Learning to learn by gradient descent by gradient descent

Marcin Andrychowicz , Misha Denil , Sergio Gómez Colmenarejo , Matthew W. Hoffman , David Pfau , Tom Schau , Brendan Shillingford , Nando de Freitas

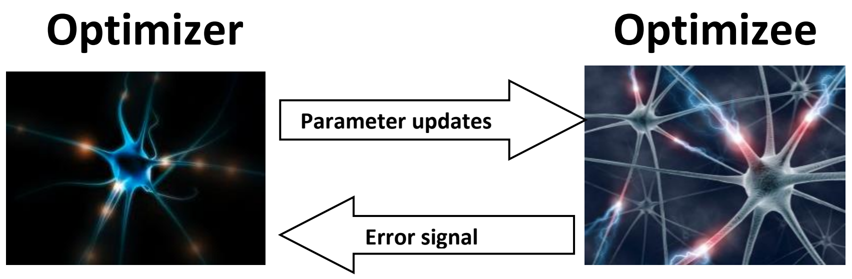
1. Introduction

The performance of vanilla gradient descent is hampered by the fact that it only makes use of gradients and ignores second-order information. There are many optimization methods which exploit structure in their problems of interest at the expense of potentially poor performance on problems outside of that scope. Moreover the *No Free Lunch Theorems for Optimization* shows that in the setting of combinatorial optimization, no algorithm is able to do better than a random strategy in expectation. This suggests that specialization to a subclass of problems is in fact the only way that improved performance can be achieved in general. This work aims to leverage this generalization power, but also to lift it from simple supervised learning to the more general setting of optimization.

This paper proposes to replace hand-designed update rules with a learned update rule, which is called as optimizer g, specified by its own set of parameters ϕ. This results in updates to the optimizee f.

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

It explicitly models the update rule g using a recurrent neural network (RNN) which maintains its own state and hence dynamically updates as a function of its iterates.



In simple words they are 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.

2. Learning to learn with RNN

The loss function or the loss of the optimizer described in the paper is the sum of the losses of the optimizee as it learns. The paper includes some notion of weighing but gives a weight of 1 to everything.

$$\mathscr{L}(\phi) = \mathbb{E}_f\left[\sum_{t=1}^{T} w_t f(\theta_2)\right]$$

where

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

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

And

$$\nabla_t = \nabla_\theta f(\theta_t)$$

The w_t are arbitrary weights for each time step. If only the last w_t is 1 and the rest is 0, we are optimizing for the best **final** result with our optimizee. This seems reasonable, but it makes it much harder to train. Instead they use $w_t = 1$ for all t.

f is the *optimizee* function, and θ_t is its parameters at time *t*. *m* is the *optimizer* function or RNN, ϕ is its parameters. h_t is its state at time *t* and g_t 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.

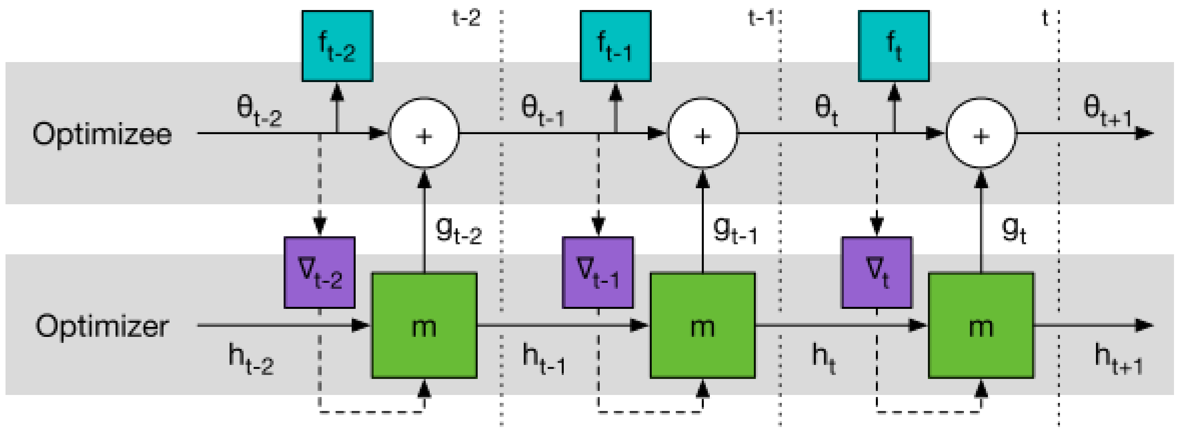


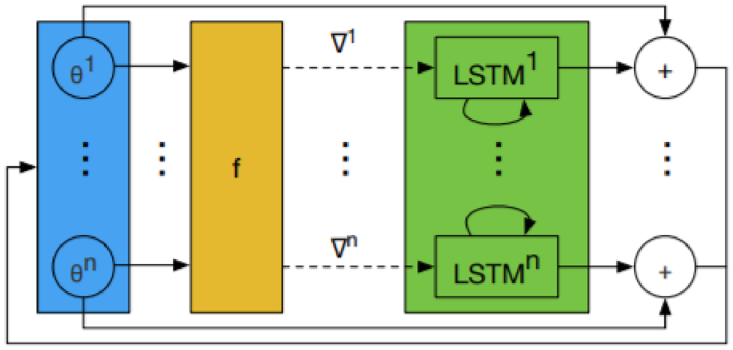
Figure 2: Computational graph used for computing the gradient of the optimizer.

The loss of the optimizer neural net is simply the average training loss of the optimizee as it is trained by the optimizer. The optimizer takes in the gradient of the current coordinate of the optimizee as well as its previous state, and outputs a suggested update that will reduce the optimizee's loss as fast as possible.

2.1 Coordinatewise LSTM optimizer

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 they 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.

They implement the update rule for each coordinate using a two-layer LongShortTermMemory (LSTM) network, using the now-standard forget gate architecture. The network takes as input the optimizee gradient for a single coordinate as well as the previous hidden state and outputs the update for the corresponding optimizee parameter.



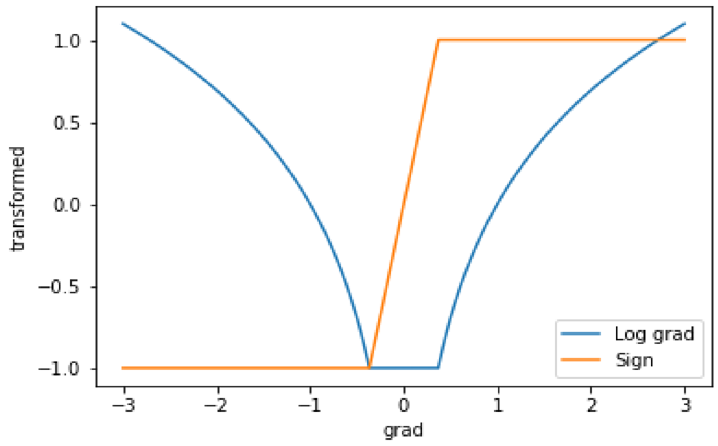
2.2 Gradient Preprocessing

One potential challenge in training optimizers is that different input coordinates (i.e. the gradients w.r.t. different optimizee parameters) can have very different magnitudes. This is indeed the case e.g. when the optimizee is a neural network and different parameters correspond to weights in different layers. This can make training an optimizer difficult, because neural networks naturally disregard small variations in input signals and concentrate on bigger input values. For this preprocess the optimizer's inputs. One solution would be to give the optimizer (log(|∇|), sgn(∇)) as an input, where ∇ is the gradient in the current time step. This has a problem that log(|∇|) diverges for ∇ → 0. Therefore, using the following preprocessing formula

$$\nabla^k \rightarrow \begin{cases} \left(\frac{\log(|\nabla|)}{p}, \operatorname{sgn}(\nabla)\right) & \text{if } |\nabla| \geq e^{-p} \\ (-1, e^p \nabla) & \text{otherwise} \end{cases}$$

otherwise where p > 0 is a parameter controlling how small gradients are disregarded (we use p = 10 in all our experiments).

The function is plotted below (with p = 1), as we can see, the sign component takes over when the log grad component is clipped, and vice-versa.



3. Experiments

3.1 Quadratic function

The optimizer is supposed to find a 10-element vector called θ that, when multiplied by a 10x10 matrix called W, is as close as possible to a 10-element vector called y. Both y and W are generated randomly. The error is simply the squared error. Each epoch is made up of trying to optimize a new random function for 100 steps, and doing an update of the optimizer at every 20 steps.

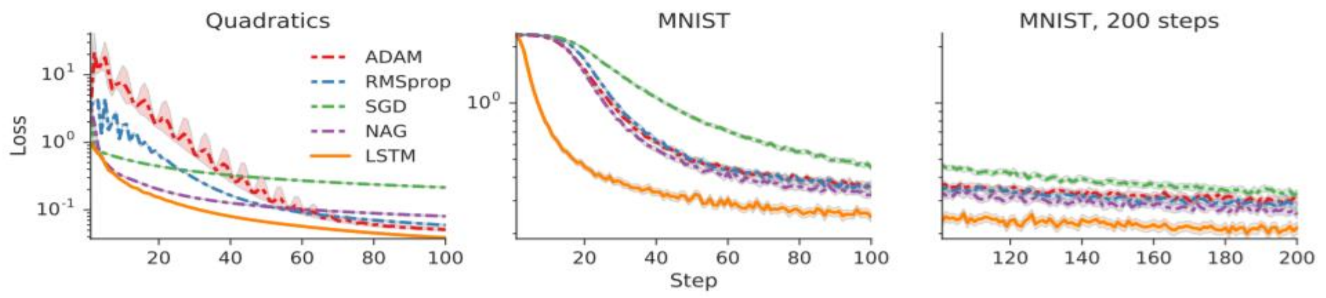


Figure 4: Comparisons between learned and hand-crafted optimizers performance. Learned optimizers are shown with solid lines and hand-crafted optimizers are shown with dashed lines. Units for the y axis in the MNIST plots are logits. **Left:** Performance of different optimizers on randomly sampled 10-dimensional 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).

3.2 Training a small network on MNIST

This test whether trainable optimizers can learn to optimize a small neural network on MNIST, and also explore how the trained optimizers generalize to functions beyond those they were trained on. To this end, the optimizer is trained to optimize a base network and explore a series of modifications to the network architecture and training procedure at test time.

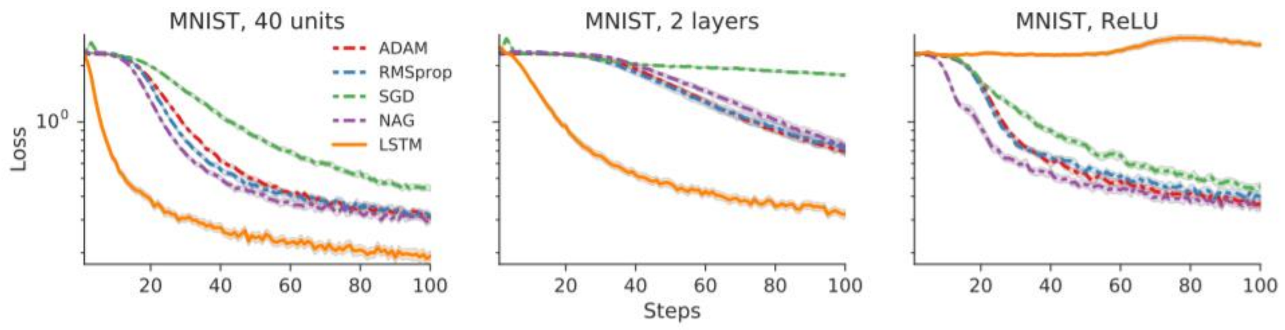


Figure 5: Comparisons between learned and hand-crafted optimizers performance. Units for the y axis are logits. **Left:** Generalization to the different number of hidden units (40 instead of 20). **Center:** Generalization to the different number of hidden layers (2 instead of 1). This optimization 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.

3.3 Training a convolution network on CIFAR-10

This is to test the performance of the trained neural optimizers on optimizing classification performance for the CIFAR-10 dataset. In these experiments they used a model with both convolutional and feed-forward layers. In particular, the model used for these experiments includes three convolutional layers with max pooling followed by a fully-connected layer with 32 hidden units; all non-linearities were ReLU activations with batch normalization.
In all these examples it can be seen that the LSTM optimizer learns much more quickly than the baseline optimizers, with significant boosts in performance for the CIFAR-5 and especially CIFAR-2 datsets.

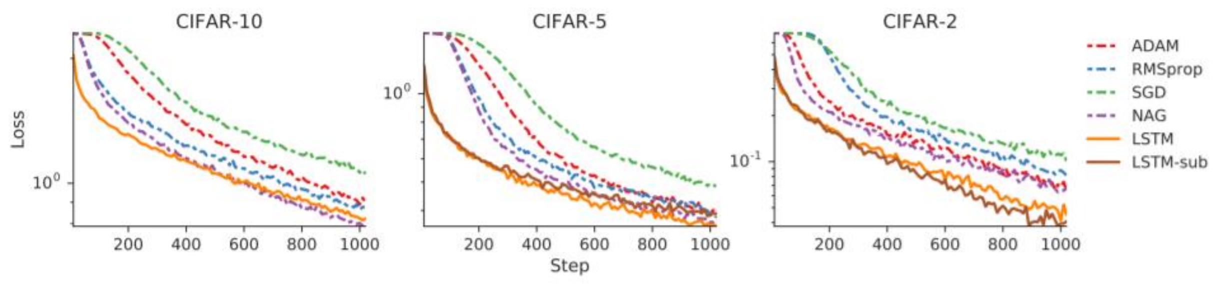


Figure 7: Optimization performance on the CIFAR-10 dataset and subsets. Shown on the left is the LSTM optimizer versus various baselines trained on CIFAR-10 and tested on a held-out test set. The two plots on the right are the performance of these optimizers on subsets of the CIFAR labels. The additional optimizer *LSTM-sub* has been trained only on the heldout labels and is hence transferring to a completely novel dataset.

3.4 Neural Art

The recent work on artistic style transfer using convolutional networks, or Neural Art, gives a natural test bed for the method, since each content and style image pair gives rise to a different optimization problem. Each Neural Art problem starts from a content image, c, and a style image, s, and is given by

$$f(\theta) = \alpha L_{content}(c, \theta) + \beta L_{style}(s, \theta) + \gamma L_{reg}(\theta)$$

The minimizer of f is the styled image. The first two terms try to match the content and style of the styled image to that of their first argument, and the third term is a regularizer that encourages smoothness in the styled image.

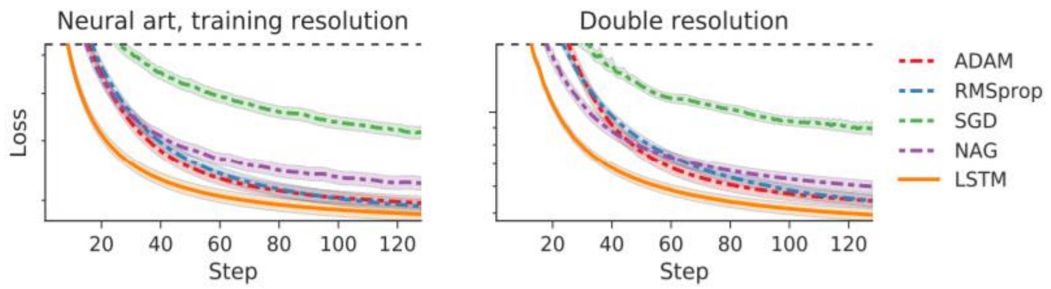


Figure 8: Optimization curves for Neural Art. Content images come from the test set, which was not used during the LSTM optimizer training. Note: the y-axis is in log scale and we zoom in on the interesting portion of this plot. **Left:** Applying the training style at the training resolution. **Right:** Applying the test style at double the training resolution.

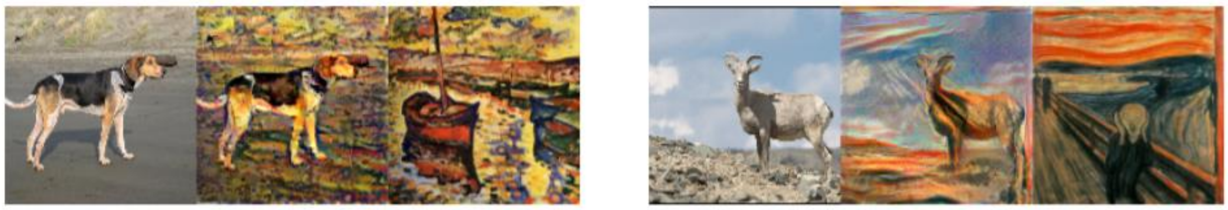


Figure 9: Examples of images styled using the LSTM optimizer. Each triple consists of the content image (left), style (right) and image generated by the LSTM optimizer (center). **Left:** The result of applying the training style at the training resolution to a test image. **Right:** The result of applying a new style to a test image at double the resolution on which the optimizer was trained.

4. Conclusion

The paper shows how to cast the design of optimization algorithms as a learning problem. The experiments have confirmed that learned neural optimizers compare favorably against state-of-the-art optimization methods used in deep learning. It was witnessed as a remarkable degree of transfer, with for example the LSTM optimizer trained on 12,288 parameter neural art tasks being able to generalize to tasks with 49,152 parameters, different styles, and different content images all at the same time. Observed similar impressive results when transferring to different architectures in the MNIST task. Results on the CIFAR image labeling task show that the LSTM optimizers outperform hand-engineered optimizers when transferring to datasets drawn from the same data distributio