Delayed and Higher-Order Transfer Entropy

Michael Hansen (April 23, 2011)

Background

Transfer entropy (TE) is an information-theoretic measure of directed information flow introduced by Thomas Schreiber in 2000 [6]. It has received much attention in neuroscience lately for its potential to identify connections between nodes in a biological network, such as individual neurons, groups of neurons, or entire regions of the brain [1]. This is important, as accurate conclusions drawn from a network analysis must be grounded in a meaningful functional-connectivity matrix.

In comparison to other commonly used information-theoretic measures, TE has many desirable properties:

- It is *model-free*, since no explicit model of how events influence each other is assumed[1]
- It can detect *non-linear* interactions between nodes
  - In contrast to Granger Causality, which can only detect linear interactions
- It is *asymmetric*, so the direction of information flow is included
  - In contrast to Mutual Information, which is a symmetric measure

In its discrete form, TE is computed over a time series which has been binned and whose state space is often binary. This form is ideally suited for neuron spike trains, where information is presumed to be transmitted by neurons modulating the timing of all-or-nothing spikes. The TE algorithm is run for every pair of neurons $i, j \in N$ and produces a square matrix T of size $|N|$ where T_{ij} is the estimated information flow from $j \rightarrow i$ (Figure 1).

[1] Aside from the assumption that events in the future cannot influence events in the past!

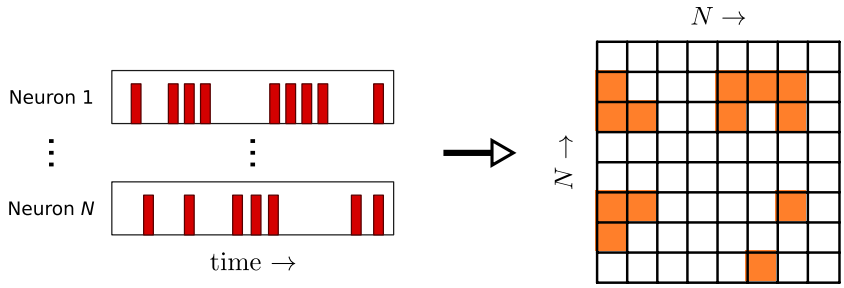


Figure 1: *High-level view of TE algorithm. N time series are used to produce an $N \times N$ matrix T where T_{ij} is the TE from $j \to i$.*

Mathematically, we can derive the formula for TE by calculating entropies for a node i, both with another node j and by itself. In the first case, we can write

$$H_{ij} = -\sum_t p(i_{t+1}, i_t, j_t) \log p(i_{t+1}|i_t, j_t)$$

where i_t is 1 if a spike occurred at time t and 0 otherwise (same for j_t), t ranges over all recorded time bins, and p is the probability of a given spike pattern. As with other entropic measures, these probabilities are estimated from the data itself, which can lead problems due to a finite sample size (more on this later). If we exclude node j from the entropy calculation for i, we get

$$H_i = -\sum_t p(i_{t+1}, i_t, j_t) \log p(i_{t+1}|i_t)$$

If j is transferring information to i, we should see a *drop* in H_{ij} relative to H_i, since knowing the history of j would reduce uncertainty about the future of i. Thus, TE is the difference between these two entropies[2]:

$$T_{ij} = H_i - H_{ij} = \sum_t p(i_{t+1}, i_t, j_t) \log \frac{p(i_{t+1}|i_t, j_t)}{p(i_{t+1}|i_t)} \tag{1}$$

Calculating TE using Equation 1 results in a measure called *delay 1 transfer entropy*

[2]We subtract H_{ij} from H_i in order to keep TE positive. Luckily, we can always ignore node j, so knowing its history will never increase our uncertainty about i.

(D1TE). This measure assumes that j is transferring information to i with a delay of one time bin (Figure 2 left). This isn't always ideal, however, because neurons (or other kinds of nodes) may communicate at a variety of delays. Therefore, it is useful to parameterize D1TE with a delay parameter d, resulting in a measure called *delayed transfer entropy* (Figure 2 right).

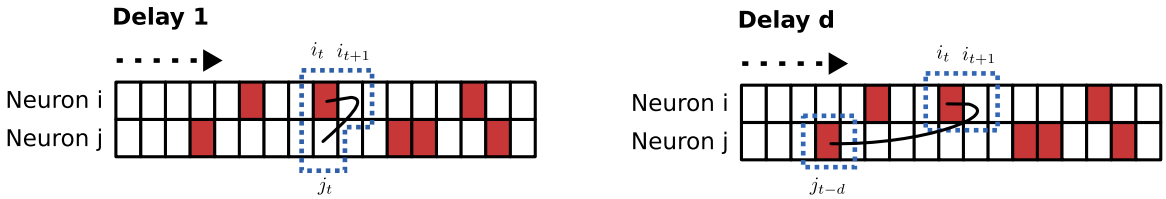


Figure 2: *Delay 1 TE for the time series of neurons i and j (left). Delayed TE for the time series of neurons i and j at delay d (right).*

Because the delay time between each node in the network is not known beforehand, delayed TE is often computed for several time delays. This produces a TE matrix for each delay time ($\overset{d}{T}$), requiring a reduction step to produce the final TE matrix T (Figure 3). Two reductions are common:

- For T_{ij}, take the maximum value of $\overset{d}{T}_{ij}$ for all d
  - This is called *peak transfer entropy* (TEPk)
- Compute the coincidence index for some window w such that
  - $T_{ij} = \sum_{d \in w} \overset{d}{T}_{ij} / \sum_d \overset{d}{T}_{ij}$ where w is centered on the peak value
  - This is called (surprise, surprise) *coincidence entropy transfer entropy* (TECI)

The final kind of transfer entropy we'll investigate in this TE tour-de-force is called *higher-order transfer entropy* (TEHo). With TEHo, we consider more than two time bins for the receiving time series i and/or more than a single time bin for the sending time series j (Figure 4). This may also be computed over multiple time delays like delayed TE, resulting in

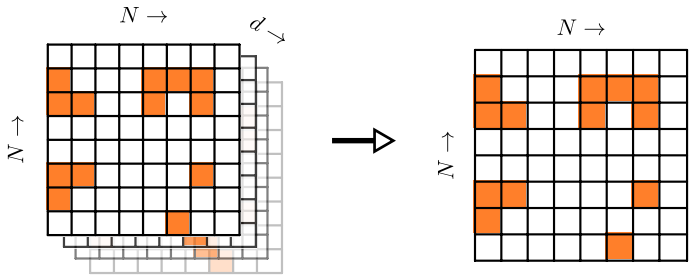


Figure 3: *Multiple delayed TE matrices must be reduced down to a single TE matrix.*

analogous peak and coincidence index measures (TEHoPk, TEHoCI). In contrast to higher-order TE, the previously discussed TE measures (TEPk, TECI) are called *first-order*.

With TEHo, we must be especially careful of the finite sample problem. As the number of bins we consider increases, it becomes more unlikely that we will observe enough samples of each pattern to correctly compute entropies.

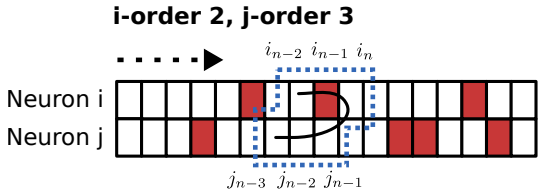


Figure 4: *Higher order TE with a receiving order of 2 and a sending order of 3. It can also computed over a number of delays like TEPk and TECI*

Algorithm Analysis

As is often the case, the mathematical formula for TE does not make it immediately obvious how computation time will scale with its various parameters. TE may have great information-theoretic properties, but the algorithms that actually compute it must be fast enough to allow for the analysis of real world data. For example, despite TE giving better results compared to other measures in [1], the authors concluded that Joint Entropy may provide a better trade-off in terms of computation time (their TE implementation took 16 hours to run on 5 minutes of data from 60 neurons for a **single time bin**).

Here at IU, Shinya Ito, John Beggs, and I have developed an algorithm for computing TE that is several orders of magnitude faster than previous implementations (tens of seconds for calculations with hundreds of neurons). At a high level, our TE algorithm looks like Listing 1, though the actual code is mostly written in C with a MATLAB interface[3].

The first-order and higher-order algorithms are essentially identical, with the only difference being that the sizes of certain variables are known at compile time for first-order (i.e. only 2^3 patterns are possible); this allows for a slight speed-up when compiler optimizations are turned on. We calculate pattern frequencies by (metaphorically) stacking shifted copies of the i and j time series on top of one another (line 7) and counting the individual column vectors of the resulting $R \times D$ matrix as the patterns (line 15).

To quantify the computational time complexity of the algorithm, we must identify which variables will dramatically affect the algorithm's run-time. For our TE algorithm, the relevant variables for a time complexity analysis are as follows:

- `duration`[4]
  - The last time bin for which we have data. This assumes a fairly consistent firing rate for all neurons.
- `order`
  - The sum of the receiving neuron order (i) and the sending neuron order (j) (plus one for the predicted time bin)
- `neurons`
  - The total number of neurons in the calculation

[3]A copy of the code is currently available at `http://code.google.com/p/transfer-entropy-toolbox/`
[4]This changes for sparse time series. See below.

Listing 1: *High-level transfer entropy algorithm*

```
duration = ... % From data
order = i_order + j_order + 1

for i = 1:neurons
  for j = 1:neurons
    % Pair of time series plus shifted copies
    ij_series = [all_series(i); all_series(j); ...]

    % Count patterns for all time bins
    counts = zeros(1, 2^order)
    while t <= duration
      pattern = zeros(1, order) % Pattern at time t

      % Check each series for a spike at time t
      for s = ij_series
        if current(s) == t
          pattern(index(s)) = 1 % Spike occurred at t
          current(s) = next(s)
        end
      end

      counts(pattern) = counts(pattern) + 1 % Count patterns
    end

    % Calculate TE from j -> i
    for c = counts
      ... % Estimate transition probabilities
    end
  end % for j
end % for i
```

For brevity, let's say $N =$ `neurons`, $D =$ `duration`, and $R =$ `order`.

Starting on line 4, we begin the two outer loops for all pairs of neurons. We must run N^2 iterations of the inner loop (starting on line 7), so we should expect our calculation time to grow quadratically in the number of neurons. Figure 5 shows actual timing results where N is varied while D and R are held constant[5].

The data for these timing calculations were generated randomly using an unbiased Bernoulli process with noise ($p \approx 0.5$) for all triplets $\{n, d, r\}$ such that:

$$\{n, d, r\} \in \begin{array}{ll} \{75, 100, 125, 150, 200, 250\} \times & \textit{neurons} \\ \{7500, 10000, 25000, 50000, 75000, 100000\} \times & \textit{durations} \\ \{3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15\} & \textit{orders} \end{array}$$

where n is the number of neurons, d is the duration, and r is the total order ($1 + i^{\text{th}}$ order $+ j^{\text{th}}$ order). For first-order calculations, $r = 3$.

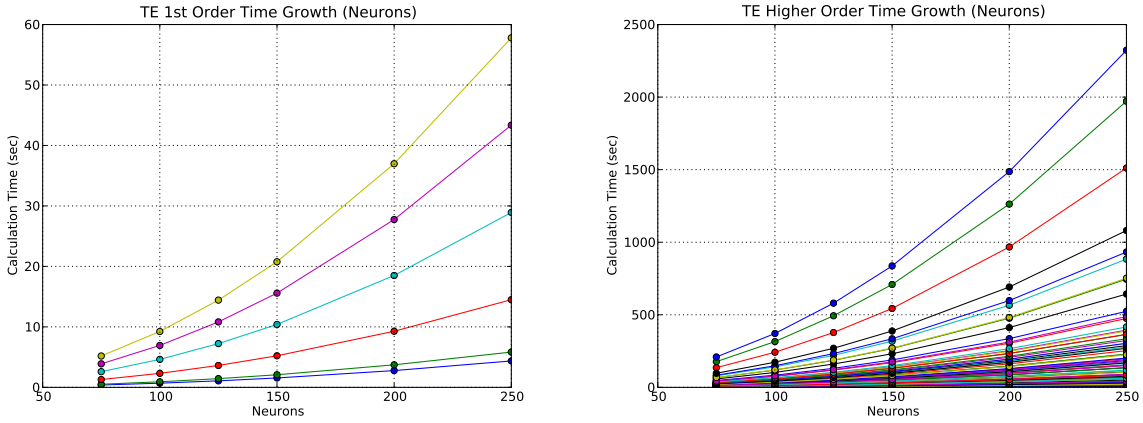


Figure 5: *TE calculation time as a function of the number of neurons (N). Each line represents an $\{n, d, r\}$ triplet. Both 1st order and higher-order times grow quadratically.*

The spike counting loop starting on line 11 does not actually need to visit all times $t \in \{1 \cdots D\}$. For sparse time series, a considerable speed-up can be had by only visiting

[5]All timing results were obtained on a 3.4GHz 8-core Intel Xeon X5492 with 32GB of RAM (Octave, 64-bit Ubuntu Linux, no parallelism).

the time bins in which spikes occurred. In this case, D is a poor estimator for the run-time of this loop. Because real world time series are usually quite sparse, I will be using D_{avg} instead, which is D times the average firing rate of all neurons ($\sum_n (S_n/D)$ where S_n is the number of spikes for neuron n).

Using D_{avg}, we should expect the calculation time to grow linearly when N and R are held constant (once again, assuming a consistent firing rate). In Figure 6, we can see this is the case for first-order calculations. Higher-order calculations appear to grow sub-linearly in D_{avg} due to the increasing overlap of time bins between the randomly generated series. This is an important point, as correlations between time series may significantly reduce the calculation time at higher orders.

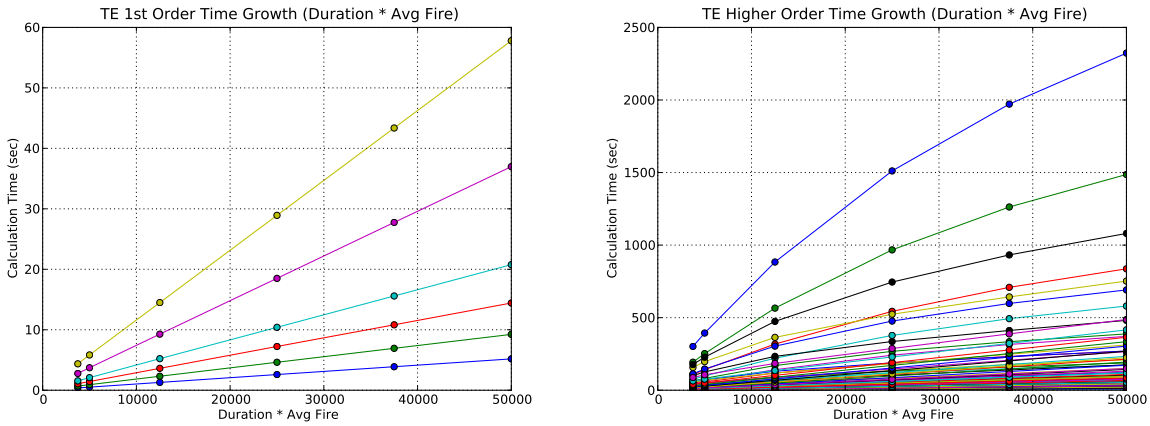


Figure 6: *TE calculation time as a function of the time series length (D_{avg}). Each line represents an $\{n, d, r\}$ triplet.*

Our last variable, R represents the order of the calculation and is used (implicitly) on line 15 and in the reduction loop on line 26. The vector `ij_series` holds all of the time series examined within the inner loop. This includes the original `i` and `j` series as well as their shifted copies[6], making `ij_series` size $1 \times R$. With N and D_{avg} held constant, we should expect our calculation time to grow exponentially in R with a linear component that

[6]The C code does not actually copy any of the series for shifting, saving an approximately $O(D_{avg}R)$ operation for each inner loop iteration. Hooray for pointers!

depends on D_{avg}. As shown in Figure 7, this becomes apparent once orders beyond 12 are measured. Note that while the time growth is exponential in R, the longest calculation time shown only took about 39 minutes ($n = 250$, $d = 10,000$, $r = 16$). Therefore, the orders used in practice ($R < 20$) will be tractable, especially if parallelism is used (see next section).

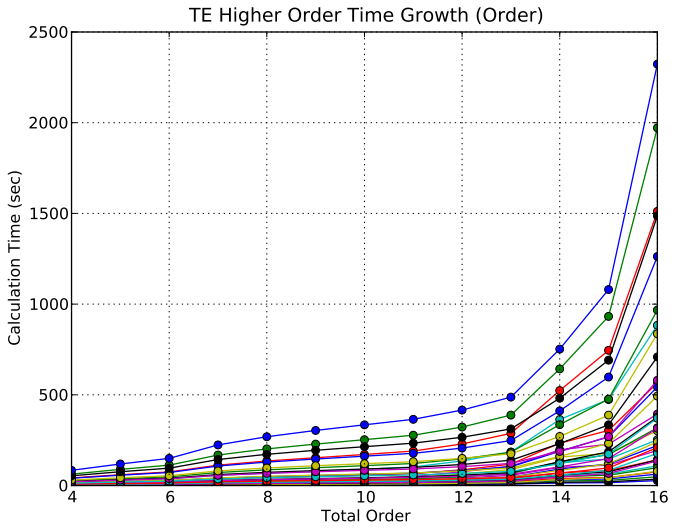


Figure 7: *TE calculation time as a function of the total order. Each line represents an* $\{n, d, r\}$ *triplet.*

Parallelism

All timing results given thus far have not exploited the inherently parallel nature of our algorithm. The inner loop starting on line 7 of Listing 1 can be executed independently for every pair of neurons, making this algorithm "embarrassingly parallel"[7].

A simple, coarse way to make use of this inherent parallelism is to compute different blocks of the final TE matrix on different threads, processors, or computers (Figure 8 left). Given the typical number of neurons in our calculations (hundreds), it's feasible to split the work among N processors and reduce the outer-loop complexity from quadratic to linear.

[7]There are additional opportunities for parallelism in the inner loop, such as dividing up portions of the time series, but I do not explore them in this paper.

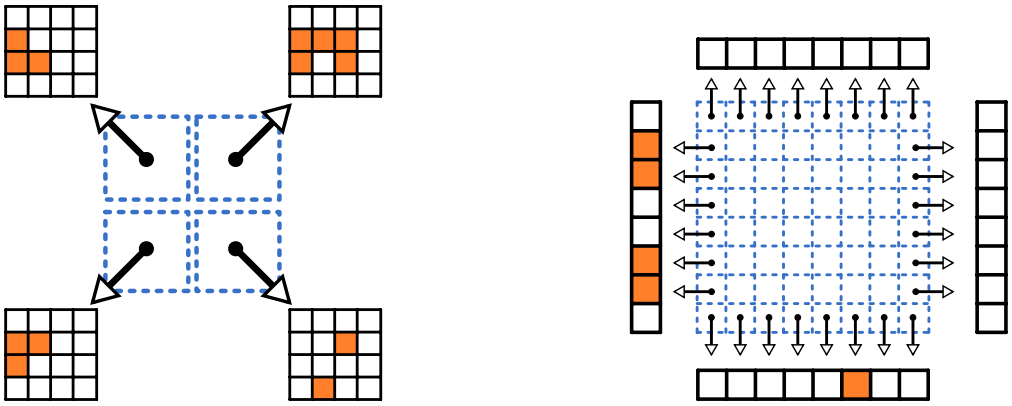


Figure 8: *Calculate blocks of the TE matrix in parallel (left) or calculate individual cells on separate GPU processors (right).*

Embarrassingly parallel algorithms often go well with a particular piece of hardware: the GPU (Graphics Processing Unit). Most modern computers have GPUs that are capable of processing dozens or hundreds of work items in parallel. This would mean that many cells of the TE matrix could be computed simultaneously (Figure 8).

With scientific computing platforms like NVIDIA's CUDA[5] and OpenCL[2], it is now possible to offload calculations onto an off-the-shelf GPU and gain considerable performance improvements. I am currently porting the algorithm to the CUDA platform to see how it compares in speed to the MATLAB version.

Model Evaluation

From the analysis above, we should be able to predict the (non-parallel) run time T of a transfer entropy calculation using the following equation:

$$T(N, D_{avg}, R) = c_1 N^2 [c_2 (D_{avg} R + 2^R)] \tag{2}$$

where c_1 and c_2 are constants that depend on the particular machine and desired time units (I use seconds for my calculations).

As mentioned above, D may only be a rough estimate of the number of visited time bins for real world data. The actual value for each iteration of the inner loop (starting at line 7 in Listing 1) will depend on the time series in `ij_series`. Specifically, this will be the size of the unioned set of all time bins in `ij_series` ($|\bigcup S_x|, x \in$ `ij_series`). Since this quantity is almost as expensive to compute as TE itself, I will be using D_{avg} as an estimate.

A simple way to evaluate this model is to do the following:

1. Collect timing data on a particular machine for randomly generated time series using a variety of N, D, and R values

2. Search for suitable values for c_1 and c_2 using the data in step 1

3. Plug the obtained values into Equation 2 and attempt to predict the calculation times of real-world time series

For step 1, I used the same random time series generated in the analysis to measure calculation times. Step 2 was accomplished by running a simple Bayesian model that attempted to fit proper c_1 and c_2 values in *Equation* 2 for the random data[8]. In my model, each calculation time was assumed to be drawn from a normal distribution whose mean was calculated using *Equation* 2 and whose precision was fixed at 0.01 (to accommodate noise). c_1 had a uniform prior in $[0, 1]$ and c_2 had a uniform prior in $[0, 1000]$.

For my test machine, I obtained the values $c_1 = 0.04678$ and $c_2 = 71.93$ for the first-order data. For the higher order data, I obtained $c_1 = 3.817 \times 10^{-11}$ and $c_2 = 580.7$. Using these values, I compared the predicted calculation times of 14 real-world time series to the actual calculation times[9]. As shown in Figure 9, my model is quite good at predicting calculation times, even when trained exclusively on random data.

[8]We used the OpenBUGS software [4] with a burn-in of 2,000 steps, a chain length of 50,000 steps, and a thinning value of 2.

[9]These "real-world" time series were from rat cortical cells, measured with an electrode array.

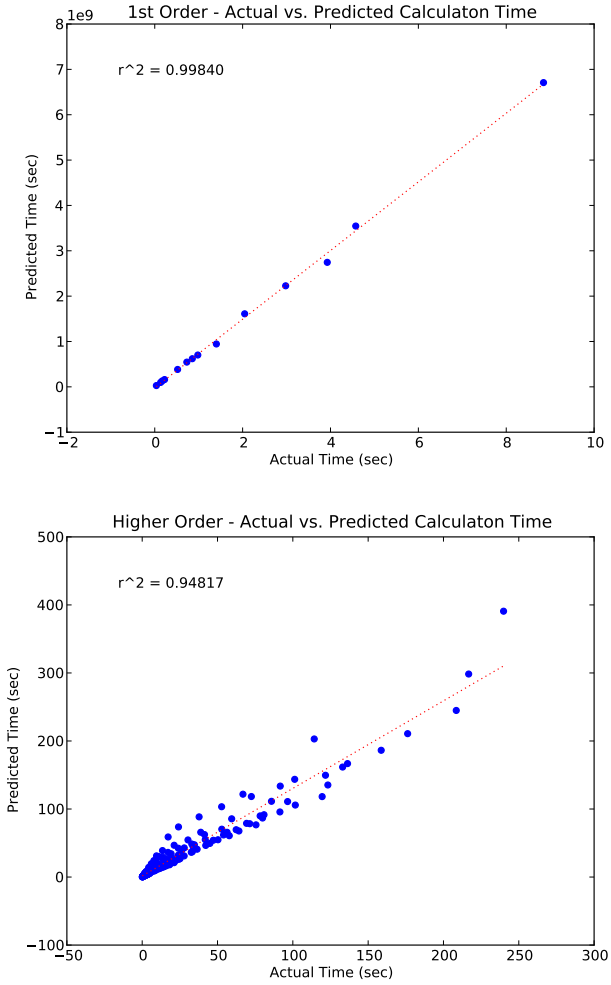


Figure 9: *Actual run time versus predicted run time.*

Comparison of 1st Order and Higher Order

In a forthcoming paper, Shinya Ito investigates the performance of D1TE, delayed TE (TEP-k/TECI), and two variants of normalized cross-correlation (called NCCPk and NCCCI) on simulation data from Izhikevich's neuron model [3]. He finds that TECI performs better on average, achieving a nearly 70% TPR when FPR = 0.01. TPR (true positive rate) = $\frac{TP}{TP+FN}$ and FPR (false positive) = $\frac{FP}{FP+TN}$ where TP, TN, FP, and FN are the counts of true positives, true negatives, false positives, and false negatives respectively.

Methods

Given that higher-order TE calculations are much more tractable with the algorithm above, it makes sense to investigate whether these kinds of calculations can offer better TPR/FPR values than first-order. To investigate this, I used the same data from Shinya's experiments and calculated TEPk, TECI, TEHoPk, and TEHoCI for the parameters listed in Figure 10 (all measures were computed over delays of 1 to 30).

Measure	i **Orders**	j **Orders**	**Windows**
TEPk	1	1	1
TECI	1	1	3,5,7,9
TEHoPk	1-5	1-5	1
TEHoCI	1-5	1-5	3,5,7,9

Figure 10: *Actual run time versus predicted run time.*

The simulation data was generated from code provided by Izhikevich in [3], modified slightly to turn off spike-time dependent plasticity after one hour. The model had 1000 neurons, 80% of which were regular spiking excitatory neurons and 20% of which were fast-spiking inhibitory neurons. Each neuron had 100 synaptic connections with varying delays for excitatory neurons (uniform distribution from 1-20 ms) and a fixed delay of 5 ms for inhibitory neurons. Inhibitory neurons were only connected to excitatory neurons, but excitatory neurons had no restrictions. While inhibitory synaptic weights were fixed at 5 mV, excitatory weights could change over time due to STDP in a range from 0 to 10 mV.

Eight simulations were run, each for two hours (with STDP turned off in the final hour to fix synaptic weights). Random "thalamic" input was provided to all neurons throughout the simulation. Only the final 30 minutes of spike train data was used to estimate connectivity. In addition, 100 neurons (80 excitatory, 20 inhibitory) were sub-sampled from the entire population to approximate electrode array recording. TE was calculated for these 100 neurons, with performance being evaluated based on the number of correctly identified connections whose absolute weights were considered significant ($\geq$ 1 mV).

Results

With the many different combinations of parameters, the final data set I obtained was high-dimensional, and thus difficult to visualize. As a first pass, it helped to do an ROC plot where curve each represented a particular combination of parameters. Figure 11 shows that no measure stands out immediately as "better", though TEHoPk appears qualitatively to have an advantage. There are many more parameter combinations for the higher-order measures, so TEPk and TECI are difficult to make out. This means, though, that neither first-order measure is qualitatively different from the higher-order measures.

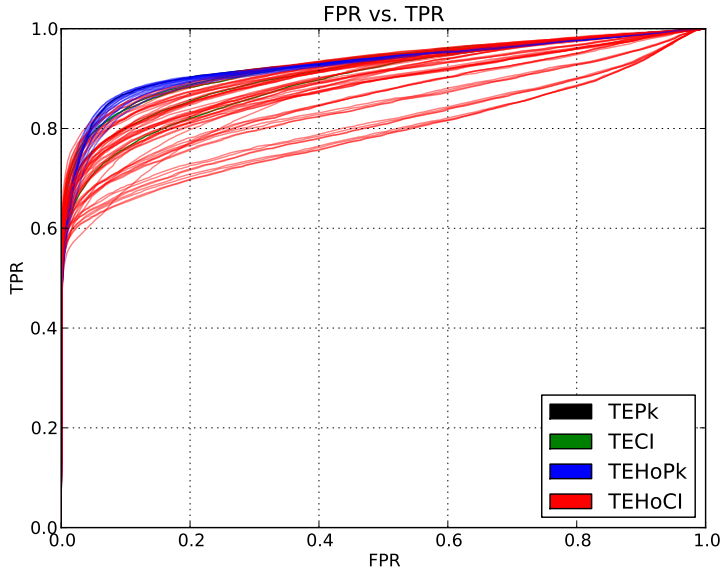


Figure 11: *ROC curves for every combination of parameters (averaged over all data sets).*

In order to compare measures more directly, I plotted the TPR values for each measure at an FPR of 0.01 (Figure 12). The "best" measure is labeled with its window size and order. Underneath the plot is a table with the top 20 measures, sorted descending by TPR.

From the plot and corresponding table, we can observe several things:

1. TEHoCI dominates with 17 of the top 20 slots. Only two first-order measures made it, and neither have an average TPR > 0.7 (#9 and #12).

2. Coincidence index is clearly a better reduction method. Only a single peak-based measure made it into the top 20 (#19).

3. The top TEHoCI measures have window sizes of 3 and 5 as well as orders ≤ 3 (remember that orders up to 5 were tested). This suggests that bumping up the calculation order just slightly could result in better performance.

Conclusions and Future Work

With this preliminary analysis, it appears that TEHoCI with orders ≤ 3 gives significantly better results than TEPk, TECI, and TEHoPk. Since the computational complexity of second or third order TE is close to that of first-order, it is recommended that TEHoCI be used.

For future work, it will be necessary to determine what kinds of connections are identified by the different measures (i.e. excitatory vs. inhibitory, synaptic weights). In addition, TEHoCI should be benchmarked against other methods on real-world data sets.

References

[1] M Garofalo, T Nieus, P Massobrio, and S Martinoia. Evaluation of the performance of information theory-based methods and cross-correlation to estimate the functional connectivity in cortical networks. *PLoS One*, 4:e6482, 2009.

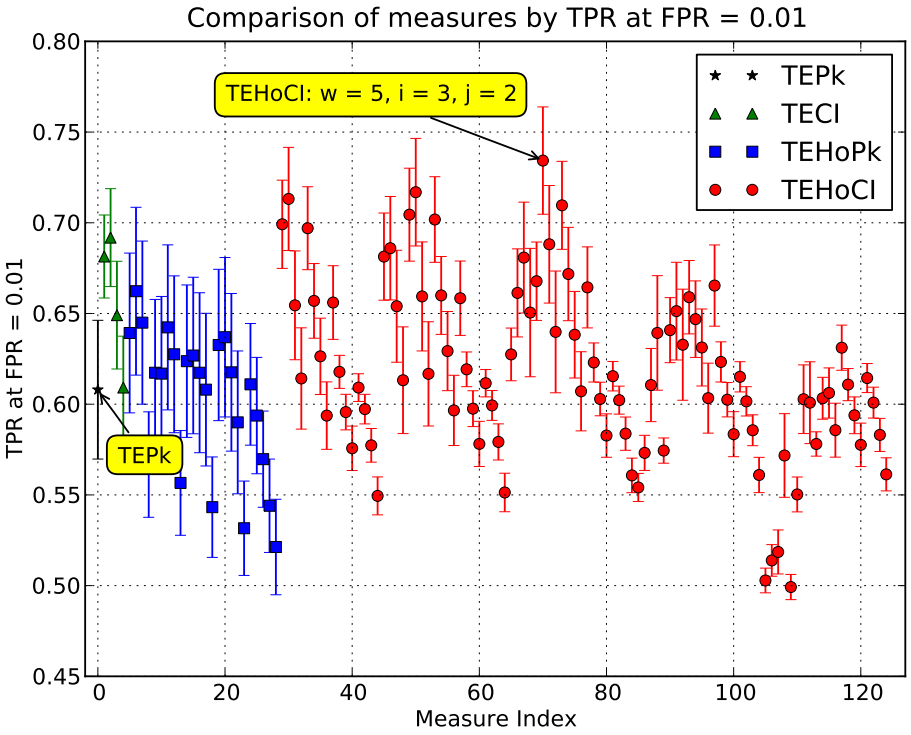
[2] Khronos Group. Opencl - the open standard for parallel programming of heterogeneous systems. `http://www.khronos.org/opencl/`, April 2011.

[3] EM Izhikevich. Polychronization: computation with spikes. *Neural Comput*, 18:245–82, Feb 2006.

[4] David Lunn, David Spiegelhalter, Andrew Thomas, and Nicky Best. The BUGS project: Evolution, critique and future directions. *Statist. Med.*, 28(25):3049–3067, November 2009.

[5] NVIDIA. Cuda zone - what is cuda? `http://www.nvidia.com/object/what_is_cuda_new.html`, April 2011.

[6] T Schreiber. Measuring information transfer. *Phys Rev Lett*, 85:461–4, Jul 2000.



	Measure	Window	i, j Order	Avg. TPR	Error
1	TEHoCI	5	3, 2	0.7343	0.0296
2	TEHoCI	5	2, 2	0.7169	0.0296
3	TEHoCI	5	1, 2	0.7132	0.0284
4	TEHoCI	3	3, 3	0.7096	0.0243
5	TEHoCI	3	2, 2	0.7045	0.0256
6	TEHoCI	3	2, 3	0.7018	0.0236
7	TEHoCI	3	1, 2	0.6992	0.0243
8	TEHoCI	3	1, 3	0.6970	0.0229
9	TECI	5	1, 1	0.6918	0.0270
10	TEHoCI	7	3, 2	0.6882	0.0324
11	TEHoCI	5	2, 1	0.6860	0.0285
12	TECI	3	1, 1	0.6814	0.0229
13	TEHoCI	3	2, 1	0.6814	0.0240
14	TEHoCI	7	3, 1	0.6808	0.0306
15	TEHoCI	5	3, 3	0.6718	0.0257
16	TEHoCI	3	3, 2	0.6678	0.0216
17	TEHoCI	3	4, 4	0.6654	0.0224
18	TEHoCI	3	3, 4	0.6644	0.0223
19	TEHoPk	1	1, 3	0.6623	0.0463
20	TEHoCI	5	3, 1	0.6614	0.0243

Figure 12: *Top: TPR of various measures with different sets of parameters. Bottom: details of top 20 measures. x-axis order is not meaningful.*