Notes on learning gene modules

Anthea Cheung

August 31, 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 $b \in \mathbb{R}^G$.

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:

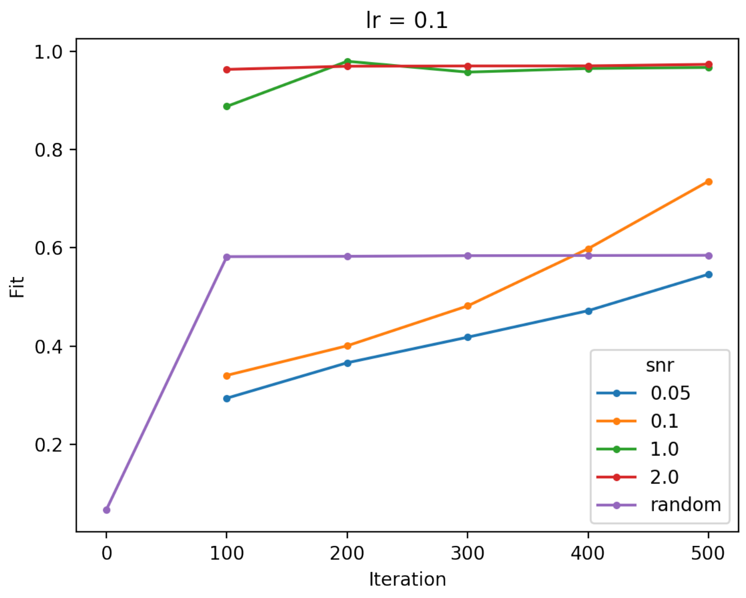
1. Assign pooling weights with values $\in \mathbb{R}$ to each node in the convolutional step.
2. Transform each convolutional node n via $n \rightarrow 1 - \sigma(w) + \sigma(w)n$, where w is the weight and $\sigma(\cdot)$ is the sigmoid function.
3. Product pool the resulting transformations.

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.

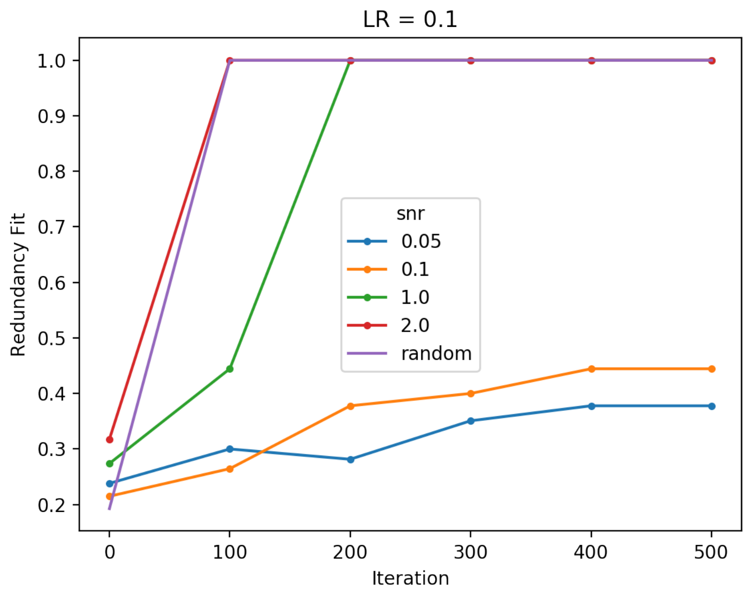
If we extend the problem to multi-layered deep network, using ReLU activation in all layers can lead to the gradient becoming unmanageably large. Therefore, more care needs to be given to designing the right activations. One promising idea is to add sigmoid activation functions to each layer except the 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

4 Preliminary Results

5 Things yet to be implemented


LR = 0.1
Redundancy Fit
Iteration
snr
0.05
0.1
1.0
2.0
random