Notes on learning gene modules

Anthea Cheung

September 5, 2017

Contents

1 Model

In this project, we aim to learn gene modules of signaling components in a signaling pathway. Briefly speaking, a module may consist of multiple components, with each component containing multiple elements. A single gene module may contribute to the activation or repression of a given gene. Genetic perturbations are applied by knocking out a select number of genes amongst or candidate genes. Each experiment is denoted $e \in \{0, 1\}^J$, where J is the number of candidate genes, and the observed pre-stimulus profiles are denoted $\boldsymbol{x} \in \mathbb{R}^J$.

We model the pre-stimulus profiles as $\boldsymbol{x} \sim \text{Poiss}(\boldsymbol{\lambda})$, where $\boldsymbol{\lambda}$ is a vector of expected values for each gene in a given experiment. We then model the post-stimulus RNA abundance levels $\boldsymbol{y} \in \mathbb{R}^G$ via $\boldsymbol{y} \sim \text{Poiss}(\boldsymbol{m}A(x) + \boldsymbol{b})$.

The gene module activity level is given by

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

Here f_ℓ is an activation function. Thus $A(x)$ corresponds to a tensor of rank L and order d.

Given this set-up, we would like to learn the parameters of a gene module. Thus, our objective is to learn the weights w_ℓ, (sparse) vectors v_i^ℓ, the module activity levels $\boldsymbol{m}$, biases $\boldsymbol{b}$, as well as any parameters governing the activation functions f_l, given a set of pre-stimulus profiles $\boldsymbol{x}$ from genetic perturbation experiments. The eventual goal is to be able to adapt such algorithms to incorporate compressed experiments, where the input profiles are pooled from a set of experiments $\mathcal{E}$: $\boldsymbol{x} = \sum_{e \in \mathcal{E}} \boldsymbol{x_e}$

2 Implementation

In order to learn the parameters of a gene module, we formulated the module into a convolutional neural network. The input layer is a layer with J nodes (corresponding to pre-stimulus profiles $\boldsymbol{x} \in \mathbb{R}^J$. The network architecture is as follows:

1. An input layer of J nodes.
2. A convolution step from applying inner products over the input layer. Each resulting node in the layer is the value of $< v_i^\ell, x >$ for each vector v_i^ℓ in the shallow decomposition of the tensor. The weights are stored in a tensor $V \in \mathbb{R}^{J \times L \times d}$.
3. A product pooling layer consisting of L nodes. Each node is the value of the product $\Pi_{i=1}^d < v_i^\ell, x >$. The variable *pool_weights* $\in \mathbb{R}^{L \times d}$ represent which elements are selected in each product.
4. An activation layer. Each node is the value of the $f_\ell(\Pi_{i=1}^d < v_i^\ell, x >)$. The offset variables for the activation layer are stored in a tensor *offset* $\in \mathbb{R}^{L \times d}$
5. Output layer with G nodes. Each node is the value of

$$y_g = m_g * \left(\sum_\ell w_\ell \; f_\ell \left(\Pi_{i=1}^d < v_i^\ell, x > \right) \right) + b_g$$

The weights for this step are stored in the tensors $m \in \mathbb{R}^G$ and $b \in \mathbb{R}^G$.

To train the model, we perform stochastic gradient descent using the loss function given by :

$$\mathcal{L} = \frac{1}{N}\sum_i \|\hat{y}_i - y_i\|^2 + \lambda_1\|V\|_1 + \lambda_2\|pool\_weights\|_1$$

Below, we give more specific implementation details based on our observations of what was the most conducive to learning the gene modules.

2.1 Product Pooling

Part of the network structure we would like to learn is which nodes in the convolution step get selected in product pooling. For example, if the original number of nodes in the convolutional layer is greater than the order of the tensor, not all of the nodes need to be included in the product pooling step.

Simply having a binary parameter for including or not including a particular node in the product pooling layer ends up not being conducive to learning the right structure, since the gradient for such a set-up ends up being a step function. Therefore, we have implemented a continuous version of product pooling as follows:

1. Assign pooling weights with values $\in \mathbb{R}$ to each node in the convolutional step.
2. Transform each convolutional node n via $n \to 1 - \sigma(w) + \sigma(w)n$, where w is the weight and $\sigma(\cdot)$ is the sigmoid function.
3. Product pool the resulting transformations.

The function $g(n, w) = 1 - \sigma(w) + \sigma(w)n$ gives a continuous output between 1 and n.

2.2 Activation function

The choice of activation also greatly affects how well the neural network is able to learn the weights. For example, when using a sigmoid function one needs to be careful with choosing the right offset amount and steepness so that the output of the function lands in the dynamic window of the sigmoid. From our experience with a single-layer shallow network, a ReLU activation function works much better for this reason. Still, there is some sensitivity to the initial offset that is selected in the initialization step.

If we extend the problem to multi-layered deep network, using ReLU activation in all layers can lead to the gradient becoming unmanageably large. Therefore, more care needs to be given to designing the right activations. One promising idea is to add sigmoid activation functions to each layer except the last, in order to control the magnitude of the output at each layer. Moreover this is biologically compelling as the output of sigmoidal functions can be interpreted as concentrations for kinetic reactions. The final layer may be contain a ReLU activation function. More work needs to be done in experimenting with different architectures in order to figure out which structure is most conducive to learning the correct weights (and more biologically feasible).

3 Metrics for Evaluating the Predicted Model

The convolutional neural network implementation of this model poses many challenges with regards to comparing the predicted model with the true model. For instance, we may not know ahead of time the correct rank and the correct order of the model. Thus, we cannot simply take the weights predicted by the CNN and do a direct comparison with the true model.

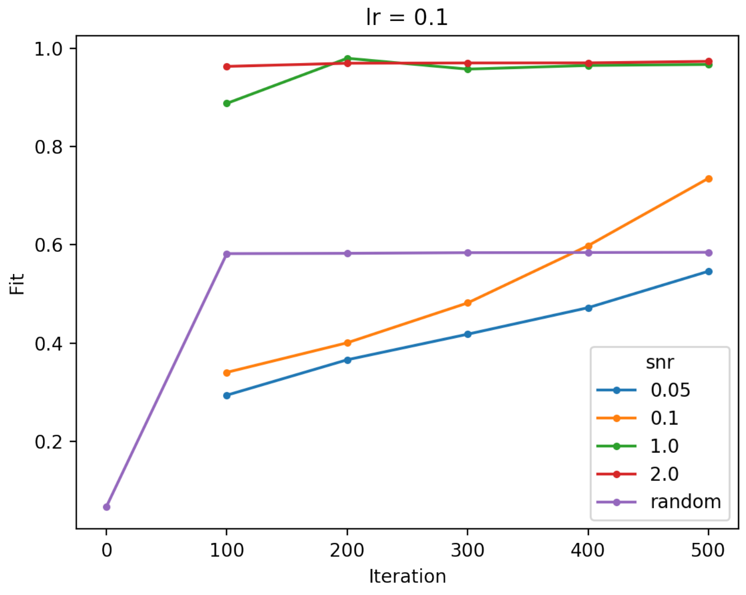
Results from the model also show that the convolution weights V are not necessarily k-sparse, although many weights are often orders of magnitude smaller than others. To actually interpret the model, we therefore take the final weights after training and apply a pruning step to set relatively small weights to zero. The pruning process is carried out in the following stpes:

1. Choose a threshold multiplier μ.
2. For each column in $v_i^\ell \in V$, let $m_i^\ell = \|v_i^\ell\|_0$ be the maximum absolute value of the vector.
3. Prune the column by setting all elements w in the vector such that $\mu|w| < m_i^\ell$ to zero.

After pruning the weights, we compare the pruned convolution weights to the true model. Since a direct comparison is not possible for misspecified model, we are interested in seeing: 1) how close is the predicted order of the tensor compared to the true model; 2) can the model recover functionally redundant sets of genes; and 3) does the model select the right subset of genes that are in a module? In questions 2) and 3), the criteria for goodness of fit is relaxed as the magnitudes of the weights are not compared.

To see if the model selects the right subset of genes, we simply compare the genes with nonzero weights in the true model and compare it with the predicted model. Since selecting a gene when it does not belong in the module is subjectively worse than not selecting a gene that belongs in a module, we penalize false positives more heavily than false negatives. One metric for evaluating this is given by:

$$score = \frac{\text{true positives}}{\text{true positives} + \text{false negatives} + 1.5 \cdot \text{false positives}}$$



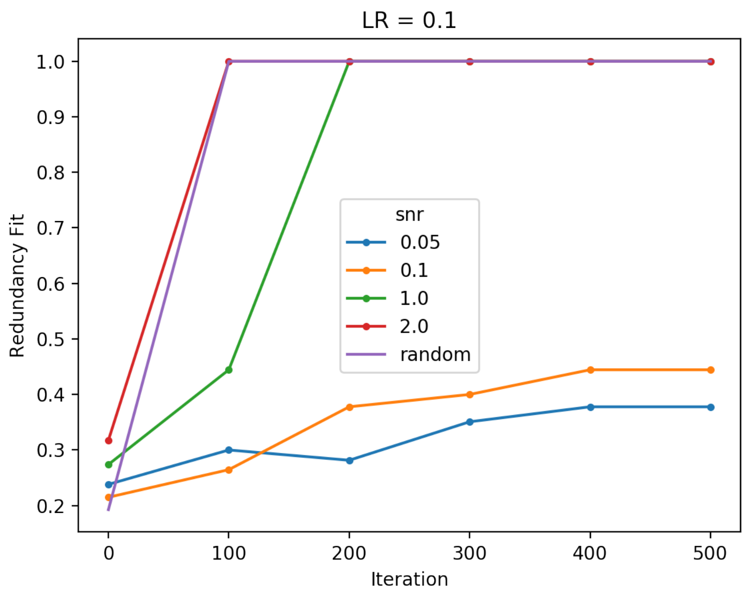
The relative weight of false negatives and false positives may be adjusted.

To evaluate how well a model recovers functional redundancies, we need to compare specific vectors to specify an element of a module, i.e. a column in the weight matrix V. Since the learned vectors may have been permuted from vectors in the true model, we compare the models by taking each vector v_i^ℓ, finding the closest vector in the predicted model, and then comparing the two vectors by the score above. The mean of the score over all vectors v_i^ℓ gives us a metric for evaluating the two models. Currently, this comparison has been implemented for rank-1 tensors. For higher rank tensors, we need to first select the best slice in $V \in \mathbb{R}^{J\times L\times d}$. While this has not been implemented yet, the subset score given above may be a rough way of estimating how to choose the best slice. Finally, we note that the assignment of vectors in the predicted model to a reference vector in the true model is not a unique assignment. Other approaches may be adopted to require a unique assignment when assessing functional redundancies.

4 Preliminary Results

5 Things yet to be implemented

A non-exhaustive list:



1. A smarter way for finding the reference vectors when the rank is greater than 1.
2. Support for multiple modules (right now we only support a single module with multiple sub-components.
3. Once multiple modules are supported, a way to assess how well the model recovers sets of co-regulated genes.
4. Better metrics for comparing models with rank greater than 1 (and when the rank is misspecified).
5. Implementing expected value of loss, assuming that the outputs y_i are Poisson random variables.
6. Incorporating compressed experiments.