Neuron population simulation using Poisson Spike Trains and Perceptron Learning models

Zarif Ahmed

May 2025

Abstract

In this report we simulate two foundational models in computational neuroscience: Poisson spike trains and the perceptron learning algorithm. The Poisson spike train is a stochastic model that simulates neurons firing through generating inter spike intervals. We run 50 trials per neuron for a total of 5 seconds per spike trains. We analyzed the average firing rate, coefficient variation and the Fano Factor. Our findings revealed that the average firing rate, CV and FF ar very close to their theoretical values. In addition we find that as we increase the length of our simulation, the FF converges closes and closer to 1 while CV does not. In the second part of the report, we model perceptron learning to classify boolean functions AND, OR and XOR. The perceptron easily learned the separating hyperplane for the linearly separable functions AND, OR but failed to converge at a solution on the non-linearly separable XOR. Finally we looked at how the model handles random inputs with high dimensionality and found that the perceptron converged in most cases.

Introduction

Understanding how neuron population communicate and learn are integral to computational neuroscience. Two models that address these problems are the Poisson spike train model and the perceptron learning algorithm. Poisson Spike trains are used to simulate the irregular inter-spike intervals in

neurons through the use of stochastic modeling. We assume that the spikes occur randomly but overtime there is a fixed firing rate that they follow. To evaluate these properties we will measure average firing rate, coefficient of variability (which captures the variability in inter spike intervals in each firing train), and the Fanon Factor (which captures the trial-to-trial variability in spike count). Each of these values will help quantify the accuracy and reliability of our model. The inter-spike intervals for our Poisson model is calculated through the equation below:

$$\text{ISI} = -\frac{1}{\lambda}\ln(u)$$

- ISI: Time between consecutive spikes.
- λ: Average firing rate (spikes/second).
- u: A random number drawn from a uniform distribution in the range $[0, 1]$, i.e., $u \sim \mathcal{U}(0, 1)$.
- $\ln(u)$: The natural logarithm of u, used to transform a uniform random variable into an exponentially distributed one.

The Perceptron learning algorithm is a simplified model that simulates how neurons in a feedforward neural network increase or decrease their synaptic strength/weights between one another as they are exposed to stimulus. The perceptron simulates this behavior by classifying inputs through a linear decision boundary. The output of a perceptron neuron is computed using:

$$y = \text{sign}(\mathbf{w} \cdot \mathbf{x})$$

- y: Output of the perceptron, either +1 or -1.
- $sign()$: Sign function that returns +1 if input is positive, -1 if negative, and 0 if zero
- w: Weight vector representing synaptic strengths.
- x: Input vector of a given size.

Our learning rule for the model is:

$$\Delta w_j = \eta(y_t - y)x_j$$

- Δw_j: The change in the j^{th} weight.
- η: The learning rate, a positive constant that controls how large the weight updates are.
- y_t: The target output for the given input pattern.
- y: The actual output of the perceptron.
- x_j: The j^{th} input value.

This rule adjusts the weight w_j after each input presentation. If the perceptron's output y differs from the target output y_t, the weight is updated in proportion to the input x_j and the learning rate η.

The MATLAB code for simulating the above two models are given in Figures 1, 2, 3 and 4 below.

```
N = 50; % Spike train count
lambda = 10; % spikes/s
T = 5; % Length of each
max_spikes = 80;

spike_trains = NaN(N, max_spikes);

u = rand(100,1);
```

```
%% Get the spike trains for 50 trials for which we will create a rastor plo
for i = 1:N
    spike_count = 0;
    current_time = 0;
    while true
        u = rand(1);
        ISI = -log(u)/lambda;
        current_time = current_time + ISI;
        if spike_count >= max_spikes || current_time > T
            break
        end
    spike_count = spike_count + 1;
    spike_trains(i, spike_count) = current_time;
    end
end
```

```
%% Get Average Spikes
avg_spikes = sum(~isnan(spike_trains), "all")/(N * T);
disp(avg_spikes)
```

```
%% build the PSTH data
bin_count = T/.2;
PSTH = zeros(1, bin_count);
bin_edges = 0:.2:5;
bin_middles = bin_edges(1:end-1) + .1;

for i = 1:bin_count
    current_time = bin_edges(i);
    next_time = bin_edges(i + 1);
    % Count spikes in this bin across all trials
    total_spikes = 0;
    for trial_num = 1:N
        % Get spikes for this trial (excluding NaNs)
        spikes = spike_trains(trial_num, ~isnan(spike_trains(trial_num, :)));
        % Count spikes in [current_time, next_time)
        spikes_in_bin = (spikes >= current_time) & (spikes < next_time);
        total_spikes = total_spikes + sum(spikes_in_bin);
    end

    PSTH(i) = total_spikes / (N * .2);
end
```

```
%% Calculate the CV of the ISIs in each trial
spike_counts = zeros(1, N);
CVs = zeros(1, N);
for i = 1:N
    spikes = spike_trains(i, ~isnan(spike_trains(i, :)));
    spike_counts(i) = length(spikes);
    ISI = diff(spikes);
    mean_ISI = mean(ISI);
    std_ISI = std(ISI);
    CVs(i) = std_ISI/mean_ISI;

end
```

```
%% Calculate average CV
CV_mean = sum(CVs)/N;
disp(CV_mean)
```

```
%% Calculate Fano Factor(FF)
FF = var(spike_counts)/mean(spike_counts);
disp(FF)
```


Figure 1: Code for Poisson spike train model with calculation of FF, CVk and the raster and PSTH plots

```
N = 50; % Spike train count
lambda = 10; % spikes/s
T_vals = [10,100,200,400,800,1600,3200,6400]; % Length of each

CVs = zeros(1, length(T_vals));
FFs = zeros(1, length(T_vals));

%% Get the spike trains for 50 trials for each time value
for t_i = 1:length(T_vals)
    T = T_vals(t_i);
    max_spikes = 2*lambda*T;
    spike_trains = NaN(N, max_spikes);

    %% Calculate spike trains for current simulation time
    for i = 1:N
        spike_count = 0;
        current_time = 0;
        while true
            u = rand(1);
            ISI = -log(u)/lambda;
            current_time = current_time + ISI;
            if current_time > T
                break
            end
            spike_count = spike_count + 1;
            spike_trains(i, spike_count) = current_time;
        end
    end

    %% Calculate CV of Trial 1
    spike_count_trial_one = spike_trains(1, ~isnan(spike_trains(1, :)));
    ISI = diff(spike_count_trial_one);
    CVs(t_i) = std(ISI)/mean(ISI);

    %% Calculate number of spikes in each trial to calculate variance between trial for FF
    spike_counts = sum(~isnan(spike_trains), 2)';
    FFs(t_i) = var(spike_counts)/mean(spike_counts);
end

figure;
semilogx(T_vals, CVs, '-o','DisplayName', 'CV')
hold on;
semilogx(T_vals, FFs, '-s', 'DisplayName', 'FF')
xlabel('Time (sec)')
ylabel('Value')
title('CV and FF at different simulation length')
grid on;
```

Figure 2: Code for Poisson spike train model graphing FF and CV at different simulation lengths

```
n = 1;
N = 2;
w = zeros(1,N+1);
T = 1000; % Stop condition

patterns = [-1 -1 -1; 1 -1 -1; -1 1 -1; 1 1 -1];

AND = [-1 -1 -1 1];
OR = [-1 1 1 1];
XOR = [-1 1 1 -1];
consective_correct = 0;
convergence_time = T;

performance = zeros(1, T);
for i = 1:T
    r = randi(4);
    x = patterns(r, :);
    y_target = XOR(r); %% Change this to AND OR or XOR depending on what you are looking for
    y_actual = sign(w*x');
    if y_actual == y_target
        performance(i) = 1;
        consective_correct = consective_correct + 1;
        if consective_correct == 200
            convergence_time = i-200;
        end
    else
        consective_correct = 0;
    end
    for j = 1:N+1
        w(j) = w(j) + n * (y_target - y_actual) * x(j);
    end
end

disp(convergence_time)
figure('Position', [600 400 500 200])
plot(performance, '.k', 'MarkerSize', 1)
xlabel('Step#')
ylabel('Performance(fail: 0, pass: 1)')
title('Performance each step for XOR')
ylim([-0.2 1.2])
```

Figure 3: Code for perceptron learning with AND OR and XOR functions

```
n = 1;
N = 50;
M = 40; %edit number of patters
T = 1000; % Stop condition
runs = 1; %edit number of runs
convergences = zeros(1, runs);

figure('Position', [600 400 500 1000])

for k = 1:runs
    w = zeros(1,N+1);
    patterns = randi(2,N,M)-1;
    patterns = 2 * patterns - 1;
    yt = [ones(1, M/2), -ones(1, M/2)];
    yt = yt(randperm(M));
    consective_correct = 0;

    convergence_time = T;
    performance = zeros(1, T);

    for i = 1:T
        r = randi(M);
        x = [patterns(:, r)' 1];
        y_target = yt(r); %% Change this to AND OR or XOR depending on what you are looking for
        y_actual = sign(w*x');
        if y_actual == y_target
            performance(i) = 1;
            consective_correct = consective_correct + 1;
            if consective_correct == 200
                convergence_time = i-200;
            end
        else
            consective_correct = 0;
        end
        for j = 1:N+1
            w(j) = w(j) + n * (y_target - y_actual) * x(j);
        end
    end

    convergences(1, k) = convergence_time;
    if k <=5
        subplot(5, 1, k)
        plot(performance, '.k', 'MarkerSize', 1)
        xlabel('Step#')
        ylabel('Performance (0 or 1)')
        title('Performance each step')
        ylim([-0.2 1.2])
    end
end
converged_runs = convergences < T;
average_convergence_time = mean(convergences(converged_runs))/M;
num_not_converged = sum(~converged_runs);
fprintf('Average convergence time per pattern (successful runs): %.2f steps\n', average_convergence_time);
fprintf('Number of runs that did NOT converge: %d\n', num_not_converged);
```

Figure 4: Code for perceptron learning at higher dimensional inputs

Results and Discussion

We begin by analyzing the Poisson spike train model through an initial simulation using a firing rate of 10 spikes/s. A total of 50 spike trains were generated, each lasting 5 seconds. To visualize the results, we created both a raster plot and a peri-stimulus time histogram (PSTH), shown in Figure 5.

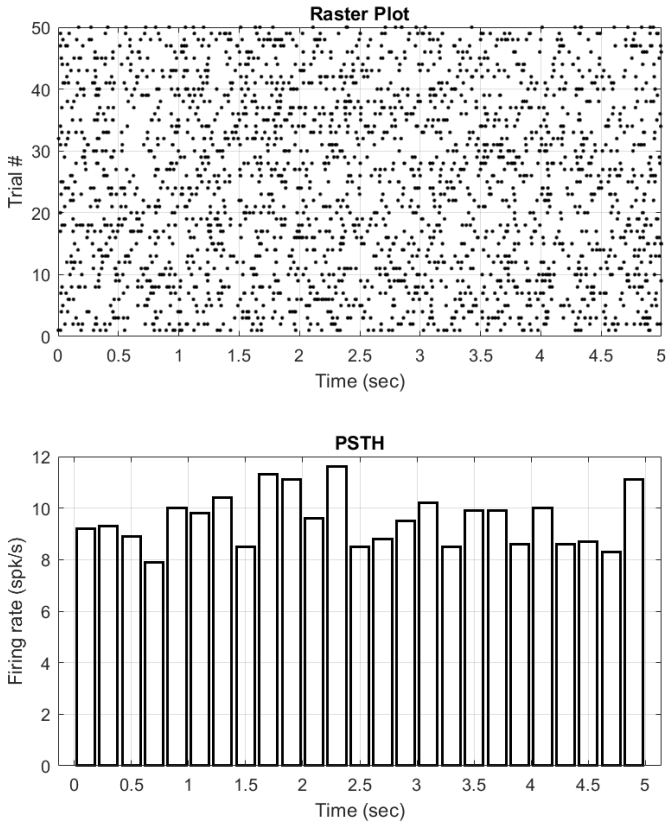

Figure 5: Raster Plot and PSTH, lambda = 10 spk/s, 50 spike trains and simulation length = 5s

In the raster plot, each horizontal line represents a single spike train (trial), and each dot corresponds to a spike time within that trial. The PSTH, plotted below the raster, shows the average firing rate over time, computed across all trials within discrete time bins. In the raster plot we find no clear pattern in spikes appearing and instead the spikes are randomly

distributed. However even though the spikes seem to appear at random times, the actual firing rate stays consistent across trials. This is consistent with how we expect neuron spike generation to behave in real world setting. To quantify this, we calculated the average spikes per second across the entire simulation by dividing the total number of spikes across all trials by the product of the number of trials and the trial duration. We found that the firing rate in this simulation was 9.5280 spikes/second which matches closely with our theoretical firing rate of 10 spikes per second.

While the PSTH reveals some variability in firing rates across different time intervals, the rates remain centered around the theoretical value. This result highlights the effectiveness of the Poisson spike train model in capturing the variable nature of neural firing, as some degree of variability is expected in real biological systems.

To further analyze the simulation we created a plot of the coefficient of variability for each trial, shown in Figure 6.

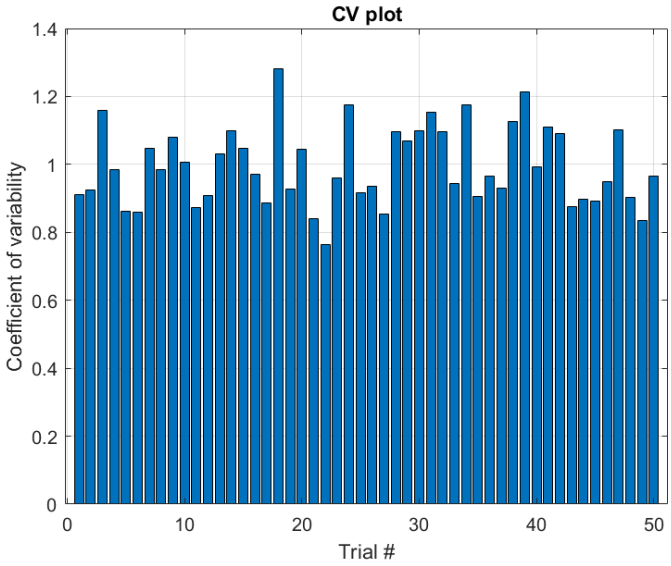

Figure 6: CV_k plot

The plot shows that the CV varies between trial to trial. However, the value consistently varies around 1 with the average CV across all trials at .9945. This shows that while ISI is random between and across trials, the total variability in ISI is consistent with properties of a Poisson distribution.

We also quantify trial-trial variability of spike count by computing the Fano-Factor. This is defined as:

$$\mathrm{FF} = \frac{\mathrm{Var}_{\mathrm{trials}}(\mathrm{count})}{\mathrm{Mean}_{\mathrm{trials}}(\mathrm{count})}$$

The Fano Factor (FF) calculated from our initial simulation was 0.8446. Theoretically, for Poisson spike trains, the FF is expected to be 1, meaning equal variance and mean in spike counts across trials. While our observed FF is reasonably close to the theoretical value, without further testing we won't know if this is true for all simulations. To further investigate, we performed 8 simulations with increasing simulation durations (10, 100, 200, 400, 800, 1600, 3200, 6400) to examine whether the FF consistently converges toward 1. Alongside this, we also plotted the coefficient of variation (CV) of the first spike train in each trial to assess whether the CV also converges towards 1 as more patterns are presented. The resulting plot is shown in Figure 7.

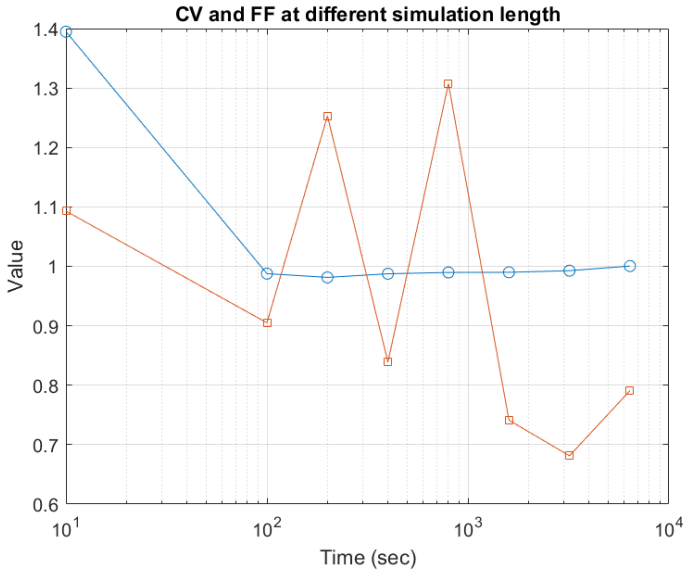

Figure 7: CV over time graph

As seen in the figure, We find that as simulation length increases, FF converges towards 1, Whereas CV does not converge to one. However CV is always some value close to 1.

Next we simulate feedforward networks using perceptron learning model. We build a perceptron with N input units x_j where j = 1...N and one output

units. We simulate our model using y = sign(w*x). For a given input pattern, if the output does not match the target output then we use our learning rule, $\Delta w_j = \eta(y_t - y)x_j$, to change the weights. We first test our model against patterns of of boolean functions AND,OR and XOR using two inputs (N=2). The performance graphs for each function is shown in Figure 8, where performance is recorded over 500 pattern presentations. In each step, one of the four possible input patterns for the Boolean function is randomly selected. For each presentation, the performance is plotted as +1 if the perceptron's output matches the target output, and 0 otherwise. This approach provides a clear visual measure of how quickly the perceptron learns to classify each Boolean function. When we have 200 consecutive successful predictions from form sign(w*x) then we say that we have successfully converged to a solution.

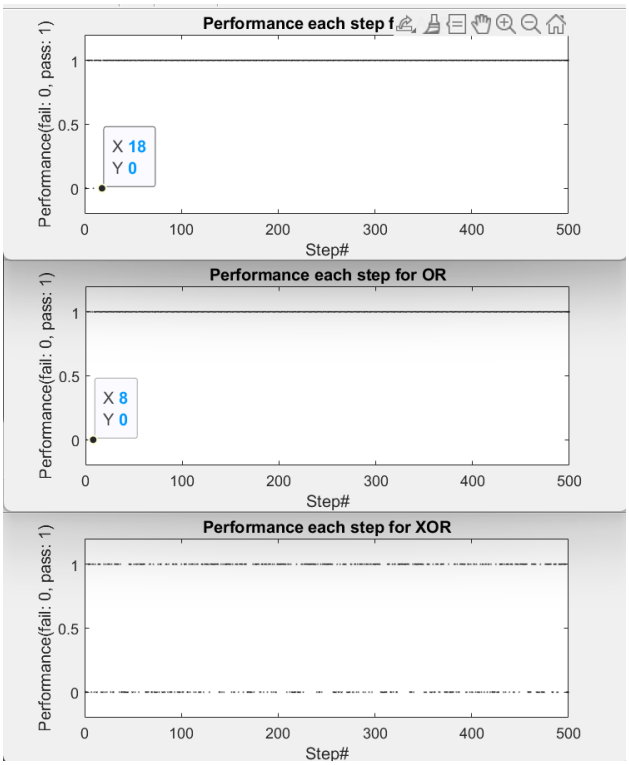

Figure 8: Performance graphs of AND, OR and XOR

From figure 8 we see that our perceptron learning model successfully converged to solutions for both the AND and OR functions but failed to

converge to a solution for the XOR function. This is because the xor function only has a non-linear solution while our perceptron model is a linear model. To further demonstrate this, we train the model with 1000 XOR patterns presentations but we still do not converge to a solution. The results are shown in Figure 9.

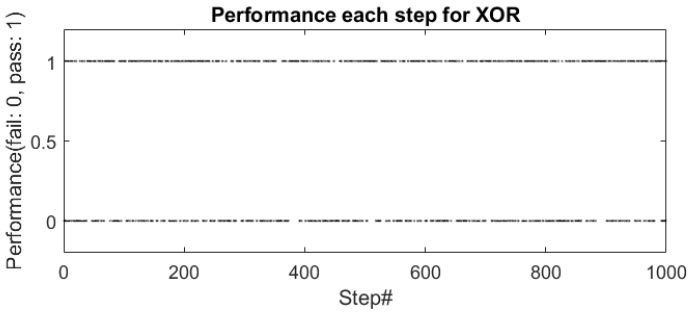

Figure 9: XOR performance graph with 1000 presentations

On the other hand the perceptron converged to a solution for the AND function within 18 steps while it converged to a solution for the OR model in 8 steps.

To further evaluate the performance of the perceptron model, we test it with inputs of higher dimensionality. We generated M=40 random input vectors each consisting of N=50 elements taking values of + or - 1. We randomly assign half the input vectors to have a target y of +1 and the other of -1. We tested whether perceptron could correctly converge to a solution to these randomly generated higher dimensional patterns. The results of 1 run is shown in Figure 10.

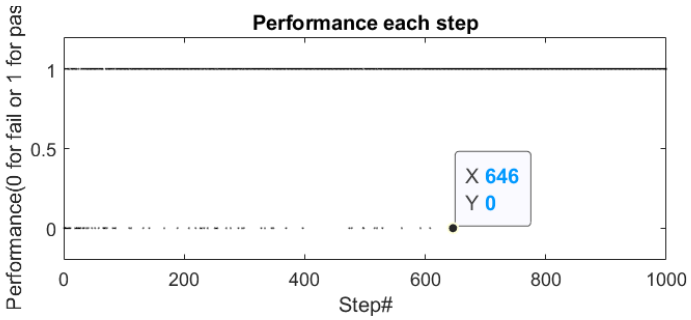

Figure 10: Performance graph for dataset of N=50 variables at n=1

After doing multiple runs, each time the elements being randomly chosen with randomly assigned target y, we find that in most cases the perceptron learning model converges to a solution in anywhere between 150 steps to 700 steps. In every other case the perceptron does not converge to a solution even if we train using 5000 presentations. We also tested the affect of changing the learning rate to .1 and 10 on the rate of convergence on our model. From running 20 tests with either n = .1 or 10 we found no pattern in how fast or slow our model converged to a solution when we change the learning rate. To quantify the average number of pattern presentation until convergence for M=40, we run 10 simulations with 1000 presentations each run. Figure 11 gives the performance graphs of 5 out of the 10 runs.

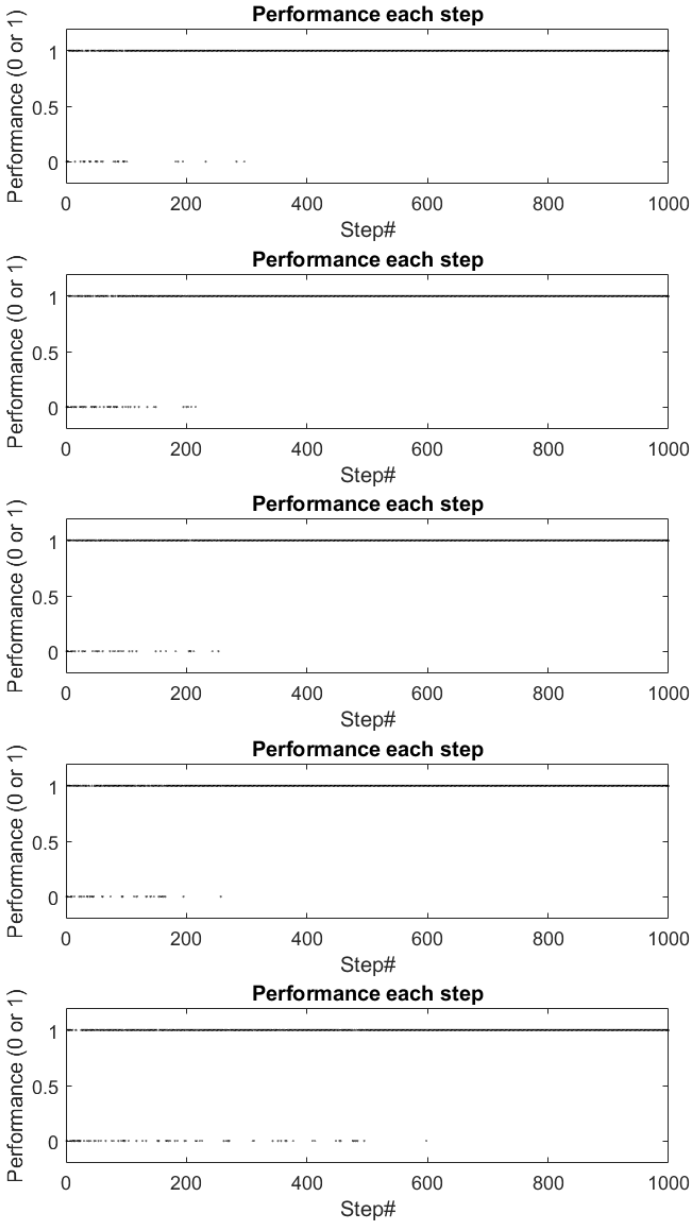

Figure 11: plots of the 5 of the 10 generated plots

The avg pattern presentation before we converged to a solution was 385 presentation. In addition, 1 out of the 10 runs failed to converge to a solution.

Finally we wanted to find the average number of presentations for each pattern i.e $< n_{conv} >_{runs} /M$ at higher number of input patterns. We repeat 10 runs with M=40 and M*100 presentations and another 10 runs with M=90 and M*1000 presentations. At M=40, $< n_{conv} >_{runs} /M$ was 9.29 steps and at M=90, $< n_{conv} >_{runs} /M$ was 307.69 steps. This suggest that at higher input amounts the perceptron learning model becomes less and less efficient.

Conclusion

In conclusion for the Poisson spike train model we found that CV and FF closely matched our theoretical expectations and the variable firing behavior produced by the stochastic model matched how neurons produce In the perceptron experiments, the model successfully converged to a solution for any linearly separable functions(AND, OR) but failed at all other cases (XOR). It also when we increase the number of input patterns in our simulation, we found that the model converged slower and slower. It also took longer to converge when we had higher dimentionality per input pattern.