Learning to Learn

Presenter: Ceyer Wakilpoor

Nando de Freitas

Departmen of Computer Science, University of Oxford

June 2017

Outline

Learning to Learn

- A lot of competitions in deep learning applications, imagine next big problem to solve
- Look to a very small phenomena of the human brain - we know how to learn
  - Child playing and figuring out a simple puzzle
  - Knowing to taste and touch things, knowing to try to get sensory information
  - Evolution and community also serve as learning methods, there is learning at a lot of timescales

Motivation

- DNNs work very well when given a lot of data, but are not necessarily good at figuring out how to learn from few data, or how to learn optimally
- How can a neural network be used to learn another neural network
- Cases of learning to learn
    - MCMC sampling
    - NN making samples for another NN
    - NN generate the parameters and/or architecture for another NN
    - Programmable NNs
    - NN controlling behavior of another NN, like reinforcement learning, gating (choosing bias, activation, etc)
    - Learning optimization algorithms
- Common practice of taking GD, applying transformation and seeing if it performs better on some popular data set
- Engineering optimizers is like feature engineering

Outline

Learning Optimization Algorithms

- Using a NN to adjust the parameters of another NN
- Should be treated as one NN in the end, no more algorithm for a network, just one "dynamic" NN

The Model

- Two networks, an optimizer and an optimizee, for example:
  - optimizee, f, implements a conv-net
  - optimizer, an RNN that gives f gradients and other information
- Take a parameter, run through optimizee, get gradient, plug into optimizer which gives update for the initial parameter, and repeat with another parameter
- The optimizer can be transferred to different optimizees
- g: optimizer, ϕ: optimizer's parameters, f: optimizee, θ: optimizee parameters

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

The Model

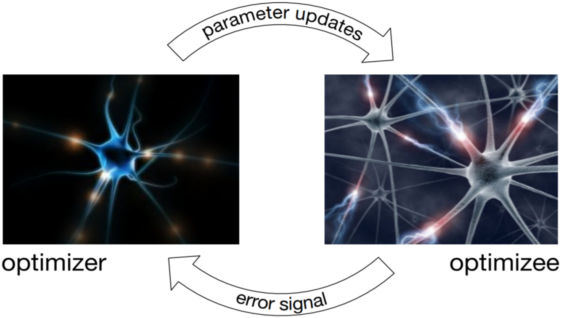


Figure 1: The optimizer (left) is provided with performance of the optimizee (right) and proposes updates to increase the optimizee's performance. [photos: Bobolas, 2009, Maley, 2011]

The Model

- Optimizee parameters, $\theta^*(f, \phi)$, as a function of optimizer parameters ϕ, yields the loss function (which is minimized by gradient descent):

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

- g_t is the output of RNN, m, parametrized by ϕ, whose state is denoted by h_t

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

where

$$\theta_{t+1} = \theta_t + g_t$$

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

The Model

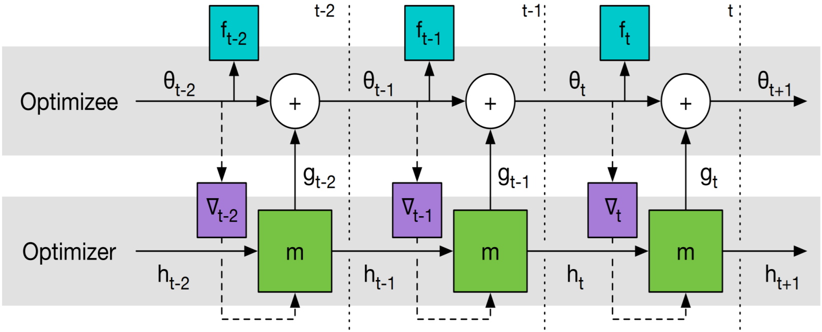


Figure 2: Computational graph used for computing the gradient of the optimizer.

The Model

- Use coordinatewise network architecture in order to allow one optimizer to learn updates for all of the optimizees parameters
- Otherwise optimizer would require a hidden layer for each parameter; when replicated over all the hidden states, this becomes too large

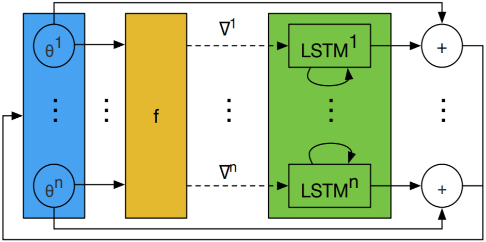


Figure 3: One step of an LSTM optimizer. All LSTMs have shared parameters, but separate hidden states.

Transferring the Optimizer

- Goal was to minimize:

$$f(\theta) = ||W\theta - y||_2^2$$

- Trained small network with 20 hidden units on MNIST
- Optimizer had 100 optimization steps (number of timesteps for RNN)
  - Performed better than RMSprop, Adam, etc
- Transferred to 40 units successfully
- Unrolled to 200 timesteps and tried on optimizee with 2 layers, and finally optimizee with using RelU instead of TanH

Results

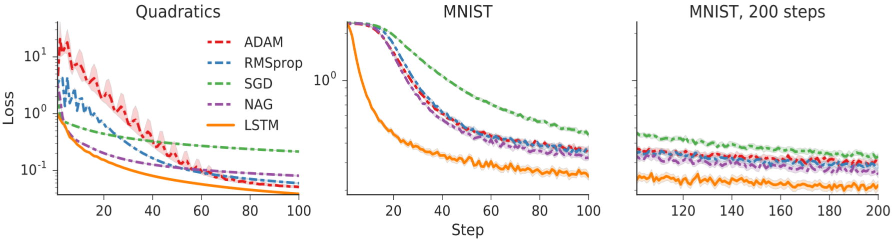


Figure 4: Comparisons between learned and hand-crafted optimizers performance. Learned optimizers are shown with solid lines and hand-crafted optimizers are shown with dashed lines. Units for the y axis in the MNIST plots are logits. **Left:** Performance of different optimizers on randomly sampled 10-dimensional quadratic functions. **Center:** the LSTM optimizer outperforms standard methods training the base network on MNIST. **Right:** Learning curves for steps 100-200 by an optimizer trained to optimize for 100 steps (continuation of center plot).

Results

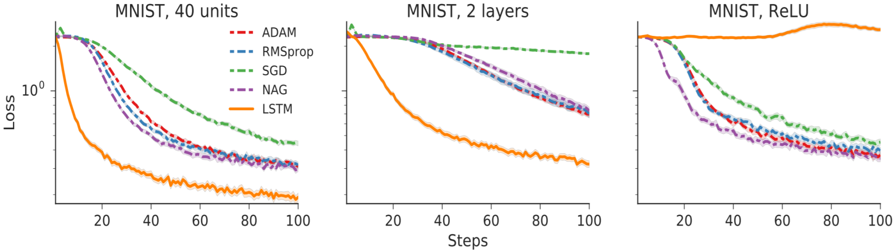


Figure 5: Comparisons between learned and hand-crafted optimizers performance. Units for the y axis are logits. **Left:** Generalization to the different number of hidden units (40 instead of 20). **Center:** Generalization to the different number of hidden layers (2 instead of 1). This optimization problem is very hard, because the hidden layers are very narrow. **Right:** Training curves for an MLP with 20 hidden units using ReLU activations. The LSTM optimizer was trained on an MLP with sigmoid activations.

Limitations

- Did not perform well when optimizee was using ReLU activation functions
- Difficult with large number of parameters
- Difficult task, evolution (a very expensive operation) was needed to teach humans how to learn
- A trained optimizer will have no hyper parameters, but does need to be trained using classical optimization methods
- Usually can't generalize to loss functions it wasn't trained on

Outline

Scaling and Generalizing

- Train on lots of different data sets, random functions, fundamental optimization functions, etc
- More variation instead of a lot of data sets from the same domain
- Used hierarchical LSTM
- Utilize optimization insights like normalization
- Training on different lengths
- Used truncated back-propagation

Hierarchical RNN

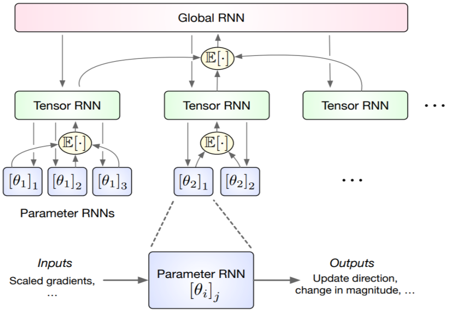
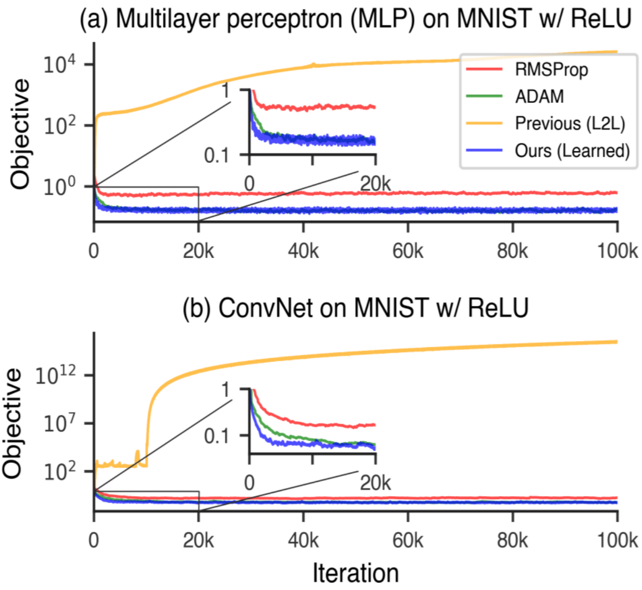


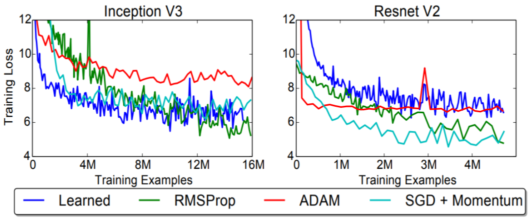
Figure 1. Hierarchical RNN architecture. At the lowest level, a small Parameter RNN processes the inputs and outputs (Section 3.3) for every parameter (θ_{ij}) in the target problem. At the intermediate level, a medium-sized Tensor RNN exists for every parameter tensor (denoted by θ_i) in the target problem. It takes as input the average latent state across all Parameter RNNs belonging to the same tensor. Its output enters those same Parameter RNNs as a bias term. At the top level, a single Global RNN receives as input the average hidden state of all Parameter RNNs, and its output enters the Tensor RNNs as a bias term and is added to the Parameter RNN bias term. This architecture has low per-parameter overhead, while the Tensor RNNs are able to capture inter-parameter dependencies, and the Global RNN is able to capture inter-tensor dependencies.

Results



(a) Multilayer perceptron (MLP) on MNIST w/ ReLU
Objective
10^4
10^2
10^0
1
0.1
0
20k
RMSProp
ADAM
Previous (L2L)
Ours (Learned)
0
20k
40k
60k
80k
100k
(b) ConvNet on MNIST w/ ReLU
Objective
10^{12}
10^7
10^2
1
0.1
0
20k
0
20k
40k
60k
80k
100k
Iteration

Results



- More robust to different learning rates

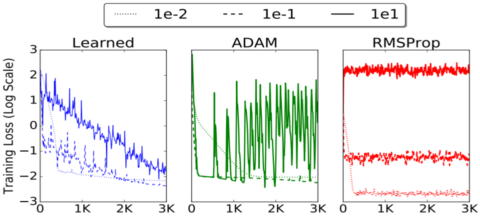


Figure 5. Learned optimizer performance is robust to learning rate hyperparameter. Training curves on a randomly generated

Outline

Model

- Motivation from Bayesian optimization
- In Gaussian optimization you want to sample the function at certain points to determine the function and find the optimum
- Useful technique for turning NN hyperparameters
- Idea is to utilize the optimizer RNN to do this instead
- Trained on simple Gaussian Process functions
- GP generates a continuous domain where every point in some input space is from a normally distributed random variable

Model

- Notion of balancing exploiting a good direction vs exploring new ones
- Teaching optimizer to sample from limited sampling size, how to learn from a few

$$L_{sum}(\theta) = \mathbb{E}_{f,y_{1:T-1}}[\sum_{t=1}^{T} f(x_t)]$$

$$L_{EI}(\theta) = -\mathbb{E}_{f,y_{1:T-1}}[\sum_{t=1}^{T} EI(x_t|y_{1:t-1})]$$

$$L_{OI}(\theta) = \mathbb{E}_{f,y_{1:T-1}}[\sum_{t=1}^{T} min\{f(x_t) - min_{i<t}(f(x_i)), 0\}]$$

Model

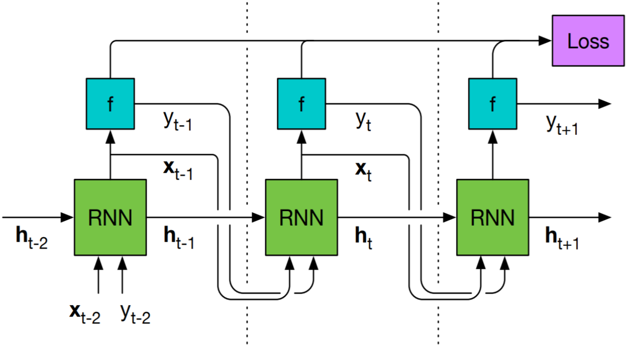


Figure 1. Computational graph of the learned black-box optimizer unrolled over multiple steps. The learning process will consist of differentiating the given loss with respect to the RNN parameters

Results

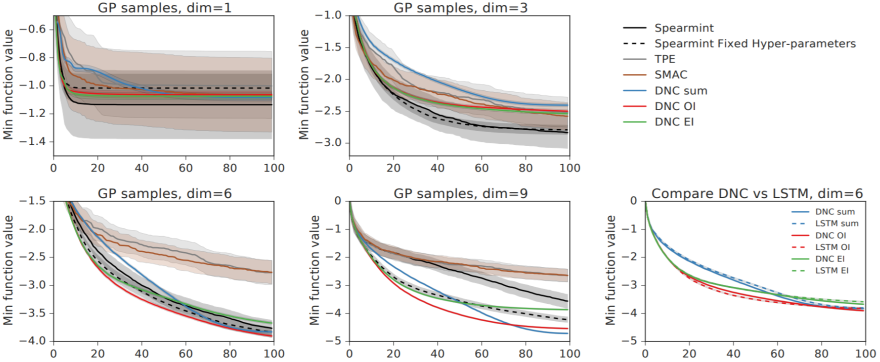


Figure 3. Average minimum observed function value, with 95% confidence intervals, as a function of search steps on functions sampled from the training GP distribution. Left four figures: Comparing DNC with different reward functions against Spearmint with fixed and estimated GP hyper-parameters, TPE and SMAC. Right bottom: Comparing different DNCs and LSTMs. As the dimension of the search space increases, the DNC's performance improves relative to the baselines.

Outline

1. Introduction
   - Learning to Learn
   - Motivation
2. Learning Optimization Algorithms
   - The Model
   - Experiments
   - Limitations
3. Learning Optimizers that Scale and Generalize
   - Model
   - Results
4. Learning to Learn without Gradient Descent by Gradient Descent
5. Few-Shot Learning
6. Reinforcement Learning

Few-Shot Learning

- Same model as before done again with 5 images of training, and 2 for test
- Repeat with a lot of small data sets

Outline

Reinforcement Learning For NN Architecture

- Optimizer RNN generate structure for another optimizee RNN
- Optimizer will receive reward based on how the optimizee does
- Very expensive task

One-Shot Reinforcement Learning

- Problem with a lot more variance
- Take policy network and condition off of demonstration
- When given a new demonstration at test time, model has learned how to react to it
- Model is trained to imitate demonstration

Citations

- https://arxiv.org/pdf/1703.07326.pdf
- https://openreview.net/pdf?id=rJY0-Kcll
- https://arxiv.org/abs/1611.05763
- https://arxiv.org/abs/1606.04474
- https://arxiv.org/pdf/1703.04813.pdf