I selected to keep the provided method of random initialization for the weight vector W. Most other initializations are similar to the provided code however sigma_history has neurons from 4 on initialized to random values between 0 and 1. The first value is set to the bias, the second the inputs. The third value is the first target. During backpropagation, I multiply the weight vector W and the sigma history at time t-1 to find the u_history vector at time t. I use a logsig squashing function to update the sigma_history vector at time t for neurons 3 and on (first two do not matter) based on the u_history vector at time t. Further, I compute F_o_vec with the following code:

```
if t<T_time_steps

    F_u_vec=F_u(:,t+1);

      for i = 3:Nneurons

        for j = 3:Nneurons

            F_o_vec(i)=F_o_vec(i)+F_u_vec(j)*W(j,i);

        end

    end

end

F_o_vec (1)=0; %don't change bias = neuron 1
F_o_vec (2)=0; %don't change input = neuron 2
%for output node, neuron 3, add in influences of partial deriv of errors w/rt
sigmas at time t
for i = 3:Nneurons
    %add in influences of partial deriv of errors w/rt sigmas at time t
    F_o_vec (i) = F_o_vec (i) + 2*errs(t);
end
```

Here the F_o_vec at neuron i depends on itself, F_u_vec at the jth neuron, and the weight from neuron j to i. Then, we set the first two neuron sigmas to 0 due to bias and input. Neurons 3 and on have their current values perturbed by 2 times the error vector at time t. The effect of this perturbation is that the network can better learn of its own mistakes and try to correct them based on the factual error measurements rather than heuristic ones. The gprimes are computed using the derivative logsig function of u_history and sigma_history but the first two are set to 0. In order to compute F_u, I multiply gprimes at neuron i and time t by the F_o at neuron i and time t. Finally, computing F_wjia is as easy as inializing the vector to zeroes, then using the following code:

```
for j=1:Nneurons
    for i=1:Nneurons
        for t=2:(T_time_steps)
            F_wji(j,i)=F_wji(j,i) + F_u(j,t) * sigma_history(i,t-1);
        end
    end
end
```

Notice how F_wji(j, i) depends on the sigma_history at neuron i and time t-1. Later, the weight vector is updated using ETA, F_wji, and W. All but the first four of sigma_history vector are also modified using sigma history, ETASIG, and F_os.

I made the back propagation loop until the beats converged to their proper sides within a certain threshold of tolerance for RMS error. I found that a suitable value for error tolerance was 0.005, just 0.5 percent error out of 100 percent. At this point the beats are converged almost exactly to their desired patterns. I added the tracking of iteration count in the BPTT loop as well. During testing I had the loop plot updates between results and targets every 300 iterations, but due to inefficiencies, I commented this code out for my final timed results. I found that when comparing the output generated by the test_dEsqd_dwji() function, my errors were fairly close to those errors generated by the test function, but they were not precise. The comparison varied from 2 to 5 decimal places of accuracy in comparison. Some values of parameters caused this derivative computation to vary more than that from the individual weight perturbations, however, the beats still converged to the desired patterns so I did not worry too much about the difference.

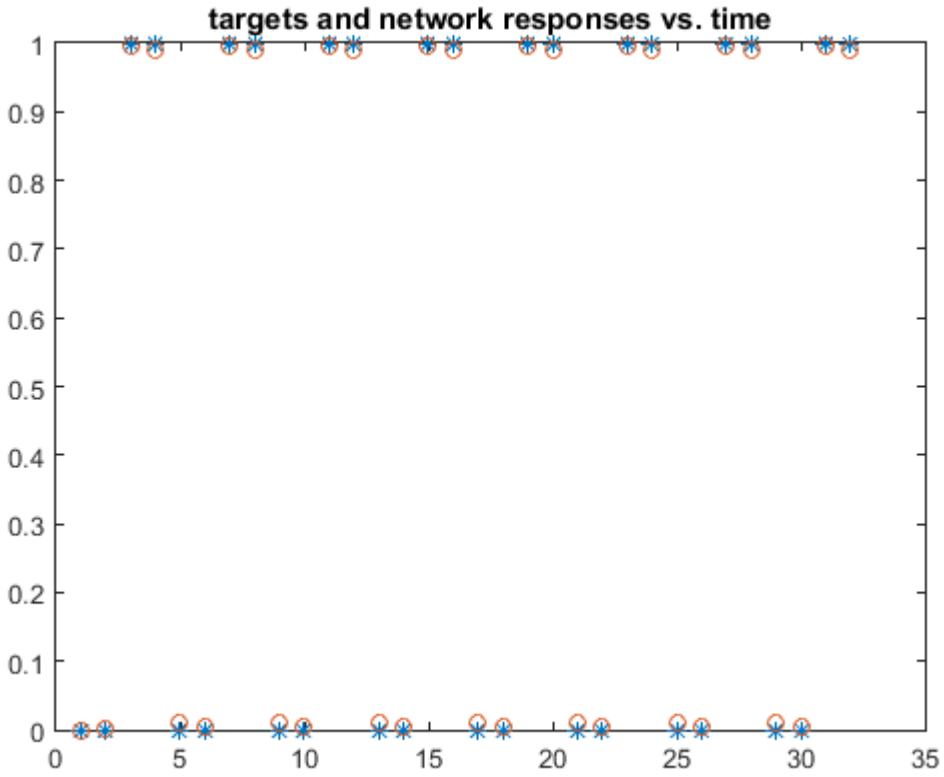
I have both beats patterns working in my implementation. For the first training set “beats.dat,” I used a total of 4 neurons to successfully train my network, i.e. a bias, input, output, and single hidden neuron. I attempted to train it with 3 neurons, but my results were poor in terms of RMS. On the second, more complex pattern “beats2.dat,” I successfully trained using 8 total neurons in the network, i.e. 5 hidden neurons. I have managed to train using less, i.e. 3 hidden neurons, but results were inconsistent. After some tweaking between testing with both datasets, I was able to settle on reasonable parameter values that allowed the network to be quick, accurate, and robust. I had to assure that the network did not overwrite its own memory too often via high learning rates (ETA and ETASIG). I experimented and found that combinations of values from the range [0.1, 0.3] were best for not wiping out valuable information. I found that holding the synapse weight magnitude at 1 was the best option because increasing it only required me to up-scale all other parameters to retain any benefit. The best parameter values I found were around ETA=0.2, ETASIG=0.25, and INIT_WEIGHT_MAG=1 for this network and problem set.

Experiments:

Beats.dat

I used an ETA of 0.2 and ETASIG of 0.25 with 4 total neurons for the first target set.

The following is a plot of my results compared to the target patterns:

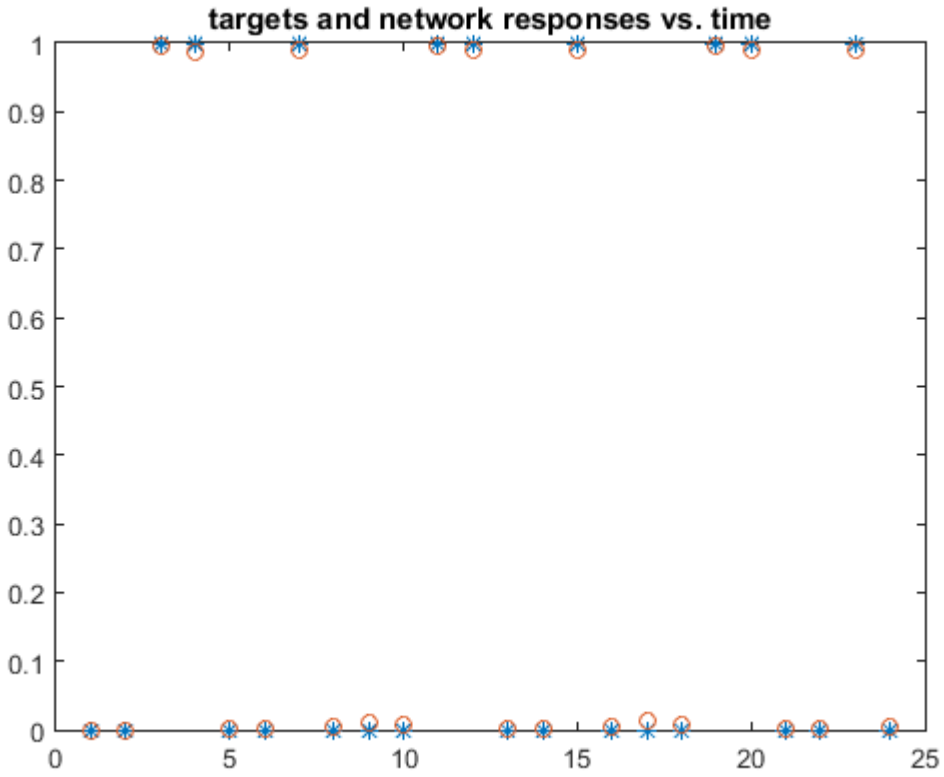


The number of iterations my program took to converge to an RMS error of 0.5% was 19,829. Obviously this process was not very quick like in Hopfield networks, but it was reasonably fast for all the computations happening in the backpropagation process. Sometimes BPTT can take a long time with non-optimal parameters and it can easily cause valuable data to be wiped from memory. This causes the network to fluctuate in error and sometimes a local minima is found. If used with too high a learning rate, BPTT can get stuck in local optima. For example, sometimes ETA=0.2 and ETASIG=0.3 work on the first target patterns but other times the loop will never converge due to a discovered local minimum in the backpropagation process. Despite this artifact, one can better tweak their learning rates to achieve a close to optimal solution in terms of convergence time and accuracy. Our convergence rate was not very long time-wise despite consisting of approximately 20k iterations. The network converged in about 1 minute to within 0.005 RMS error, which is appreciable.

Beats2.dat

I used an ETA of 0.2 and ETASIG of 0.2 to train with the second set of target patterns.

The following is a plot of my results compared to the target patterns:



The number of iterations my program took to converge to an RMS error of 0.5% was **28,657**.

Again, this number is not that high for all the calculations taking place. Sometimes the errors could get stuck in local optima if the learning rate was too high or too few hidden layer neurons were used. Hence why I lowered the ETASIG learning rate from 0.25 to 0.2 as well as increased the number of hidden interneurons to provide better capacity for holding memories and recognizing the target patterns of the beats in the second training set. Time-wise, training on this set took approximately the same amount of time if not a little more to train than the first set, i.e. ~1 minute.

Results:

Overall, we see that BPTT can be useful for recognizing patterns in datasets. BPTT can be subdued by local optima where it can never "climb" out of or "hop down" from to continue on a suitable path. Optimal network parameters must be set specifically based on dataset and complexity of patterns to recognize. It can be overwhelming testing the parameters due to the erroneous convergence, but once the proper values are found (typically smaller learning rates and larger numbers of hidden interneurons), the network can train and recognize patterns fairly efficiently and accurately. We can see that the first target set took approximately 2/3 the time of the second set with 4 more hidden interneurons in the network. The more complex dataset took a slower learning rate and more interneurons for the network to comprehend. Altogether, this shows that a more complex dataset requires more neuronal capacity in the BPTT network as well as takes both time and intricacy to train and converge on reasonable results.