export_on_save:

html: true

Learning to learn by gradient descent by gradient descent

Motivation

The paper we are looking at today is thus trying to replace the optimizers normally used for neural networks (eg Adam, RMSprop, SGD etc.) by a recurrent neural network: after all, gradient descent is fundamentally a sequence of updates (from the output layer of the neural net back to the input), in between which a state must be stored. We can think of an optimizer as a mini-RNN. The idea in this paper is to actually train that RNN instead of using a generic algorithm like Adam/SGD/etc.

Contribution

We have shown how to cast the design of optimization algorithms as a learning problem, which enables us to train optimizers that are specialized to particular classes of functions.

Learning to learn with recurrent neural networks

In this work we consider directly parameterizing the optimizer. As a result, in a slight abuse of notation we will write the final *optimize parameters θ∗(f, φ)* as a function of the optimizer parameters φ and the function in question. We can then ask the question: What does it mean for an optimizer to be good? ? Given a distribution of functions f we will write the expected loss as

$$\mathcal{L}(\phi) = \mathbb{E}_f \Big[f\big(\theta^*(f, \phi)\big) \Big] . \tag{2}$$

As noted earlier, we will take the update steps gt to be the output of a recurrent neural network m, parameterized by φ, whose state we will denote explicitly with ht. Next, while the objective function in Eq/(2) depends only on the final parameter value, for training the optimizer it will be convenient to have an objective that depends on the entire trajectory of optimization, for some horizon T

$$\mathcal{L}(\phi) = \mathbb{E}_f \left[\sum_{t=1}^{T} w_t f(\theta_t) \right] \qquad \text{where} \qquad \begin{aligned} \theta_{t+1} &= \theta_t + g_t , \\ \begin{bmatrix} g_t \\ h_{t+1} \end{bmatrix} &= m(\nabla_t, h_t, \phi) . \end{aligned} \tag{3}$$

This formulation is equivalent to (2) when wt = 1[t = T], but later we will describe why using different weights can prove useful. The loss function described in the paper seems complicated, but in reality it is very simple: **all it is saying is that the loss of the optimizer is the sum of the losses of the optimize as it learns**. The paper includes some notion of weighing but gives a weight of 1 to everything, so that it indeed is just the sum. And f is the optimizee function, and θt is its parameters at time t. m is the optimizer

function, ϕ is its parameters. ht is its state at time t. gt is the update it outputs at time t. The plan is thus to use gradient descent on ϕ in order to minimize L(ϕ), which should give us an optimizer that is capable of optimizing f efficiently. As the paper mention, it is important that the gradients in dashed lines in the figure below are not propagated during gradient descent.

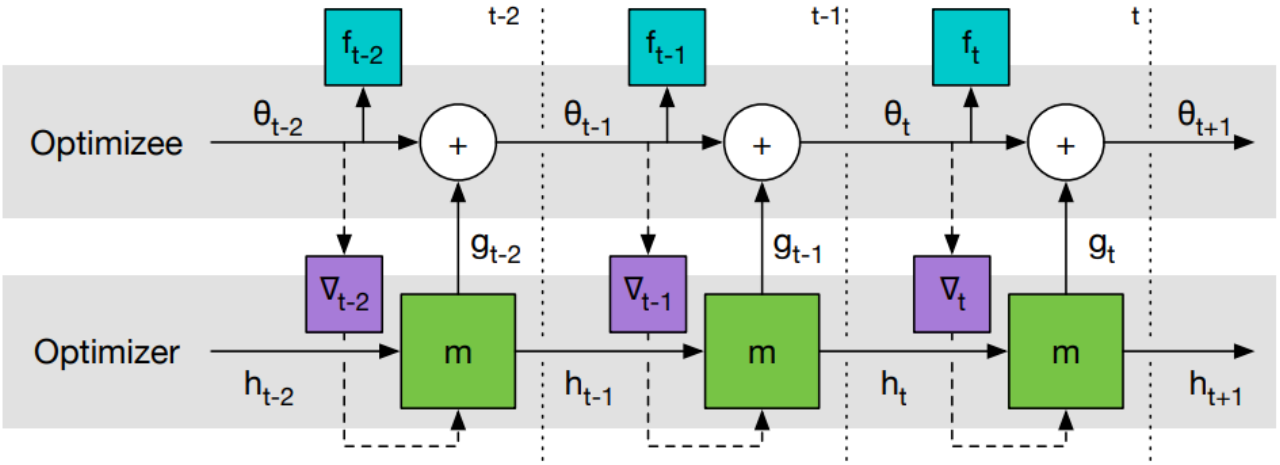

Figure 2: Computational graph used for computing the gradient of the optimizer.

Coordinatewise LSTM optimizer

One challenge in applying RNNs in our setting is that *we want to be able to optimize at least tens of thousands of parameters*. Optimizing at this scale with a fully connected RNN is not feasible as it would require a huge hidden state and an enormous number of parameters. To avoid this difficulty we will use an optimizer m which operates coordinatewise on the parameters of the objective function, similar to other common update rules like RMSprop and ADAM. This coordinatewise network architecture allows us to 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, shown in Figure 3.

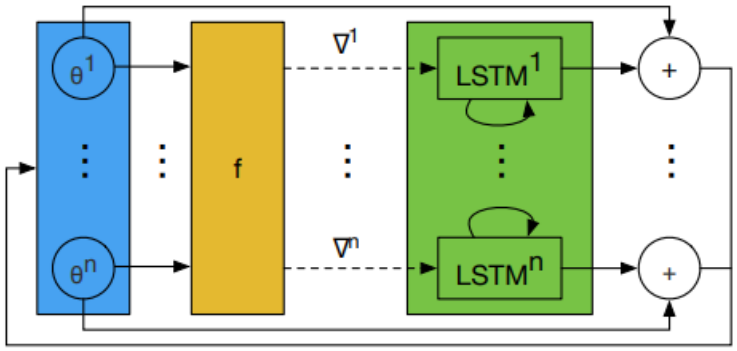

Figure 3: One step of an LSTM optimizer. All LSTMs have shared parameters, but separate hidden states.

Different behavior on each coordinate is achieved by using separate activations for each objective function parameter. In addition to allowing us to use a small network for this optimizer, this setup has the nice effect of making the optimizer invariant to the order of parameters in the network, since the same update rule is used independently on each coordinate. It is actually quite simple: what it

means is simply this: *every single “coordinate” has its own state (though the optimizer itself is shared), and information is not shared across coordinates*.