Introduction

This blog contains a reproducibility report on the paper ‘Learning to learn by gradient descent by gradient descent’ by Marcin Andrychowicz, Misha Denil, Sergio Gómez Colmenarejo, Matthew W. Hoffman, David Pfau, Tom Schaul, Brendan Shillingford, and Nando de Freitas [1].

In this paper, the authors claim to have found a gradient descent algorithm, implemented by a Long Short Term Memory (LSTM) Recurrent Neural Network (RNN), that can generate update rules for optimizers that outperform hand-designed update rules. This claim is supported by various experiments on:

- A class of synthetic 10-dimensional quadratic functions
- A small neural network based on the MNIST database
- Classification performance for the CIFAR-10 dataset
- Neural Style Transfer with ImageNet data

In this blog, we will examine the reproducibility of these results and check their robustness by doing some additional experiments/parameter tweaking.

The code that was used as a base for the reproduction is the repository AdrienLE/learning_by_grad_by_grad_repro, which contains a Notebook in which the quadratic and MNIST experiments of the paper are reproduced.
In addition to running and extending this piece of code, we also propose a number of updates to this repository that have to do with actualisation of imports and libraries used, improvements of the code’s readability, and the addition of a setup guide/README file. These additions can be found in lucilenikkels/learning_by_grad_by_grad_repro.

Part 1 | Theory

Motivation
Before we dive into the peculiarities of the experiments conducted in this research, we want to discuss the underlying motivation and architectures for this project. At the core of this research is the problem of optimizing an objective function. Such an objective function is often some type of loss function over a machine learning model’s output, which we then aim to minimize. The standard approach for minimizing differentiable functions is gradient descent. Researchers have designed a number of enhanced update rules in an attempt to further optimize the optimization process for specific targets or use cases. Examples of this for deep learning are RMSprop and ADAM, which are tailored specifically to high-dimensional non-convex optimization problems. The development, and success, of these techniques has led to a trend in which many optimization methods are designed that are specific to work for a small sub-problem, instead of in a more general case. The authors of ‘learning to learn by gradient descent by gradient descent’ propose that an optimizer that learns how to optimize could be carried over to multiple different use cases by default.

Mathematical background

Let's put this in a mathematical frame. A basic gradient descent update rule can be represented as follows: $\theta_{t+1} = \theta_t - \alpha_t \nabla f(\theta_t)$, where f represents the optimizee function (e.g. a loss function).

A specialization of this update rule, like ADAM, stems from this simple equation, but then has some added complexities that should make it more accurate for advanced deep learning use cases. Specifically, they use adaptive learning rates per parameter, instead of one learning rate for all. We now illustrate the specific mathematical workings of ADAM to provide more background knowledge and context. The reader may skip this technical part as it is not directly of influence to this reproduction.

ADAM stores a decaying average of the gradient as follows: $m_t = \beta_1 m_{t-1} + (1 - \beta_1) g_t$, where g_t is the gradient at timestep t and β_1 is an ADAM-specific parameter that can be finetuned per use case.

The same goes for the squared gradient: $v_t = \beta_2 v_{t-1} + (1 - \beta_2) g_t^2$.

To counteract a bias towards zero, the following is issued: $\widehat{m}_t = \frac{m_t}{1-\beta_1^t}$ and $\widehat{v}_t = \frac{v_t}{1-\beta_2^t}$.

Finally, the ADAM parameter update rule becomes: $\theta_{t+1} = \theta_t - \frac{\alpha}{\sqrt{\widehat{v}_t}+\varepsilon}\widehat{m}_t$.

Having established this commonly accepted baseline approach for in-depth comparison later, we now move on to the representation of the learned optimizer from the paper. This update rule can be represented simply by $\theta_{t+1} = \theta_t - g(\nabla f(\theta_t), \phi)$ where ϕ represents the optimizer's parameters. This representation allows to write the expected loss as follows: $\mathcal{L}(\phi) = \mathbb{E}_f\left[f(\theta^*(f, \phi))\right]$, where we write the parameters of the optimizee f as a function of the optimizer parameters ϕ. Now, we see that we can minimize the value of $\mathcal{L}(\phi)$ by performing gradient descent on ϕ.

Assumptions and limitations

In order to avoid computing second-order derivatives of f, the authors make the assumption that the gradients of the optimizee do not depend on the optimizer parameters. In addition, the authors note that, due to the design's architecture, 'gradients of trajectory prefixes are zero and only the final optimization step provides information for training the optimizer', which renders many computations inefficient. Therefore they relax the objective, such that the weight of a step at time t is larger than 0, i.e. $w_t > 0$ at intermediate points in the trajectory. More specifically, they use $w_t = 1$ for all t.

Finally, the authors address the difficulty of optimizing large magnitudes of parameters. They claim to solve this issue by defining a coordinatewise network architecture that uses a small Long Short Term Memory network for each coordinate.

We will now move on to discuss the experiments and reproduction process of this research.

Part 2 | Reproducibility

Process

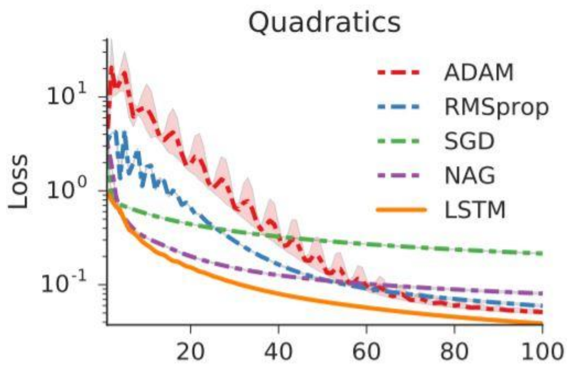
Since code is available for reproducing the work we thought replicating would be relatively easy. Unfortunately, we came across some challenges and we discovered that replicating is not as simple as just pressing the “execute” button in a notebook. The first challenge we had was related to hardware. One of our computers had a GPU that was not CUDA enabled. It was not mentioned in the requirements, this could have been more clear.

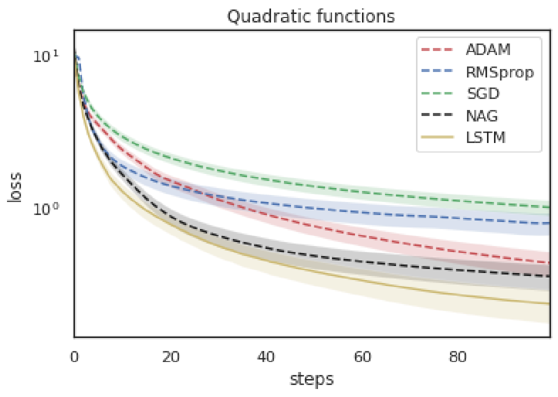
After arranging a laptop that did support CUDA, we faced our second challenge. It appeared that very specific versions of the required packages had to be installed. For example, in seaborn version 0.11.0 the function seaborn.tsplot() is removed, but the code uses this function to plot the loss. Therefore seaborn version 0.9.0 had to be installed. Also, torch.cuda.is_available() always returned False, because the version of cudatoolkit did not match the version of CUDA we installed on our computer. Lastly, we discovered that a virtualvenv had more problems with installing torch and torchvision than a Conda environment. Therefore we advise that people work in a Conda environment.

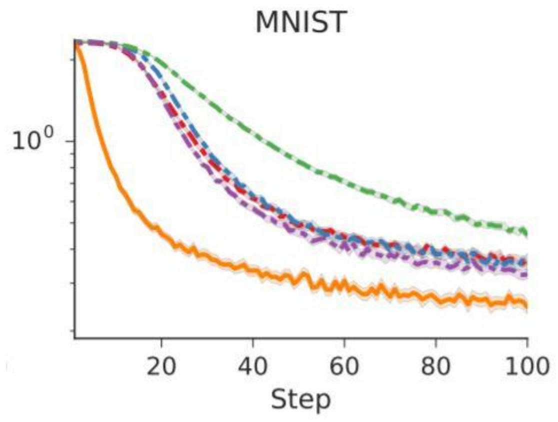
When looking at the code, you see that the code developer also had issues with implementing certain parts of the code. It starts in the theoretical background provided, where the developer says “The thing I am not 100% sure about is what a "coordinate" is supposed to be.” The developer guesses these coordinates, which sounds not very convincing. We thoroughly reread section 2.1 of the paper and have to agree that the explanation is not very clear. It seems that the coordinate pairs are formed by the current objective function of the optimizer and a parameter of the optimizee, a weight or bias. Another issue the developer mentions in his paper are the limitations of Pytorch. Parameters of the optimizee can’t be updated in place, so the convenient Pytorch building blocks like nn.Linear can’t be used since they can’t retain the gradients after an update. The Pytorch function .detach() can’t be used, because it does not guarantee that the gradients will be preserved. Unfortunately, we cannot think of a better package than Pytorch to support building a recurrent neural network, so these are limitations that we have to accept.

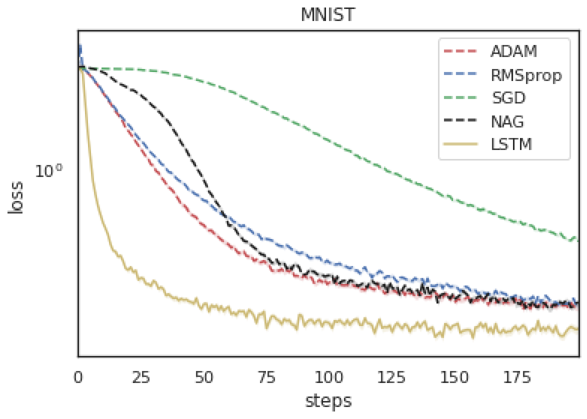
Results

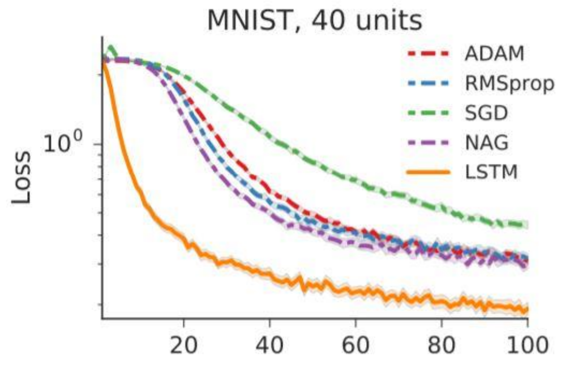
After the setup of the reproduction, the code is complete and we are able to run the code, we can examine the results. An overview of these results can be found in the list of images below, which show the paper’s resulting graphs on the left, and ours on the right.

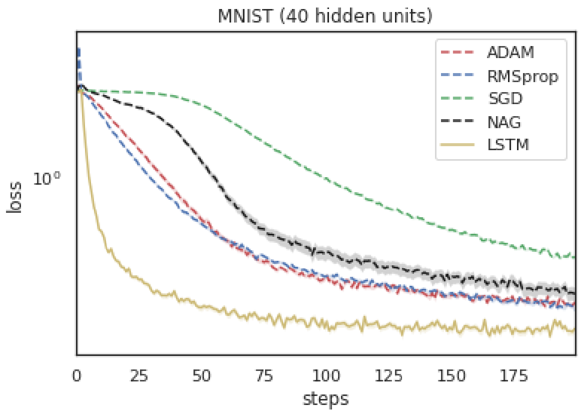

Quadratics
ADAM
RMSprop
SGD
NAG
LSTM
Loss
10^1
10^0
10^{-1}
20
40
60
80
100

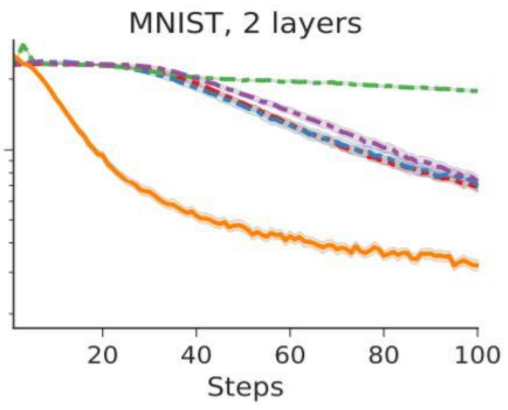

Quadratic functions
ADAM
RMSprop
SGD
NAG
LSTM
loss
10^1
10^0
0
20
40
60
80
steps

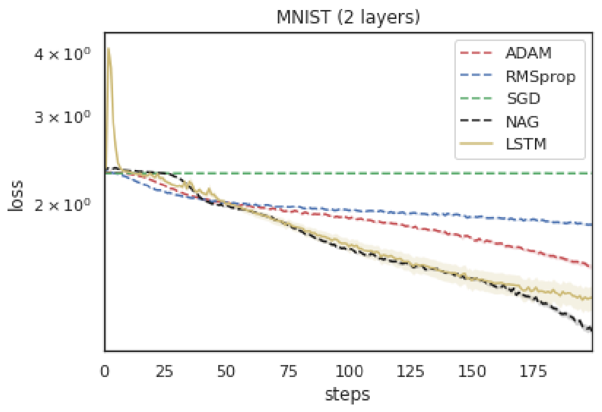

MNIST
10^0
20
40
60
80
100
Step

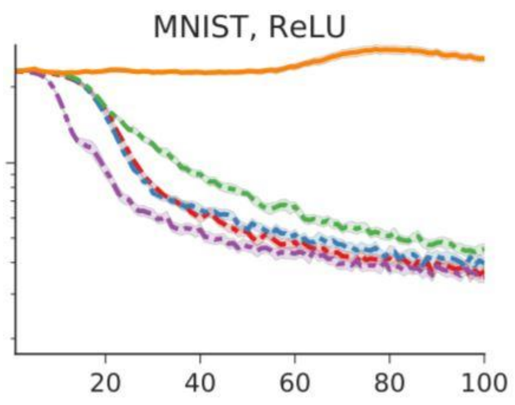

MNIST
ADAM
RMSprop
SGD
NAG
LSTM
loss
10^0
0
25
50
75
100
125
150
175
steps

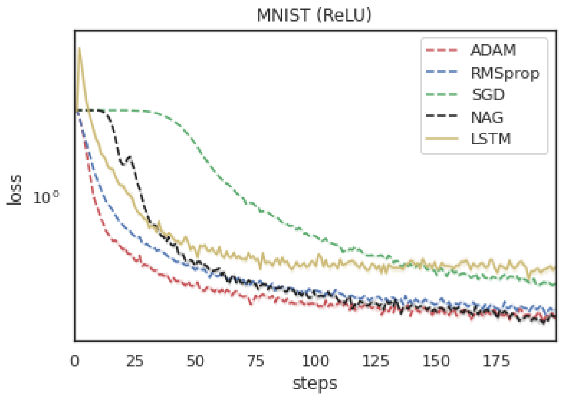

MNIST, 40 units
ADAM
RMSprop
SGD
NAG
LSTM
Loss
10^0
20
40
60
80
100


MNIST (40 hidden units)
ADAM
RMSprop
SGD
NAG
LSTM
loss
10^0
0
25
50
75
100
125
150
175
steps


MNIST, 2 layers
20
40
60
80
100
Steps


MNIST (2 layers)
ADAM
RMSprop
SGD
NAG
LSTM
loss
4×10^0
3×10^0
2×10^0
0
25
50
75
100
125
150
175
steps





In general, it is clear that our results are very similar to the ones found in the paper, so that is certainly a positive outcome. Despite the issues mentioned in the previous section, this means that we can conclude that the experiments in the paper are indeed reproducible and, based on these experiments, the authors have grounded reasons for their claims; using an LSTM looks promising for training an optimizer.

There are two cases in which our results seem a bit further off from the results found in the original paper. The first is the case where we use a ReLU activation function instead of the (standard) sigmoid. Surprisingly, our trained optimizer performs better than the one in the paper. A possible explanation is simply that the code used for this reproduction is not the same as the code that was originally used in the paper. The authors have not made their work public, nor have they been very precise in their implementation method and libraries used, so this discrepancy could be the results of a minor difference in implementation.

Secondly, the results in the paper for applying the learned optimizer to a 2-layer network are a lot better than the ones that we found. This could point to a weakness of the LSTM, namely that it might not generalize well to larger problems. Since the results in the paper do not reflect this bad performance, this issue is not addressed. However, the paper also doesn't include examples/proof of the LSTM performing well on larger networks.

We found this to be good reasons for further investigation into the performance of the LSTM optimizer on large problems. We will describe this investigation in the next section.

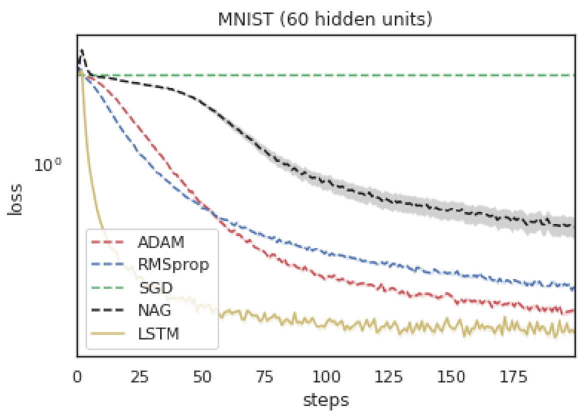
Part 3 | Parameterization

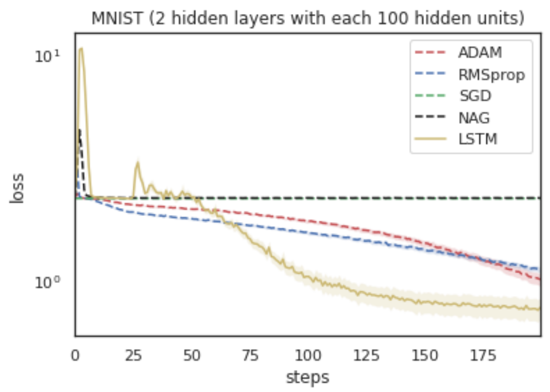
Extending the network

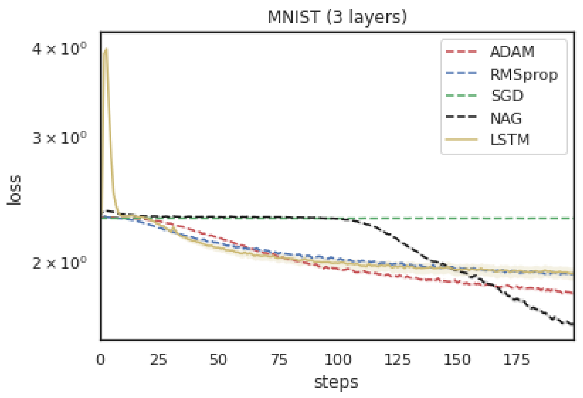
Given the underperformance of our LSTM on the 2-layer network, we suspect that the learned optimizer proposed by the authors does not generalize well to large networks. To verify this claim, we extend the notebook from *AdrienLE/learning_by_grad_by_grad_repro* with some additional test cases. The combinations we tested are the following:

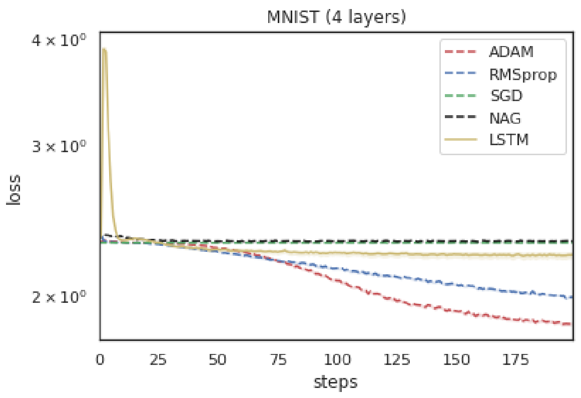
Layers	1	2	3	4	5
Hidden units per layer	60	100	20	20	20

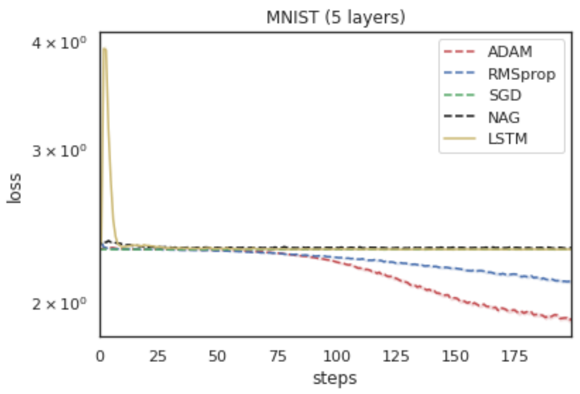
Below we list the result of each of these experiments.









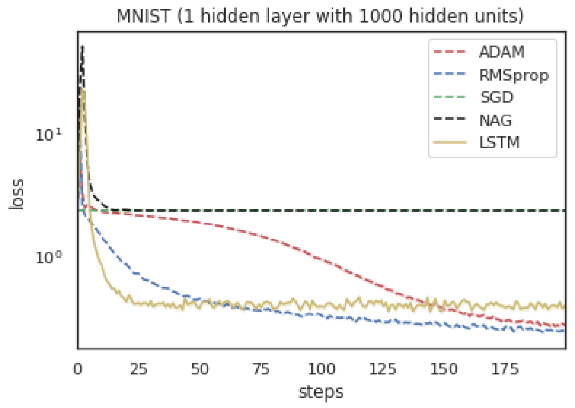


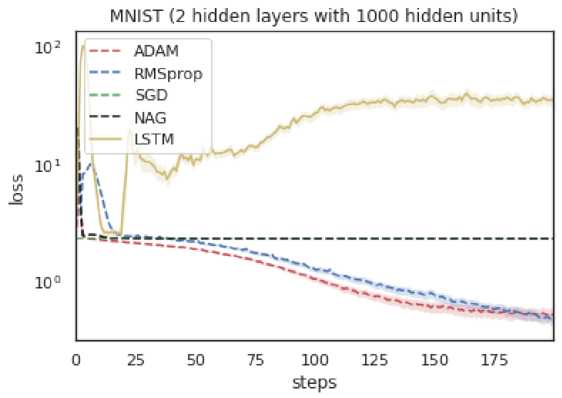
As expected, the optimizer performs worse on every increase in layers. The hand-engineered optimizers, on the other hand, are more robust to this change. Especially ADAM shows a good, stable performance, which is interesting because the authors often compare characteristics of the learned optimizer with features of ADAM.

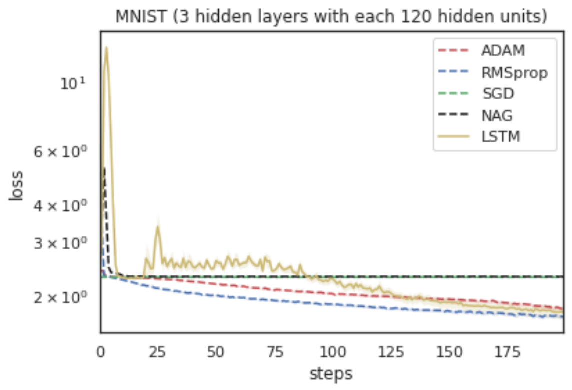
The number of hidden units that are used in a layer, however, seems to be more in the benefit of the LSTM optimizer. Interestingly, the optimizer does a better job at optimizing a multi-layered network when it has many hidden units per layer. This seems rather odd, because it adds more complexity, which we thought was not beneficial to the performance of this optimizer. A possible explanation could be that the coordinatewise setup of LSTMs is better suitable for horizontal scaling than vertical scaling because horizontal scaling does not lead to the complicated sequential computations that multiple layers can cause.

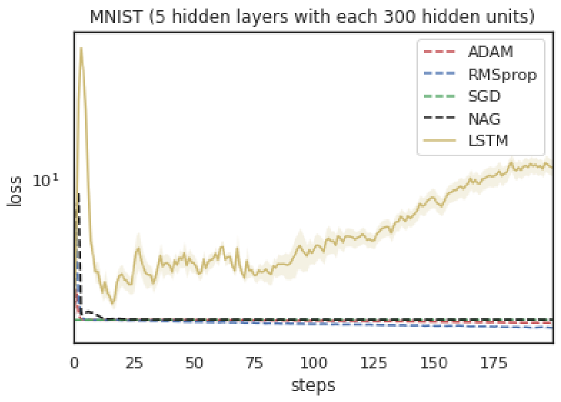
As a small follow-up experiment, we tested the boundaries of what the learned optimizer could handle in terms of a number of units in a layer of a network. It turns out that there is a limit to what the optimizer can handle in this regard as well. We suspect this to be due to the fact that the learned optimizer needs more training data to generalize on such a large scale (otherwise it may suffer from the curse of dimensionality). The figures below show the results for different (larger) layer sizes.

Layers	1	2	3	5
Hidden units per layer	1000	1000	120	300









We can now conclude with certainty, and with regret, that the learned optimizer proposed in this paper is not scalable to larger networks. While the ideas proposed in this paper are interesting and the initial results are promising, the LSTM model is simply not valuable for present-day relevant models, which are usually very large, multi-layered networks.

We attribute the deterioration in performance of the learned optimizer to the fact that, in the coordinate network, the LSTM models have shared parameters and limited memory. When the number of parameters in the network explodes, the LSTM loses precision. This can be explained by the fact that it shares parameters across networks and therefore tries to cover the optimization of a lot of optimizee parameters in a relatively small number of optimizer parameters. For comparison, we know from the description of ADAM in Part 1, that this standard optimizer simply decays the learning rate per parameter, based on a simple formula and a memory of 1 timestep per parameter. Logically, such an optimizer suffers a lot less from having to optimize a larger network.

It seems hard to imagine that the original authors have not wondered about this issue themselves. Even when considering that their results were better than ours for the 2-layered network, one would ask themselves why they did not conduct or include experiments on larger networks as well, unless they decided this was not important. Recent advancements in the field of deep learning have only led to an increase in the layer/network size, so this was perhaps not such a good assessment. Still, this paper has been cited a total of 1110 times, according to Google Scholar. This is quite a large number, and also significantly more than some other related papers published around the same time [2,3].

If we look at the context in which this paper was published, it all makes more sense, because even though the concept of meta-learning (learning to learn) has been around for a while (first introduced in the 1980s/1990s [4,5]), it has become more of a hot topic since 2015/2016; around the time of this paper's publication. Let use also be clear on the fact that the implications of having a successful model/strategy for learning to learn would be a huge development for the deep learning community [6,7], and the results shown in this paper are overwhelmingly positive and proof of potential. The combination of these two factors must have led to the paper's success.

In addition, it was around the same time (2015/2016) that the deep learning community made great advancements in producing high-performance, deep neural networks, such as ResNet. Current state-of-the-art deep architectures usually need anywhere from 15-100 layers in order to be competitive [8]. It is clear from our results that the learned optimizer would not do well on such a large scale. This is why this paper initially showed a lot of potential for learned optimizers but is not so relevant anymore nowadays.

Still, this research shows that, if they were feasible, learned optimizers would likely outperform hand-engineered optimizers. Perhaps some other researchers will be inspired by this idea and will pick up the issue of trying to extend the learning to optimize problems to larger networks in the (near) future.

Trying different learning rates

The learning rate for training the LSTM model was derived from an experiment that trained with the following learning rates in 20 epochs:
[1.0, 0.3, 0.1, 0.03, 0.01, 0.003, 0.001, 0.0003, 0.0001, 0.00003, 0.00001]
The learning rate chosen to further train is 0.003, whilst 0.01 returned lower losses. A lower rate was chosen on purpose, since the experiment was performed with only 20 epochs whilst the rest of the experiments will be performed with 100 epochs. Since the gap between 0.1 and 0.003 is rather large, we wanted to redo the experiment for determining the learning rate with lower step sizes between 0.002 and 0.006. As the eventual training will have 100 epochs the amount of epochs for this experiment will be increased from 20 to 100 as well. The result of this experiment is shown in the Table below. It appears that a learning rate of 0.004 would do best. With a lower rate, the loss starts to increase again. The estimation of the code developer to use 0.003 instead of an even lower rate is therefore quite good.

Learning rate	0.006	0.005	0.004	0.003	0.002
Loss	55.756683	55.38959	52.28015	54.798115	55.286488

Part 4 | Code optimization

As we struggled with installing the right versions of all packages, we made some changes in the existing code to make sure the code is less sensitive to versions. These changes mostly include replacing deprecated functions. For example, tqdm_notebook is replaced for tqdm.notebook. Also, all code related to plots were changed to use .lineplot() instead of .tsplot(). Furthermore, we added a requirements txt file that includes all packages needed for running the notebook with jupyter notebook, to simplify installation.

To make the repository more user friendly, we added a ReadME to guide the user through the installation. We warn the user beforehand that they need a CUDA-enabled Core, and a compatible version of Python.

Conclusion

To conclude, the paper by Andrychowicz, M. et al. showed promising results. While the experiment itself is not straightforward to reproduce, mostly because of unspecified implementation details, the reproduced results are similar to the ones in the original paper. This validates the claims of the authors. However, extending experiments to a larger scale revealed a different side to these findings. Whereas the proposed learned optimizer outperforms standard optimizers in hyper-parameter tuning on a single layer with few hidden units, its performance deteriorates rapidly when applied to larger (>2 layered) networks. Therefore, it must be concluded that the learned optimizer is not scalable. Given the context that current state-of-the-art networks are often very deep, i.e. multi-layered, the relevance of this paper has diminished over time. Future research must look into ways of improving this technique for larger networks.

References

[1]: Andrychowicz, M., Denil, M., Gomez, S., Hoffman, M. W., Pfau, D., Schaul, T., ... & De Freitas, N. (2016). Learning to learn by gradient descent by gradient descent. arXiv preprint arXiv:1606.04474.
[2]: Li, K., & Malik, J. (2017). Learning to optimize neural nets. arXiv preprint arXiv:1703.00441.
[3]: Wichrowska, O., Maheswaranathan, N., Hoffman, M. W., Colmenarejo, S. G., Denil, M., Freitas, N., & Sohl-Dickstein, J. (2017, July). Learned optimizers that scale and generalize. In International Conference on Machine Learning (pp. 3751-3760). PMLR.
[4]: https://people.idsia.ch//~juergen/diploma.html
[5]: Bengio, Y., Bengio, S., & Cloutier, J. (1990). Learning a synaptic learning rule. Université de Montréal, Département d'informatique et de recherche opérationnelle.
[6]: https://bair.berkeley.edu/blog/2017/07/18/learning-to-learn/
[7]: Vanschoren, J. (2019). Meta-learning. In Automated Machine Learning (pp. 35-61). Springer, Cham.
[8]: Targ, S., Almeida, D., & Lyman, K. (2016). Resnet in resnet: Generalizing residual architectures. arXiv preprint arXiv:1603.08029.