Learning to learn by gradient descent

by gradient descent

Marcin Andrychowicz, Misha Denil, Sergio Gomez, Matthew W. Hoffman, David Pfau, Tom Schaul, Brendan Shillingford, Nando de Freitas

CRITICAL REVIEW

MAHENDRA KUMAR (15114043)
NITIN GAURAV SINGH (15118049)

18/04/2018

INTRODUCTION

The authors propose a method for learning update functions in gradient-based optimizers. By jointly training an optimizer and optimizes, parameters of an LSTM-based DNN can be learned that accumulate information from multiple gradients, similar to momentum. Once trained, an optimizer can be reused to optimize similar tasks, and can partially generalize to new network architectures. While no theoretical properties of the resulting algorithms are known, several small empirical experiments demonstrate that the resulting optimizers are competitive with state-of-the-art alternatives.

SUMMARY

Thinking in terms of functions like this is a bridge back to the familiar (for me at least). We have function composition. For example, given a function *f* mapping images to feature representations, and a function *g* acting as a classifier mapping image feature representations to objects, we can build a systems that classifies objects in images with $g \bigcirc f$.
Each function in the system model could be learned or just implemented directly with some algorithm. For example, feature mappings (or encodings) were traditionally implemented by hand, but increasingly are learned...

"The move from hand-designed features to learned features in machine learning has been wildly successful."

Part of the art seems to be to define the overall model in such a way that no individual function needs to do too much (avoiding too big a gap between the inputs and the target output) so that learning becomes more efficient / tractable, and we can take advantage of different techniques for each function as appropriate. In the above example, we composed one learned function for creating good representations, and another function for identifying objects from those representations.

We can have higher-order functions that combine existing (learned or otherwise) functions, and of course that means we can also use *combinators*.

And what do we find when we look at the components of a 'function learner' (machine learning system)? More functions!

"Frequently, tasks in machine learning can be expressed as the problem of optimising an objective function f(θ) defined over some domain $\theta \in \Theta$."

The *optimizer* function maps from $f\ \theta$ to $argmin_{\theta \in \Theta} f\ \theta$. The standard approach is to use some form of gradient descent (e.g., SGD – stochastic gradient descent). A classic paper in optimisation is 'No Free Lunch Theorems for Optimization' which tells us that no general-purpose optimisation algorithm can dominate all others. So to get the best performance, we need to match our optimisation technique to the characteristics of the problem at hand:

"... specialisation to a subclass of problems is in fact the only way that improved performance can be achieved in general."

Thus there has been a lot of research in defining update rules tailored to different classes of problems – within deep learning these include for example *momentum*, *Rprop*, *Adagrad*, *RMSprop*, and *ADAM*.

But what if instead of hand designing an optimising algorithm (function) we *learn* it instead? That way, by training on the class of problems we're interested in solving, we can learn an optimum optimiser for the class!

"The goal of this work is to develop a procedure for constructing a learning algorithm which performs well on a particular class of optimisation problems. Casting algorithm design as a learning problem allows us to specify the class of problems we are interested in through example problem instances. This is in contrast to the ordinary approach of characterising properties of interesting problems analytically and using these analytical insights to design learning algorithms by hand."

If learned representations end up performing better than hand-designed ones, can learned optimisers end up performing better than hand-designed ones too? The answer turns out to be yes!

"Our experiments have confirmed that learned neural optimizers compare favorably against state-of-the-art optimization methods used in deep learning."

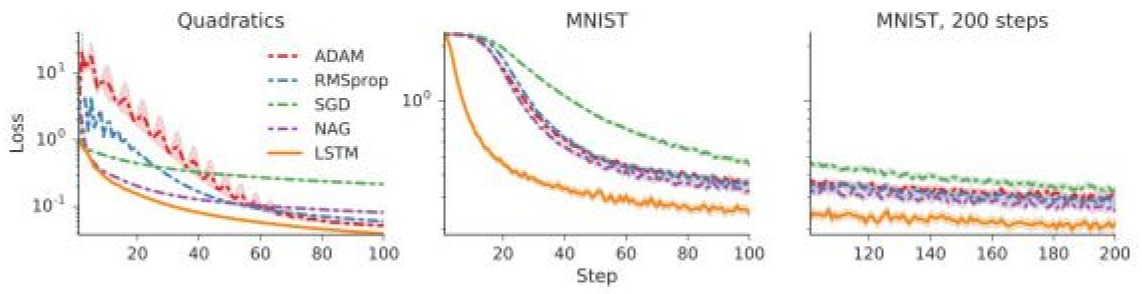

Figure 4: Comparisons between learned and hand-crafted optimizers performance. Learned optimizers are shown with solid lines and hand-crafted optimizers are shown with dashed lines. Units for the y axis in the MNIST plots are logits. **Left:** Performance of different optimizers on randomly sampled 10-dimensional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

In fact not only do these learned optimisers perform very well, but they also provide an interesting way to transfer learning across problems sets. Traditionally *transfer learning* is a hard problem studied in its own right. But in this context, because we're learning how to learn, straightforward *generalization* (the key property of ML that lets us learn on a training set and then perform well on previously unseen examples) provides for transfer learning!!

"We witnessed 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 49,512 parameters, different styles, and different content images all at the same time. We observed similar impressive results when transferring to different architectures in the MNIST task."

Learning how to learn

Thinking functionally, here's my mental model of what's going on… In the beginning, you might have hand-coded a classifier function, *c*, which maps from some *Input* to a *Class*:

" c :: Input -> Class"

With machine learning, we figured out for certain types of functions it's better to learn an implementation than try and code it by hand. An optimisation function *f* takes some *TrainingData* and an existing classifier function, and returns an updated classifier function:

"type Classifier = (Input -> Class)
f :: TrainingData -> Classifier -> Classifier"

What we're doing now is saying, "well, if we can learn a function, why don't we learn *f* itself?"

"type Optimiser = (TrainingData -> Classifier -> Classifier)
g :: TrainingData -> Optimiser -> Optimiser"

Let ϕ be the (to be learned) update rule for our (optimiser) optimiser. We need to evaluate how effective *g* is over a number of iterations, and for this reason *g* is modelled using a recurrent neural network (LSTM). The state of this network at time *t* is represented by h_t.

Suppose we are training *g* to optimise an optimisation function *f*. Let $g(\phi)$ result in a learned set of parameters for $f\ \theta$, The loss function for training $g(\phi)$ uses as *its* expected loss the expected loss of *f* as trained by $g(\phi)$.

"We can minimise the value of $L(\phi)$ using gradient descent on ϕ."

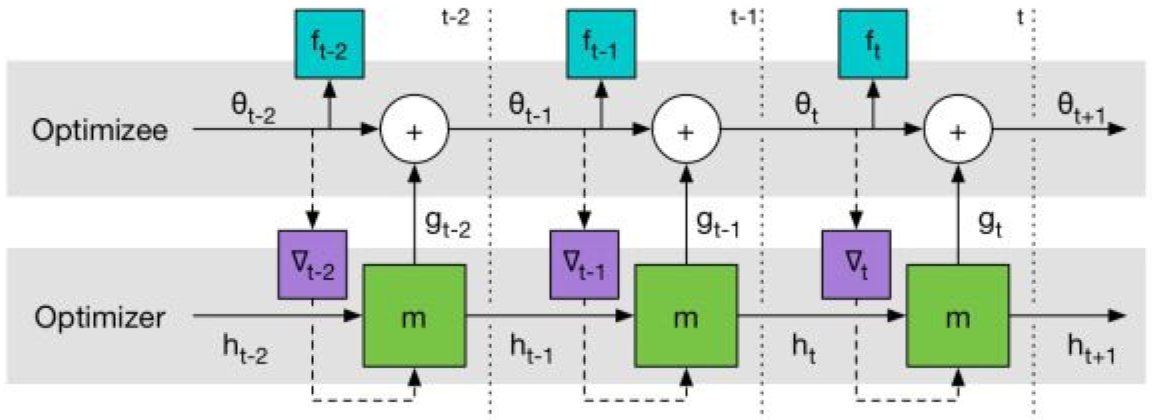

Figure 2: Computational graph used for computing the gradient of the optimizer.

To scale to tens of thousands of parameters or more, the optimiser network *m* operators coordinatewise on the parameters of the objective function, similar to update rules like RMSProp and ADAM. The update rule for each coordinate is implemented using a 2-layer LSTM network using a 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 optimise parameter. We refer to this architecture as an LSTM optimiser."

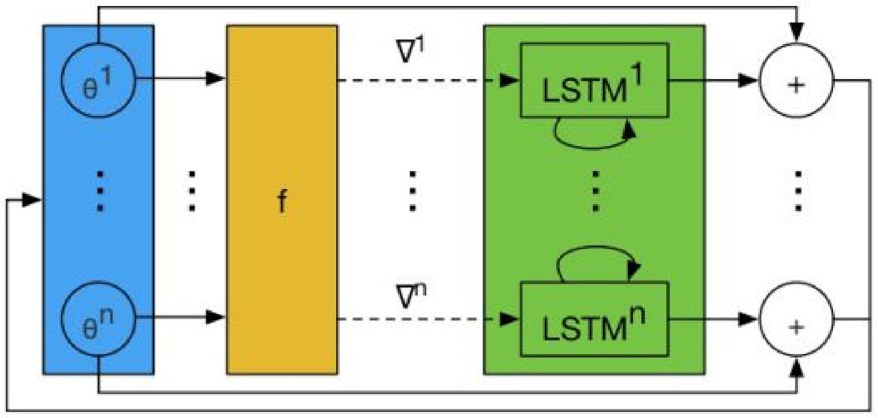

Figure 3: One step of an LSTM optimizer. All LSTMs have shared parameters, but separate hidden states.

Learned learners in action

"We compare our trained optimizers with standard optimisers used in Deep Learning: SGD, RMSprop, ADAM, and Nesterov's accelerated gradient (NAG). For each of these optimizers and each problem we tuned the learning rate, and report results with the rate that gives the best final error for each problem."

Optimisers were trained for 10-dimensional quadratic functions, for optimising a small neural network on MNIST, and on the CIFAR-10 dataset, and on learning optimisers for neural art (see e.g. Texture Networks).

Here's a closer look at the performance of the trained LSTM optimiser on the Neural Art task vs standard optimisers:

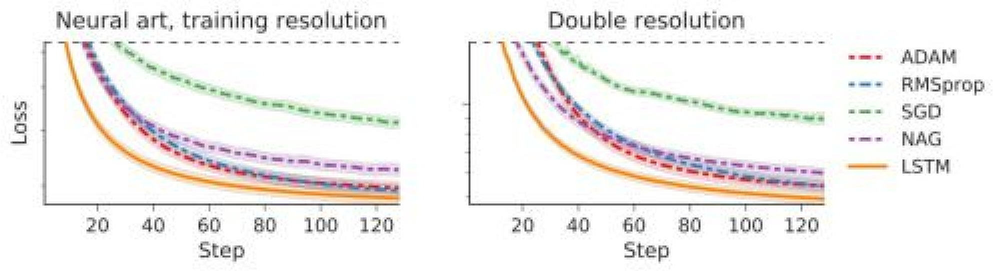

Figure 8: Optimization curves for Neural Art. Content images come from the test set, which was not used during the LSTM optimizer training. Note: the y-axis is in log scale and we zoom in on the interesting portion of this plot. **Left:** Applying the training style at the training resolution. **Right:**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

And because they're pretty… here are some images styled by the LSTM optimiser!

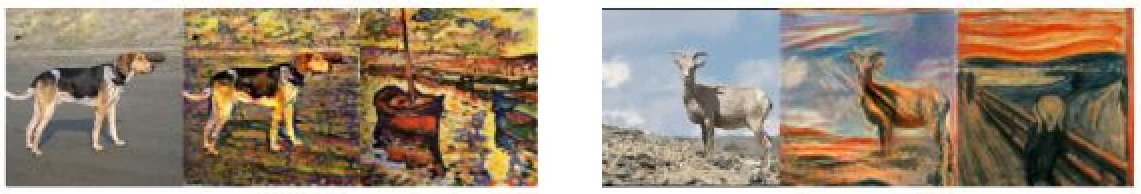

Figure 9: Examples of images styled using the LSTM optimizer. Each triple consists of the content image (left), style (right) and image generated by the LSTM optimizer (center). **Left:** The result of applying the training style at the training resolution to a test image. **Right:** The result of applying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

PROS

Some of the positives of this paper are:

1. This is a really novel optimizing technique that replaces hand-crafted parameters update rules with learnable update rule. The learning curve achieved much better result than traditional methods. It's a well written paper enjoyable to read and very easy to follow.

2. It's creative, timely, interesting, and could have significant impact. The idea of "learning algorithms" is an important next step in deep learning. This seems like a reasonable step in that direction.

3. The experimental results seemed fine. While it was a bit surprising to see that the resulting algorithms were better than state-of-the-art alternatives, it was more surprising to see that they weren't better by all that much. This suggests that our current algorithms are pretty good; a testament to the hard work done by the optimization community.

CONS

There are a few questions hopefully the authors could have addressed.

1. What is the time cost of running LSTM optimizer? Apparently the LSTM optimizer is not free lunch, the proposed update rule requires much more complicated computation than SGD (computer a LSTM v.s. just do ax+y). How much slower will the training process be?

2. In the result section, the authors provided the loss value over iterations, what about the final accuracy?

3. The authors provided experiment results on simple tasks such as quadratic functions, MNIST and Neural Art, how is this method generalizable to modern deep networks such as ResNet on large scale datasets? will it be hard to train?

FUTURE SCOPE

From a conceptual point of view, the idea proposed in the paper is very interesting. It is simple to understand, but it opens for many possibilities.

First-order optimization algorithms have two main advantages:

1. They can be applied to almost any (sub)differentiable problem

2. They are very cheap to implement once the gradients are known.

The method appears to be promising particularly for situations in which there is the need for solving the same optimization problem thousands of times over. In this sense, I guess the most interesting applications can come from outside the realm of machine learning.

CONCLUSION

This paper shows 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.

The experiments performed have confirmed that learned neural optimizers compare favorably against state-of-the-art optimization methods used in deep learning.

The authors observed similar impressive results when transferring to different architectures in the MNIST task. The 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 distribution.

Paper source:

https://papers.nips.cc/paper/6461-learning-to-learn-by-gradient-descent-by-gradient-descent.pdf