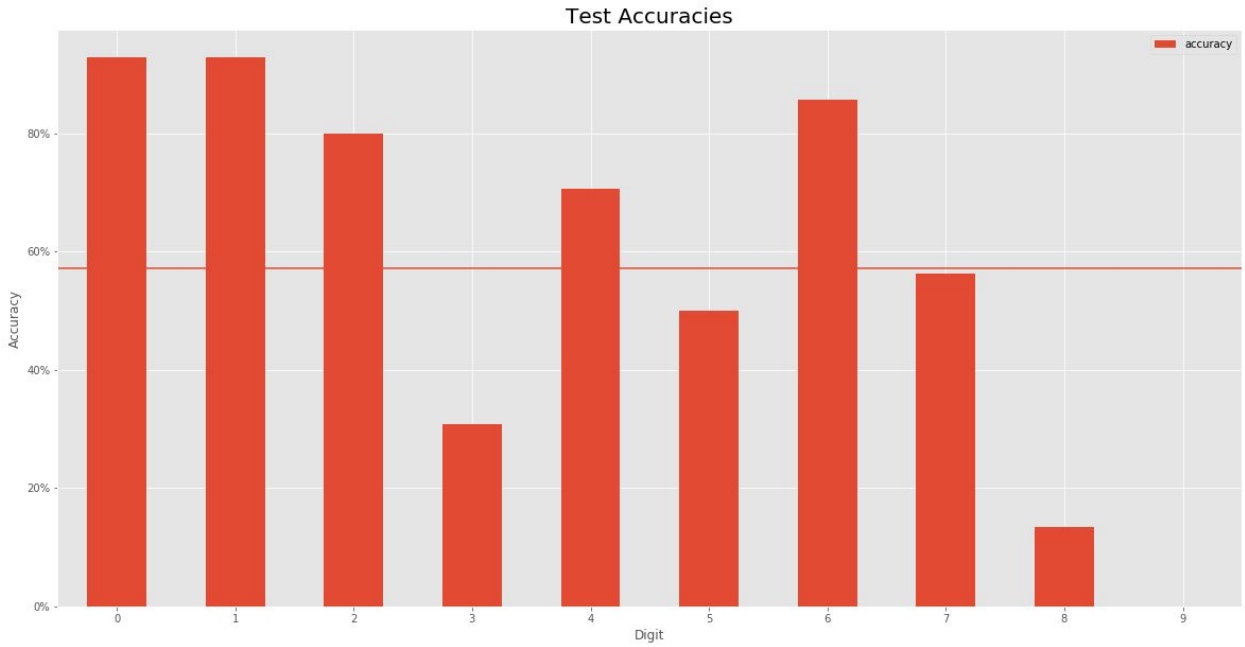
Digit 6: 96.15384615384616%

Digit 7: 58.139534883720934%

Digit 8: 27.34375%

Digit 9: 4.615384615384616%

Mean Accuracy: 60.14616552728738%



Digit Accuracies

Digit 0: 92.85714285714286%

Digit 1: 92.85714285714286%

Digit 2: 80.0%

Digit 3: 30.76923076923077%

Digit 4: 70.58823529411765%

Digit 5: 50.0%

Digit 6: 85.71428571428571%

Digit 7: 56.25%

Digit 8: 13.333333333333334%

Digit 9: 0.0%

Mean Accuracy: 57.236937082525316%

References

[1] M.Reljan-Delaney and J. Wall. Solving the Linearly Inseparable XOR Problem with Spiking Neural Networks.

[2] Fatahi, M., Ahmadi, M., Shahsavari, M., Ahmadi, A., & Devienne, P. (2016). Evt_MNIST: A spike based version of traditional MNIST. *2016 1st International Conference on New Research Achievements in Electrical and Computer Engineering*.