Learning to learn by gradient descend by gradient descend

Paper overview by A. Khachiyants

Marcin Andrychowicz et al.

24 April 2018

Table of contents

Back to basics

- Typical machine learning problem — optimize objective function:

$$f(\theta) \to \min_{\theta \in \Theta}.$$

- Typical solution — use gradient descent:

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

But my second-order information!

- Gradient descent is slow.
- Can be corrected with rescaling step using information about curvature:
  - Hessian matrix;
  - Generalized Gauss-Newton matrix (Gauss-Newton method);
  - Fisher information matrix.
- Computing second-order information for neural network is a pain in the ass.

How to optimize

- Nowadays there are lots of methods for high-dimensional, non-convex optimization:
  - Momentum
  - Rprop
  - Adagrad
  - RMSprop
  - ADAM
  - insert your variation of GD
- If you know the structure of optimization problem, you can use specialized methods.
- If the problem is sparse, there are other specific approaches.
- Oh boy, combinatorial optimization.

Pay for your lunch

- If you don't exploit the structure of your problem, you are doomed.

Theorem (Wolpert and Macready, 1997)

In the setting of combinatorial optimization, no algorithm is able to do better than a random strategy in expectation.

- What to do? Learn how to exploit the structure.

But I'm a lazy bum

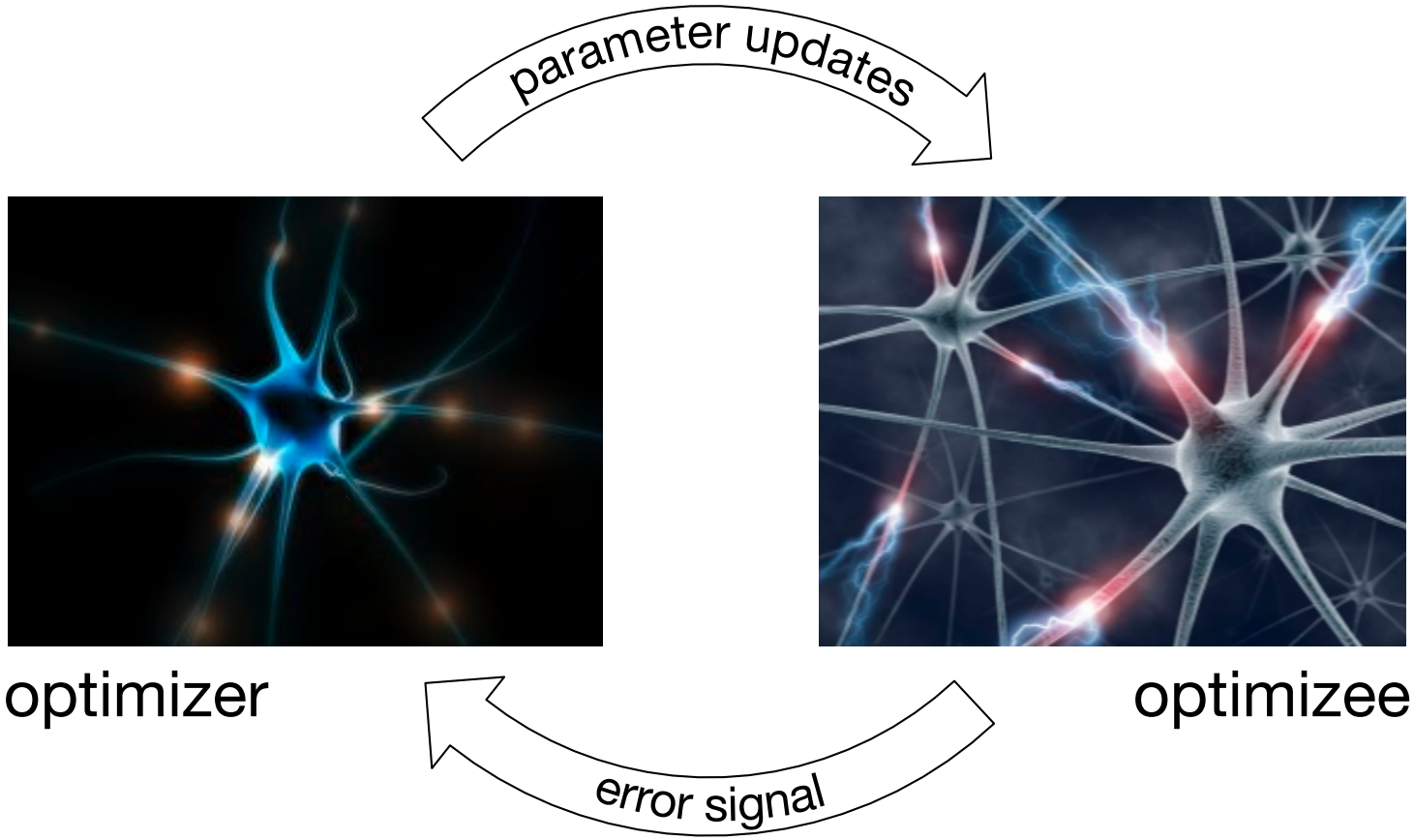
- Rather than exploiting the structure yourself, let the recurrent neural network do the job and propose the learned update rule g:

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

- Definitions: g is *optimizer*, f is *optimizee*, ϕ are optimizer parameters, θ are optimizee parameters.

Table of contents

High level view



Is this loss?

- Let $\theta^*(f, \phi)$ be the final optimizee parameters. Define expected loss the following way:

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

- This loss depends only on the end of trajectory. As a result, backpropagation through time is broken.
- How about adding information about trajectory?

No, this is loss

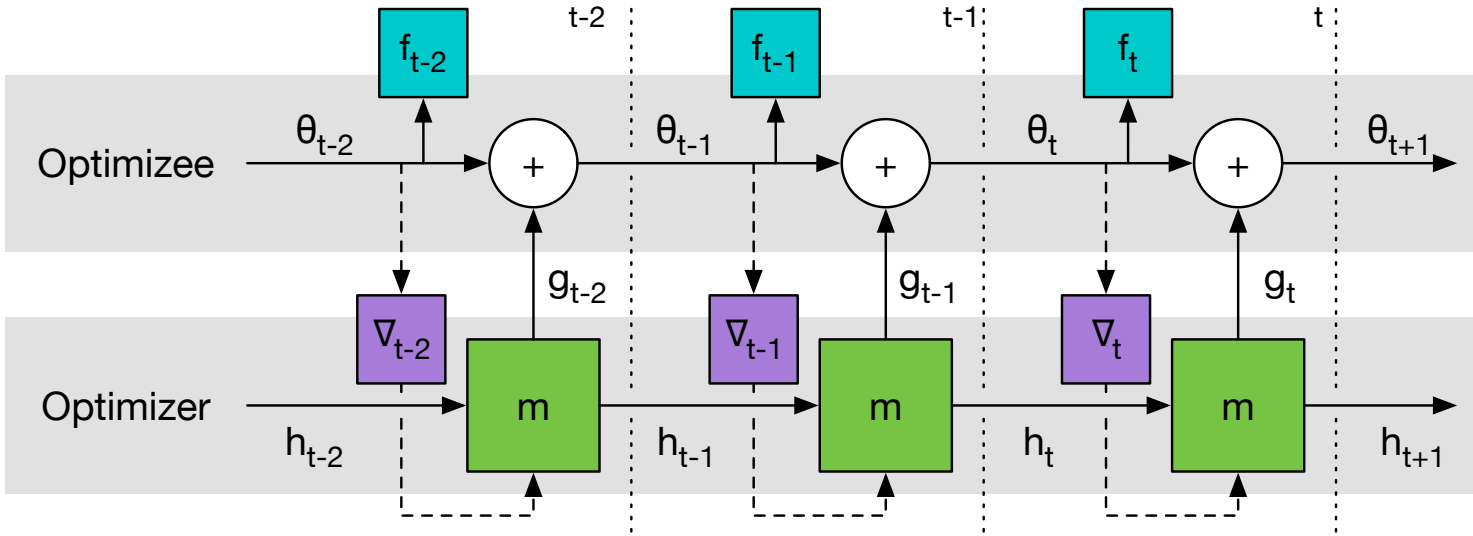
- Let g_t be the output of RNN m with parameters ϕ and state h_t. Then we can use the following loss:

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

  $\nabla_t = \nabla_{\theta_t} f(\theta_t)$, T is horizon and $w_t \in \mathbb{R}_+$, $t \in \{1, 2, \ldots, T\}$ are arbitrary weights.
- It is equal to the previous loss is we say that $w_t = \mathbf{1}_{t=T}$.
- For fixing BPTT, relax the objective: set $w_t > 0$ at intermediate points.

I heard you like gradients

- We'll minimize $\mathcal{L}(\phi)$ using gradient descent on ϕ. Gradient is computed by sampling random function f and used for backpropagating through the following computational graph:



- Gradients flow only along the solid lines: gradients along dashed lines are dropped.
- That helps us to avoid computing second derivatives of f.

Coordinatewise optimizer

- Neural networks have tons of parameters. As a result, using fully connected RNN is troublesome: it will need huge hidden state and another ton of parameters.
- Solution: use optimizer that operates coordinatewise on the parameters of the objective function.
- Different behavior on each coordinate is achieved by using separate activations for each objective function parameter.

LSTM optimizer

- Update rule is implemented using two-layer LSTM with forget-gate architecture. All LSTMs have shared parameters, but separate hidden states.

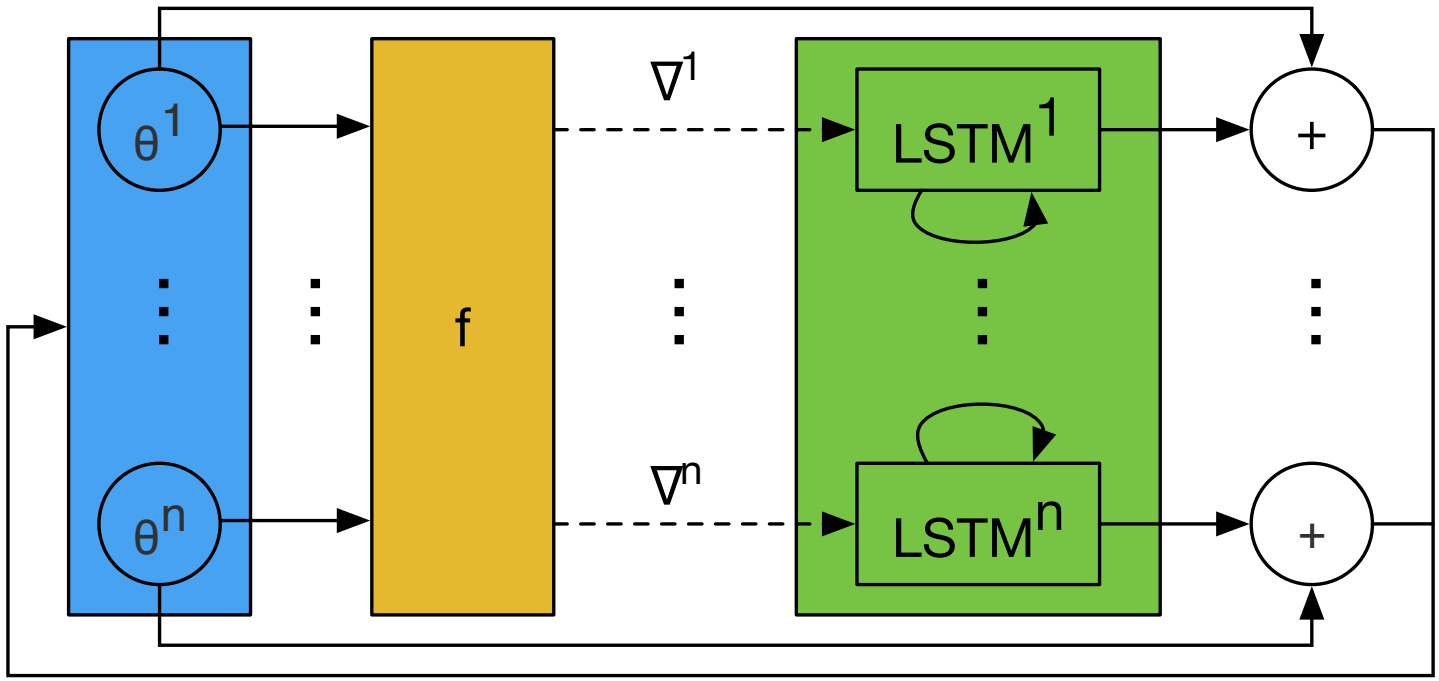


Table of contents

Preprocessing and models

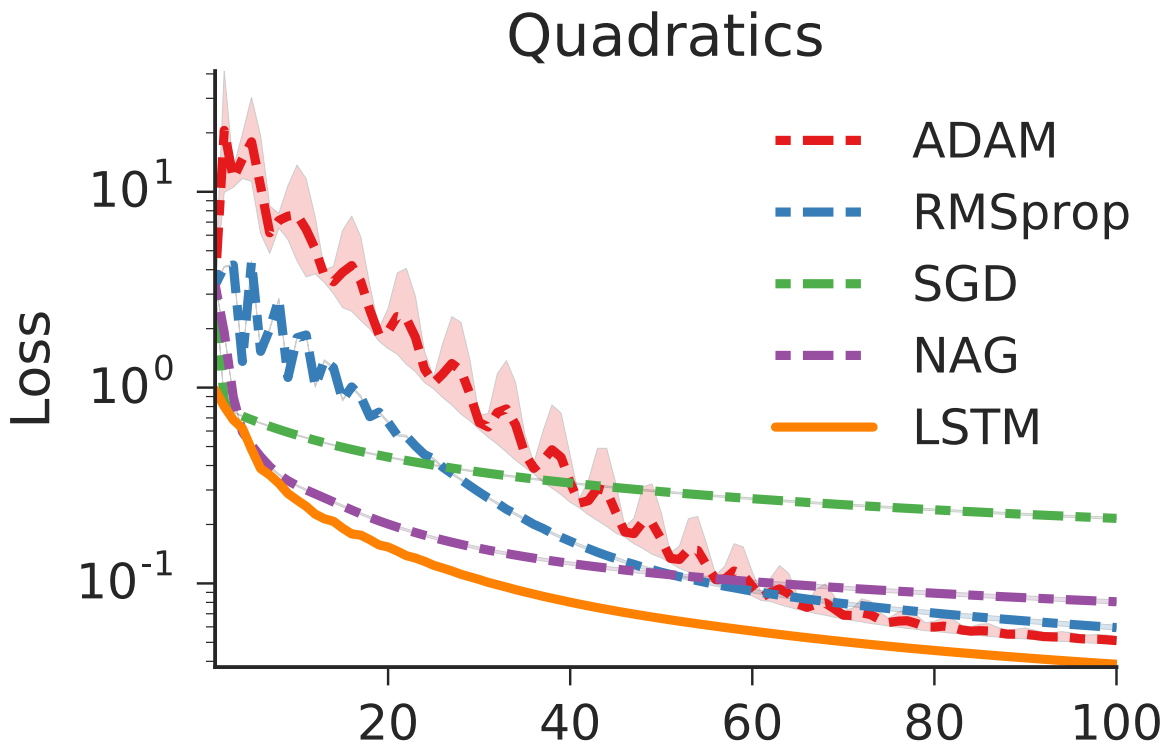
- Gradient preprocessing:

$$\nabla \to \begin{cases} \left(\frac{\ln(|\nabla|)}{p}, \operatorname{sgn}(\nabla)\right) & \text{if } |\nabla| \geqslant e^{-p} \\ (-1, e^{p}\nabla) & \text{otherwise} \end{cases}$$

- In all experiments $w_t = 1$ for all $t \in \{1, 2, \ldots, T\}$ and $p = 10$.
- The trained optimizers use two-layer LSTMs with 20 hidden units in each layer, trained with early stopping using ADAM.

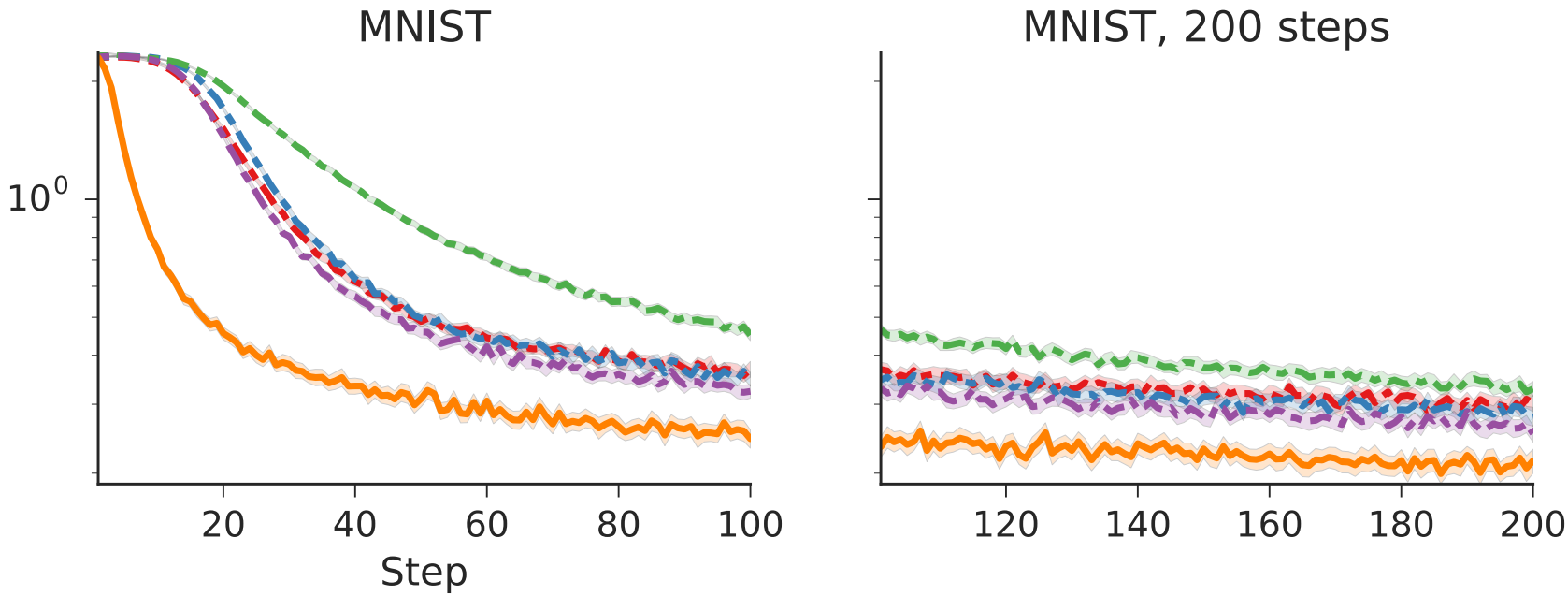
Quadratic functions

We minimize the quadratic function $f(\theta) = \|W\theta - y\|_2^2$, where $W \in \mathbb{R}^{10\times 10}$, θ, $y \in \mathbb{R}^{10}$ and W, y are IID Gaussian.



MNIST

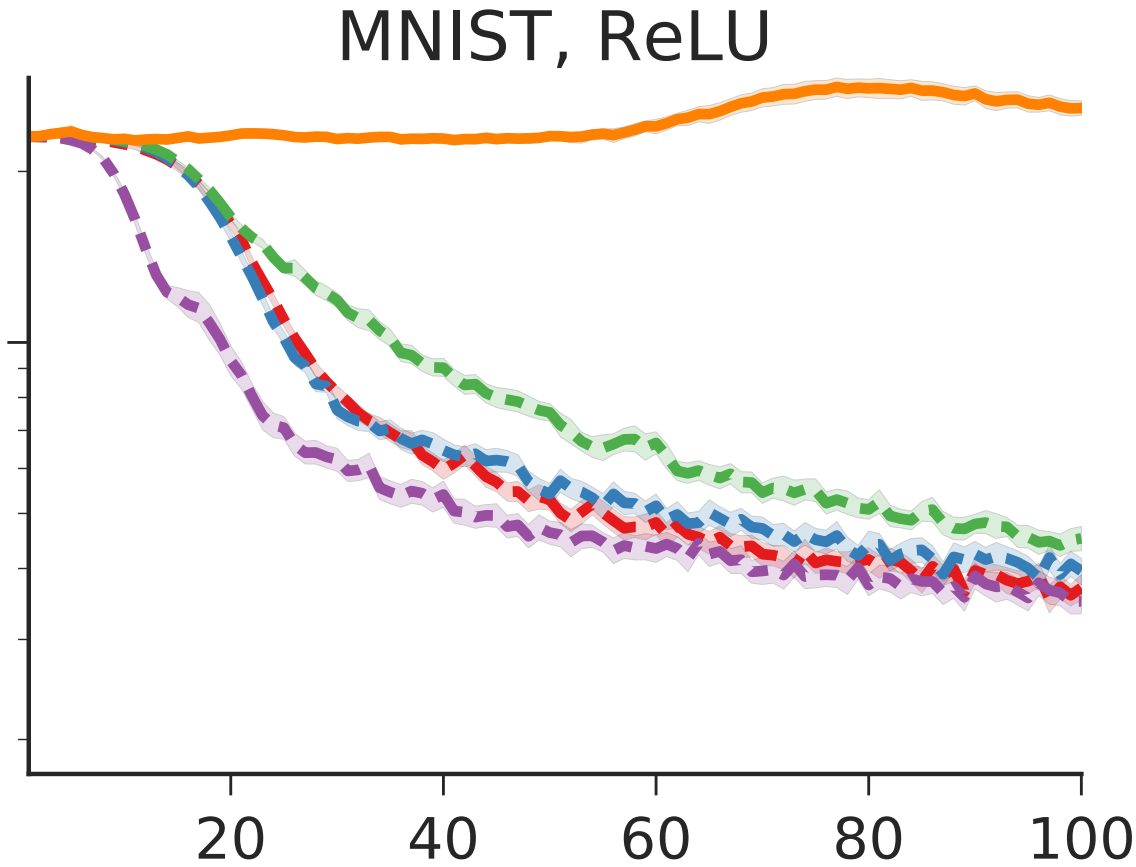
You know the drill. Classifier is a single hidden layer FCN with 20 hidden units network with sigmoid activation. And it works!



Changing number of hidden units or adding layers doesn't break anything.

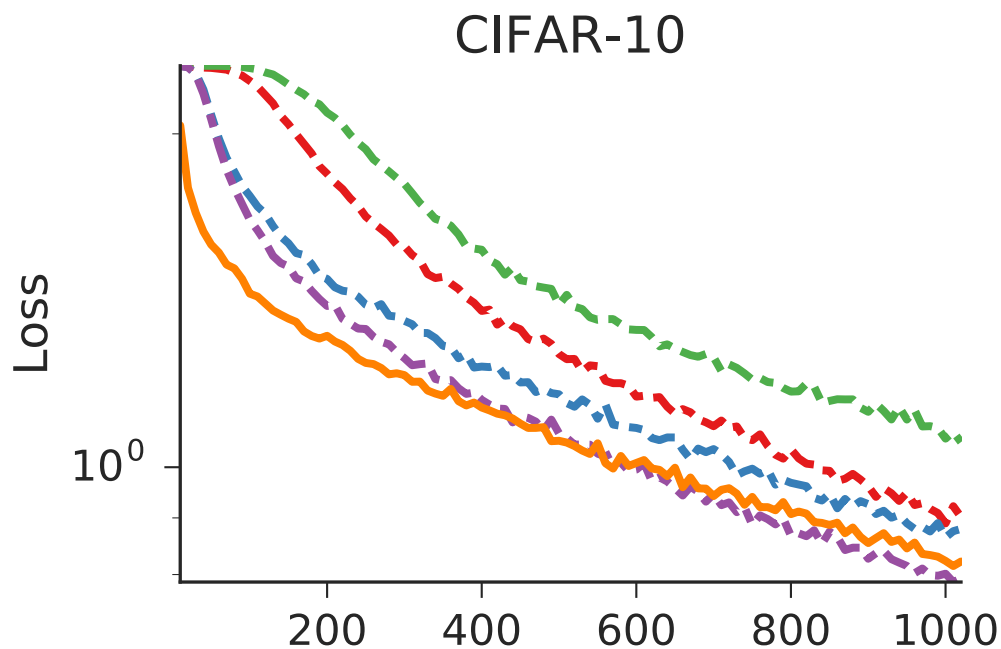
How to shoot yourself in foot in one easy step

Let's change the activation function to ReLU. Will it still work?
Nope, it won't converge.



CIFAR-10

- Next step: try to optimize CNN with ReLU activations and batch normalization.
- Naive approach is not sufficient. Solution: use two LSTMs — one for convolutional layers and one for fully-connected layer.
- And it works:



Conclusion

Pros:

- New approach for optimizing neural networks.
- Shows great results that are comparable to state-of-the-art first-order optimization methods (sometimes).

Cons:

- Not really scalable.

Table of contents

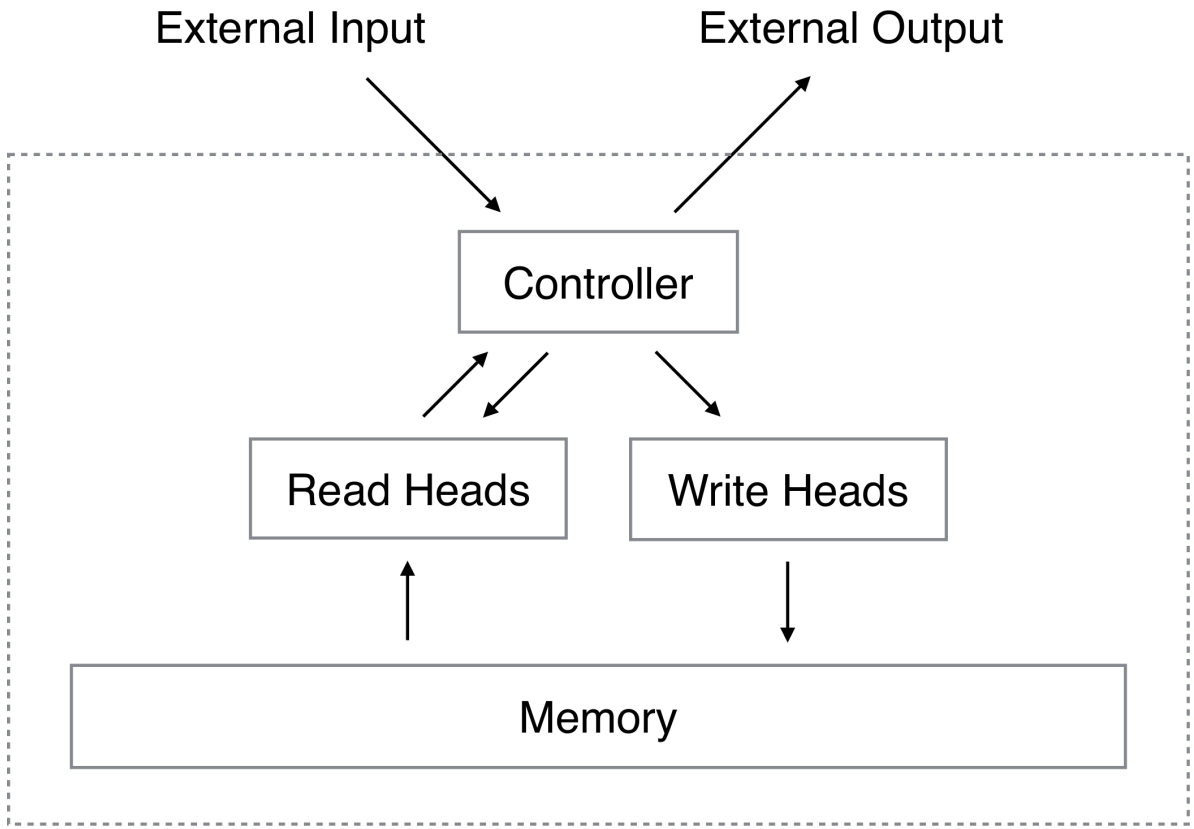
Global averaging cells

- Let's take a subset of LSTM cells in each layer.
- Next, average their outgoing activations at each step across all coordinates.
- Voilà — you get global averaging cells (GAC for short). Such architecture is denoted as LSTM+GAC.
- They are sufficient to allow networks to implement L_2 gradient clipping:

$$\nabla \to \begin{cases} \nabla, & \|\nabla\|_2 \leq c \\ \frac{\nabla}{\|\nabla\|_2}, & \text{otherwise} \end{cases}$$

Little reminder: NTM

- Neural Turing Machines (NTM for short) is a memory augmented neural network, where the interactions with the external memory (address, read, write) are done using differentiable transformations.



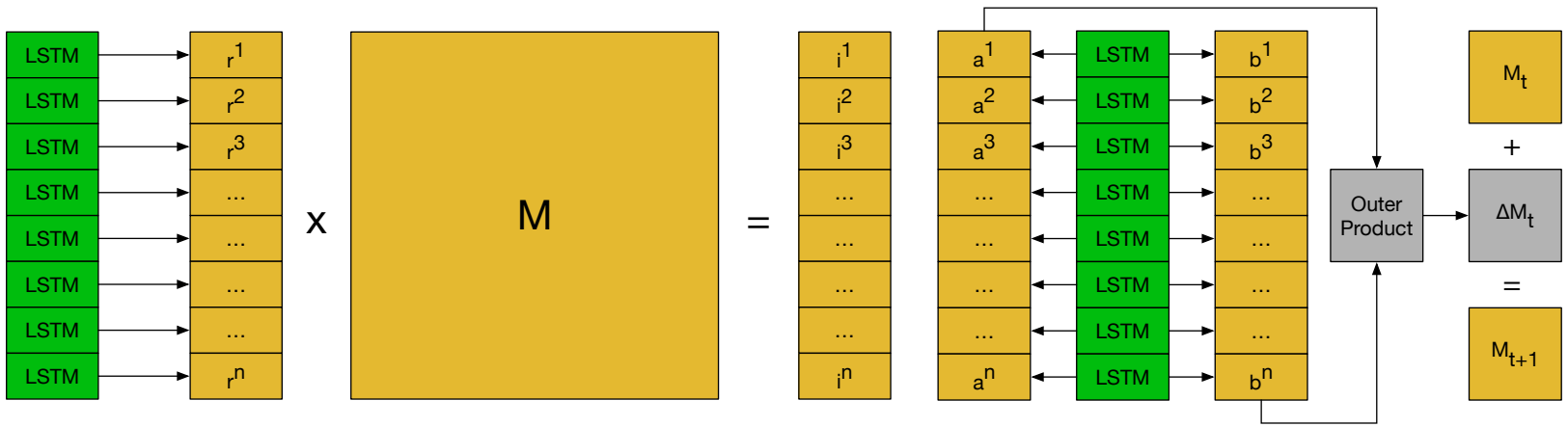
NTM-BFGS optimizer: idea

- Skeletonized algorithm:

$$
\begin{aligned}
g_t &= \text{read}(M_t, \theta_t) \\
\theta_{t+1} &= \theta_t + g_t \\
M_{t+1} &= \text{write}(M_t, \theta_t, g_t)
\end{aligned}
$$

- In quasi-Newton methods M_t is an approximation to inverse Hessian and, for example, $\text{read}(M_t, \theta_t) = -M_t \nabla f(\theta_t)$. Write operation depends on the algorithm.
- Let's save the idea, but discard operations' particular form and learn something better.

Caveat emptor: architecture of NTM-BFGS

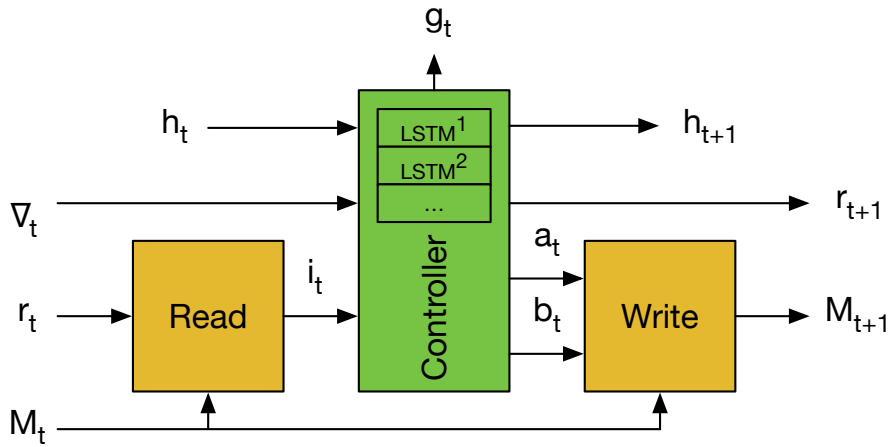


(a) read operation.

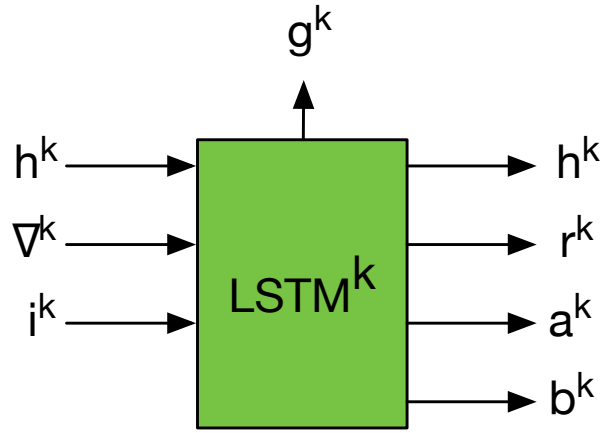
(b) write operation.

Figure: NTM-BFGS operations.

Architecture of NTM-BFGS, part two



(a) Interaction between controller and external memory.



(b) A single LSTM for the k-th coordinate.

Figure: General idea.