Identifying gene modules via convolutional neural networks

August 31, 2017

Outline

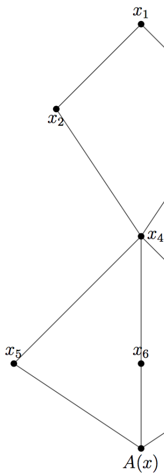
Introduction

- Interested in identifying elements of a signaling pathway
- Genetic perturbation $\rightarrow$ signaling stimulus $\rightarrow$ post-stimulus RNA levels
- If we wanted to test all single knockouts, this would require about 20,000 experiments.
- If there exists functional redundancy between two or three genes, the naive way would be to knock out all combinations of 2, 3 genes. Even if we restrict to 100 candidate genes, that gives us $\binom{100}{2} + \binom{100}{3} = 4,950 + 161,700$ different experiments.

Motivation: compressing the number of experiments

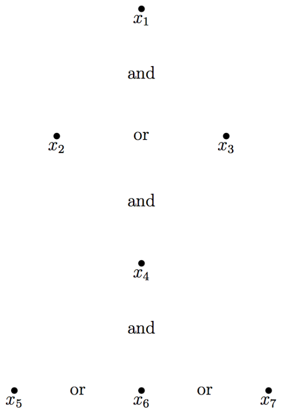
- If the transcriptional response of a given gene is regulated by a few modular components, then perturbing the elements in those components should lead to similar responses.
- In other words, many experiments should have correlated post-stimulus responses.
- We can use this assumption to justify compressing the number of experiments we perform, i.e. we can still learn signaling pathways without performing all combinations.
- For now, focus on learning pathways from uncompressed experiments.

Gene modules



- Assume that there are a few modular components that affect the activation/repression of a gene (or multiple genes).
- Each component is a module of with different layers (called elements). All elements are required to activate the module (functional dependency).
- Within an element, there may be multiple genes (functional redundancy). Eg: if gene 3 is knocked out but not gene 2, the module may still function.

Defining gene modules mathematically



- We can model functional redundancy as addition and functional dependency as multiplication.
- Eg: in this case model as $A(x) = x_1(x_2 + x_3)x_4(x_5 + x_6 + x_7)$. Thus we can go from gene modules to polynomials.
- Note: the order of the polynomial corresponds to the number of elements in the module.

Defining gene modules mathematically

- Recall module activity level is given by a polynomial, eg: $A(x) = x_1(x_2 + x_3)x_4(x_5 + x_6 + x_7)$.
- The space of possible polynomials can be very large. We can impose some structure on the polynomials we're interested in by casting it as a tensor.
- Each term in $A(x)$ above can be written as an inner product:

$$\text{e.g. } x_2 + x_3 = < v, x > \text{ where } v = \begin{pmatrix} 0 \\ 1 \\ 1 \\ 0 \\ 0 \\ 0 \\ 0 \end{pmatrix} \text{ and } x = \begin{pmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \\ x_6 \\ x_7 \end{pmatrix}$$

Why use tensors?

- Thus $A(x) = \prod_{i=1}^{d} < v_i, x > = < v_1 \otimes \ldots \otimes v_d, x^{\otimes^d} >$, where d is the order of the polynomial, and v_i are sparse.
- Tensor notation: recall that $v \otimes w = \begin{bmatrix} v_1 w_1 & v_1 w_2 \ldots v_1 w_J \\ & \vdots & \\ v_J w_1 & v_J w_2 \ldots v_J w_J \end{bmatrix}$
- To see why imposing structure on tensors restricts our state space, consider an quadratic polynomial on J variables, where all terms of the polynomial is of order 2, with no restrictions. Then there are $J^2/2$ coefficients that need to be specified.
- However, if we restrict to coefficient tensors of form $v \otimes w$, then our polynomial is $(v_1 x_1 + \ldots + v_J x_J)(w_1 x_1 + \ldots + w_J x_J)$. Here only $2J$ constants need to be specified. A similar argument applies to higher order polynomials.

Defining gene modules mathematically

- $A(x) = \prod_{i=1}^{d} < v_i, x > = < v_1 \otimes \ldots \otimes v_d, x^{\otimes^d} >$.
- Now let's suppose there are multiple modular components that affect a gene. Then we have

$$A(x) = \sum_{\ell=1}^{L} w_\ell \, A_\ell(x) = \sum_{\ell=1}^{L} < v_i^\ell \otimes \ldots \otimes v_d^\ell, x^{\otimes^d} > .$$

- The coefficients of the polynomial are thus represented by the tensor $\sum_{\ell=1}^{L} v_i^\ell \otimes \ldots \otimes v_d^\ell \in \mathbb{R}^{J^d}$ where J is the number of pre-stimulus genes. L is the number of components in the module, and corresponds to the tensor's rank.
- Finally, we model the post-stimulus transcriptional response $y(x) \in \mathbb{R}^G$ by $y_g \sim \text{Poiss}(m_g A(x) + b_g)$, where m_g is the weight of the components and b_g is a bias variable.

How do we learn?

- To learn the coefficients of a polynomial $A(x)$, we formulate the tensor as a single-layer convolutional neural network.
- The layer consists of a convolution step followed by a product pooling step. Nodes in the convolution step represent each inner product.
- Added an activation step for each component using a sigmoid function– this corresponds to $A(x) = \sum_{\ell=1}^{L} \sigma(A_\ell(x))$.
- To the whiteboard...

Some high level questions we want to answer:

- Can we reliably learn a set of functionally redundant and functionally dependent genes?
- If two genes are co-regulated by the same module, are we able to recover this from the algorithm?
- How well does a learned network predict y from x?
- How sensitive is the algorithm to random initialization?

Implementation

- A single-layer convolutional neural network was built using tensorflow.
- To test model, we generate simulated data. First generate a random sparse tensor to give us $A(x)$. Generate multiple samples of pre-stimulus gene profiles $x \sim Poisson(\lambda)$, and calculate $y(x)$ according to the tensor.
- Split into training and testing data to learn the model. We use train by minimizing the loss function

$$L = \frac{1}{N}\sum_{i=1}^{N} \|y_i - \hat{y}_i\|^2 + \lambda_1 \|V\|_1 + \lambda_2 \|P\|_1$$

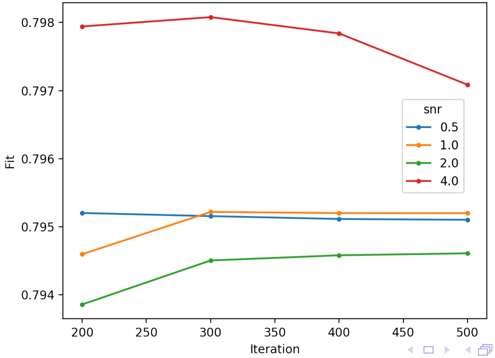
- For our initial tests, we generated 2000 samples from 100 experiments with $x \in \mathbb{R}^{12}$, $y \in \mathbb{R}^{20}$, and a tensor $T \in \mathbb{R}^{12\times12\times12}$ of rank 1 and order 3.

How do we evaluate?

- If we know the order of the polynomial ahead of time, we can easily calculate $\|V - \hat{V}\|$.
- If we don't know the order of the polynomial ahead of time, there is no norm we can use the compare tensors of different orders. Similarly, we might not know the rank of a tensor ahead of time (number of components in a module).
- Salvage: we might still be able to evaluate how well the learned model captures functional redundancies and dependencies between genes. We can measure the number of true/false positives and true/false negatives.
- We may also be interested simply in whether or not the model identifies the correct set of genes in a module, even if the exact structure is not preserved.

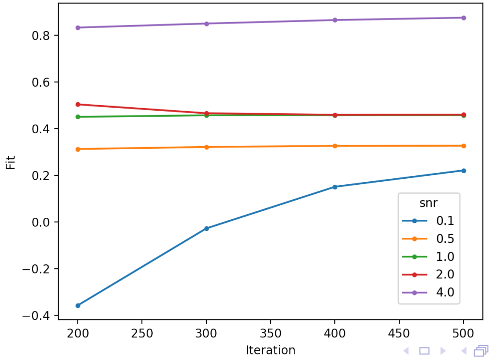
Preliminary results

- Implemented model as discussed and tested on simulated data. Tried initializing weights to true model with added noise (varying the signal-to-noise ratio), as well as completely random initialization.
- Predicts post-stimulus responses y from pre-stimulus levels x to some extent, but pretty mediocre.



Some technical details/difficulties

- However, the model doesn't do a very good job predicting the actual polynomial.
- Even if we know the order and rank of the polynomial ahead of time, the model doesn't always converge to the true network.
- In fact, the fit of the model doesn't seem to change much sometimes. Why?

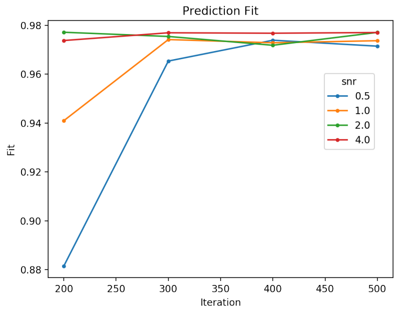


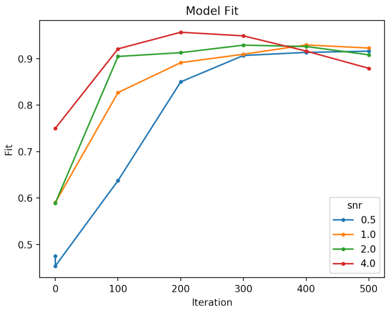
Refining our model

- Upon further investigation, it appears that many of the weights don't even get updated a lot of the time!
- The reason is has to do with the gradient of the sigmoid function. If the output of the sigmoid is close to 0 or 1, the gradient is essentially zero. Thus when we back propagate, none of the weights below that layer will get updated.
- One way to salvage this is to use rectified linear activation instead: $f(x) = max(0, x)$. If we want to stick with a sigmoid function, another possibility is to normalize the inputs so that the outputs from the layers should land in the dynamic window of the sigmoid function.

Refining our model

- With ReLU activation instead, the model performs much better.





Recovering functional redundancies:

- Raw output for weights from algorithm with $snr = 0.5$:

$$\begin{bmatrix} 7.82e-03 & 4.18e-03 & 3.10e-03 \\ 1.47e-02 & 3.49e-03 & 6.33e-01 \\ 3.12e-03 & 1.03e+00 & 4.89e-03 \\ 1.02e+00 & 2.03e-02 & 3.15e-04 \\ 9.22e-03 & 1.45e-04 & 3.93e-03 \\ 2.52e-04 & 6.27e-04 & 4.81e-03 \\ 1.22e+00 & 5.74e-03 & 1.03e+00 \\ 3.33e-03 & 1.01e+00 & 7.84e-04 \\ 1.05e+00 & 7.83e-03 & 1.37e-03 \\ 2.29e-03 & 1.15e-03 & 5.40e-03 \\ 1.35e-03 & 6.5e-03 & 1.32e+00 \\ 8.73e-03 & 1.01e+00 & 1.33e-03 \end{bmatrix}$$

- Note the regularization process made some weights smaller than others. We can sparsify this matrix by only selecting the weights with the highest orders of magnitude.

Recovering functional redundancies:

Sparsified predicted matrix

$$\begin{bmatrix} 0. & 0. & 0. \\ 0. & 0. & 0.633 \\ 0. & 1.037 & 0. \\ 1.021 & 0. & 0. \\ 0. & 0. & 0. \\ 0. & 0. & 0. \\ 1.229 & 0. & 1.032 \\ 0. & 1.019 & 0. \\ 1.052 & 0. & 0. \\ 0. & 0. & 0. \\ 0. & 0. & 1.32 \\ 0. & 1.019 & 0. \end{bmatrix}$$

Actual matrix

$$\begin{bmatrix} 0. & 0. & 0. \\ 0. & 0. & 0.743 \\ 0. & 1.194 & 0. \\ 1.099 & 0. & 0. \\ 0. & 0. & 0. \\ 0. & 0. & 0. \\ 1.276 & 0. & 1.154 \\ 0. & 1.156 & 0. \\ 1.133 & 0. & 0. \\ 0. & 0. & 0. \\ 0. & 0. & 1.496 \\ 0. & 1.166 & 0. \end{bmatrix}$$

- The functionally redundant genes are selected, even if the weights are not exactly the same.

Further work

- Generate data with higher rank tensors and test the predicted models.
- Evaluate whether or not learned networks can predict genes co-regulated by the same modules
- How can we modify our model to deal with compressed experiments?
- Adding additional layers to the convolutional network: this corresponds to a hierarchical tensor decomposition.

Hierarchical tensors

- In a hierarchical tensor decomposition, you express tensors as outer products of smaller order tensors. E.g.: an order-4 tensor T can be represented as the outer product of two order-2 tensors.

$$\begin{aligned} T &= \sum_{k=1}^{K} a_k v \otimes w \\ &= \sum_{k=1}^{K} a_k \left(\sum_{k=1}^{K} b_k v_k^1 \otimes v_k^2 \right) \otimes \left(\sum_{k=1}^{K} c_k w_k^1 \otimes w_k^2 \right) \end{aligned}$$

- T is of rank K^3, but you only need to choose $3K$ weights.
- The compact representation gives us more expressive power. To realize a hierarchical tensor with a shallow network, you need an exponential number of internal nodes in general.
- Downside: comparing two hierarchical tensors is even harder. All the difficulties with comparing shallow networks become even harder with a deep network.