Incremental Few-Shot Learning with Attention Attractor Networks

Chen Zhao

NeurIPS 2019 (`https://arxiv.org/pdf/1810.07218.pdf`)

1 Key ideas

1. Graph Neural Networks with multiple types of entities and relations are useful for modeling dynamical systems.
2. We meta-learn a set of neural modules that allow us to model many dynamical systems after inferring their structure.
3. Model-based approach to relational inference is more data efficient and allows inferences for which it wasn't trained.
4. We scale up modular meta-learning, from 100 to 50k datasets, by learning a proposal function for Simulated Annealing

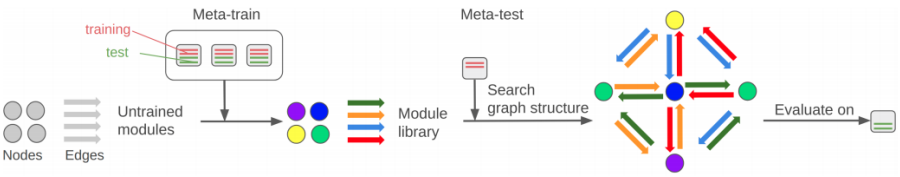


Figure 1: Modular meta-learning with graph networks; adapted from Alet et al. (2018). The system meta-learns a library of node and edge modules, represented as small neural networks; at performance (meta-test) time, it is only necessary to infer the combination of modules that best predict the observed data for the system, and use that GNN to predict further system evolution.

2 Contributions

- A model-based approach to neural relational inference by framing it as a modular meta-learning problem.

- Speeding up modular meta-learning by two orders of magnitude, allowing it to scale to big datasets and modular compositions.
- We propose to leverage meta-data coming from each inner optimization during meta-training to simultaneously learn to learn and learn to optimize.

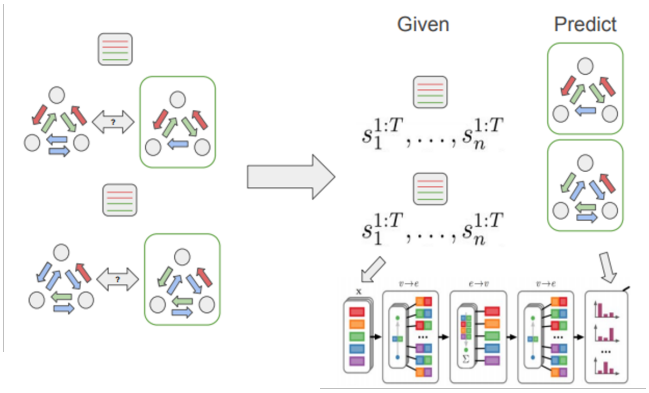
3 Methods

Consider a set of n known entities with states that evolve over T time steps: $s_1^{1:T}, ..., s_n^{1:T}$. Let $\mathcal{G}$ be a graph, with nodes $v_1, ..., v_n$ and $e_1, ..., e_{r'}$. Let S be a structure detailing a mapping from each node to its corresponding node module and from each edge to its corresponding edge module. We can now run several steps of message passing: in each step, nodes read incoming messages from their neighbors and sum them, to then update their own states. The message μ_{ij} from node i to j is computed using the edge module determined by $S, m_{S_{ij}}$, which takes the states of nodes i and j as input, so $\mu_{ij}^t = m_{S_{ij}}(s_i^t, s_j^t)$. The state of each node is then updated using its own neural network module m_{S_i}, which takes as input the sum of its incoming message,

$$s_i^{t+1} = s_i^t + m_{S_i}(s_i^t, \sum_{j \in neigh(v_i)} \mu_{ji}^t) \tag{1}$$

We apply this procedure T times to get $s^{t+1}, ..., s^T$; the whole process is differentiable, allowing us to train the parameters of $m_{S_i}, m_{S_{ij}}$ end-to-end based on predictive loss.

Instead of proposing random changes, a neural network learns to make good proposals by imitating current structures found by Simulated Annealing. Simulated Annealing uses the proposal function to make good proposals for each dataset, accepting them depending on performance. Neural network learns to predict structures from datasets, providing a prior for proposal function.



Modular meta-learning learns a set of small neural network modules and forms hypotheses by composing them into different structures. $\Theta = (\theta_1, ..., \theta_k)$ are the weights of modules $m_1, ..., m_k$, and the algorithm A operates by searching over the set of possible structures S to find the one that best fits $\mathcal{D}_{train}$ and applies it to $\mathbf{x}_{test}$. Let $h_{S,\Theta}$ be the function that predicts the output using the modular structure S and parameters Θ. Then

$$A(\mathcal{D}_{train}, \Theta) = h_{S^*,\Theta} \quad \text{where} \quad S^* = \arg\min_{S \in \mathcal{S}} \mathcal{L}(h_{S,\Theta}(\mathbf{x}_{train}), \mathbf{y}_{train}) \tag{2}$$

At meta-training time we have to find module weights $\theta_1, ..., \theta_m$ that compose well. . To do this, we proposed the BOUNCEGRAD algorithm (Alet et al., 2018) to optimize the modules and find the structure for each task. It works by alternating steps of simulated annealing and gradient descent. Simulated annealing (a stochastic combinatorial optimization algorithm) optimizes the structure of each task using its train split. Gradient descent steps optimize module weights with the test split, pooling gradients from each instance of a module applied to different tasks. At meta-test time, it has access to the final training data set, which it uses to perform structure search to arrive at a final hypothesis.