Learning to learn by GD by GD

Yuzhe Ou

Sep 2020

1 Introduction

The trend of automate the artificial learning process is important for future deep learning algorithm. In meta-learning, we model the component of a learning algorithm as in a family of options parameterized by some meta parameters ϕ. This includes MAML (to learn global initialization), NAS (to learn neural architecture) etc. However, usually optimization algorithms in learning process is designed by hand. A lot of optimization methods have been proposed for particular problems (high-dimensional, non-convex) to achieve better performance, including momentum, Rprop, Adagrad, RMSprop and ADAM. No Free Lunch Theorems for Optimization suggests that specialization to a subclass of problems is the only way to improve performance.

2 Learning to learn with RNN

The standard approach used in deep learning is to do gradient descent (GD) to update the parameters w.r.t. some objective function $f(\theta_t)$:

$$\theta_{t+1} = \theta_t - \alpha_t \nabla f(\theta_t) \tag{1}$$

The standard GD only makes use of gradients and ignores higher order information. In the paper author propose to improve this by placing a learned update rule:

$$\theta_{t+1} = \theta_t + g_t(\nabla f(\theta_t), \phi) \tag{2}$$

where the optimizer is directly parameterized by ϕ. Therefore in a meta-learning framework the task specific gradients can be learned. Given a distribution of functions f the expected loss writes

$$\mathcal{L}(\phi) = \mathbb{E}_f[f(\theta^*(f, \phi))] \tag{3}$$

where $\theta^*(f, \phi)$ is the final optimizee parameters, as a function of optimizer parameter ϕ and objective function f. The left is to define the update rule in detail parameterized by meta parameter ϕ. This is modeled with a Recurrent Neural Network (RNN) denoted as m:

$$\mathcal{L}(\phi) = \mathbb{E}_f\left[\sum_{t=1}^{T} w_t f(\theta_t)\right] \tag{4}$$

where

$$\theta_{t+1} = \theta_t + g_t, \ \begin{bmatrix} g_t \\ h_{t+1} \end{bmatrix} = m(\nabla_t, h_t, \phi), \ \nabla_t = \nabla_\theta(f(\theta_t))$$

In such way we want to learn an RNN that gives gradient for each time-step t using information of historical gradient information ∇_t. Here $w_t \in \mathbb{R}_{\geq 0}$ are weights associated with each time-step. This is equivalent to (3) when $w_t = \mathbf{1}[t = T]$. The The ϕ can be updated using GD to minimize $\mathcal{L}(\phi)$. The gradient $\partial\mathcal{L}(\phi)/\partial\phi$ can be estimated by sampling random f and applying backpropagation to the computational graph in the following figure.

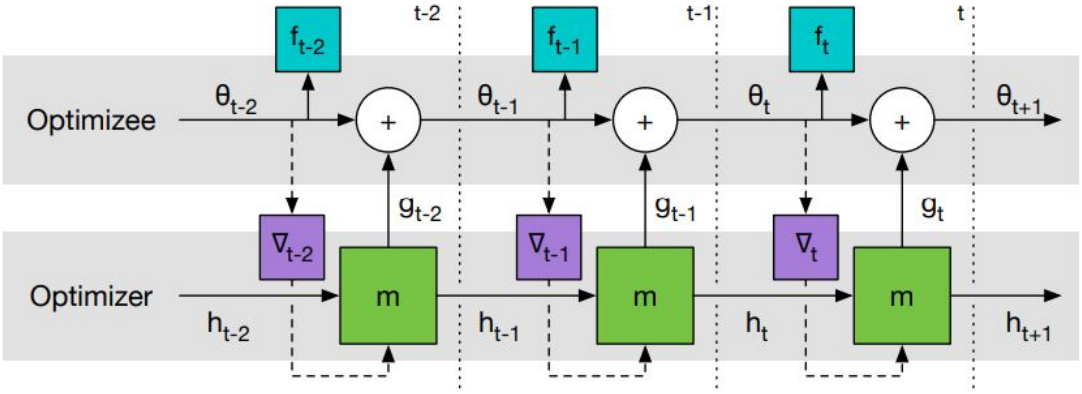

Figure 1: Computational graph used for computing the gradient of the optimizer

The gradients only flow along the solid edges in the graph. The relaxed objective (4) allow us training the optimizer on partial trajectories. For simplicity, we can choose $w_t = 1$ for every t.

3 Coordinatewise LSTM optimizer

However, directly applying fully connected RNN will face challenge of large parameter scale. The author proposed a coordinatewise network architecture that use a very small network that only looks at a single coordinate to define the optimizer and share optimizer parameters across different parameters of the optimizee.

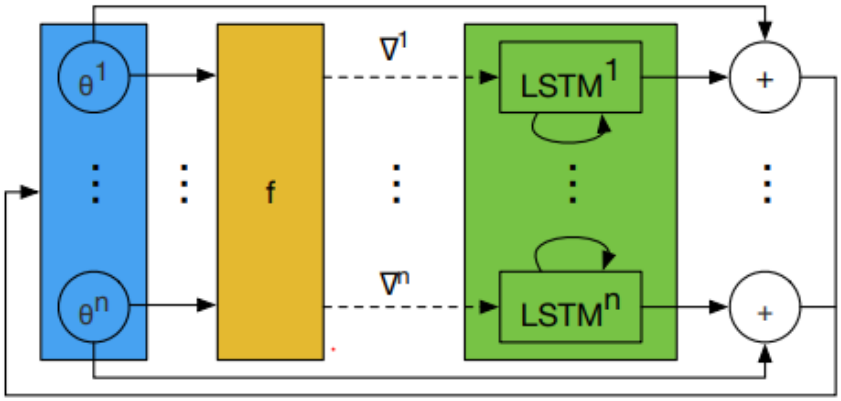

Figure 2: One step of an LSTM optimizer.

In the above graph shows the structure of optimizer, all LSTMs have shared parameters but separate hidden states. The use of recurrence allows the LSTM to learn dynamic update rules whichi integrate information from the history of gradients, similar to momentum, which then has favorable properties.