COMP6258: Reproducibility Challenge ‘Gradient Descent: The Ultimate Optimizer’

Benjamin SANATI (BES1G19) Joel EDGAR (JE1G19) Charles POWELL (CP6G18)

Abstract

Optimising machine learning models using a gradient-based approach involves the laborious tuning of hyper-parameter values. Recent work has sought to address this issue by implementing hyper-optimisers that make use of automatic differentiation to compute the optimal hyperparameter values during the standard training process. In their findings, this work showed hyperoptimsers to out-perform standard implementations across a range of different neural network models and optimiser functions. This report presents an assessment of the reproducibility of this work, considering if both its implementation details can be followed and its findings reproduced, as well as exploring some aspects that the work fails to address. The findings support the claims proposed by Chandra et al. (2022) and provides further insight regarding additional features of interest in more depth.

1 Introduction

Working with any gradient-based machine learning optimiser requires the tuning of hyper-parameters such as the learning rate. Selection of these hyperparameters is crucial to the performance of the model, and thus becomes an optimisation task in-itself. As such, it is reasonable to question if the same gradient-based optimisers can be used recursively to compute their own hyperparameters. In doing so, so-called ‘hyperoptimisers’ can be created that consider the derivative of the loss function with respect to the hyperparameters to update their values during training alongside the regular parameters of the model Chandra et al. (2022).

Chandra et al. (2022) makes use of automatic differentiation to implement hyperoptimisers that can efficiently compute optimal values for all hyperparameters, and that can be applied recursively on-top of each other to the nth degree. When investigating the performance of these implementations, Chandra et al. (2022) find that the hyperoptimisers are able to outperform standard implementations across a range of neural network models and optimiser functions. Additionally, the hyperoptimisers can successfully be stacked on-top of each other to produce higher-order hyperoptimizers.

This report details an assessment of the reproducibility of Chandra et al. (2022), considering if both its implementation details can be followed, and if the same results can be obtained. In addition, this assessment also aims to address some aspects of the problem that the report does not investigate in detail, such as how the hyperparameter values change during training and the effect of stacking hyperoptimisers on the overall performance of the models.

2 Methodology

The primary methodology of this report’s assessment was to replicate all of the implementation details and experiments of the original work and to comparatively analyse the obtained results with those it provides. During implementation, this involved the development of hyperoptimisers for an MLP, CNN and RNN according to the work’s description, as well as supporting the use of alternative optimiser functions for the MLP implementation. In experimentation, this involved investigating the effects of the hyperoptimisers on model performance, the effect of the hyperoptimiser on individual hyperparameter values, and the effects of recursively stacking hyperoptimisers.

Additionally, this assessment also sought to investigate some aspects that the original work does not address in sufficient detail. Specifically, the assessment considered how the hyperparameters of an optimiser change during training, investigated additional combinations of optimiser and hyperoptimiser in the case of MLPs, and carried out more detailed investigations to analyse the effects of stacking on model performance.

In implementation, this assessment implemented its own models, training procedure and testing procedure, but made use of the original work’s code for the implementation of the hyperoptimisers. Note, however, that this code was adjusted in places to more closely reflect the understanding achieved based on the explanations provided in the original work. When training the models and carrying out experiments, the University of Southampton Alpha Cluster was utilised with a configuration of 4 Quadro RTX 8000 GPUs to ensure speed up and avoid CUDA out-of-memory issues. This was implemented using PyTorch's `DataParrallel` module to parallelise the forward and backward passes across the multiple GPUs. Additionally, the `autocast()` method was used to reduce computations to 16-bit floating-point precision. All of the code for this assessment, along with the full set of experimental results and figures is available at the following GitHub repository : https://github.com/ben-sanati/COMP6258.

3 Analysis

Provided here is a comparative analysis of this reports findings in relation to those of the original work. Some example figures are given throughout, though the full list of figures and findings is available in the Github repository.

3.1 MLP

To investigate the performance of hyperoptimsers on MLPs, two sets of 3 layered MLPs were defined, each with 784 input neutrons, 128 hidden neutrons and 10 output neutrons, as per the original work. The first set served as a baseline and contained one MLP for each of the optimiser functions listed in the original work with no hyperoptimisation applied, while the second set contained one MLP for each combination of optimiser and hyperoptimiser. Each MLP was then trained on the MNIST dataset (Deng (2012)) for 30 epochs, as opposed to the 5 epochs used in the original work in order to allow for the investigation of the effects of the hyperoptimisers over a greater length of time.

In testing, the original work finds that the MLPs defined with hyperoptimisers outperform the baseline models, and that minimal changes are made to parameters other than the learning rate. As shown in 1, the findings of this assessment support these results over a small number of training epocshs, but find that for larger training durations, the baseline models can achieve a similar, or better performance than their hyperoptimised counterparts. That is, the hyperoptimised models achieve faster convergence, but do not always achieve improved performance. This assessment also considered additional combinations of optimiser and hyperoptimiser in comparison to the original work, and found the results to be synonymous to those given by the original combinations. Note however, that the use of an Adam optimiser and Adam hyperoptimiser failed due to the omission of a section of code from the original implementation that was neither understood, nor explained in the work.

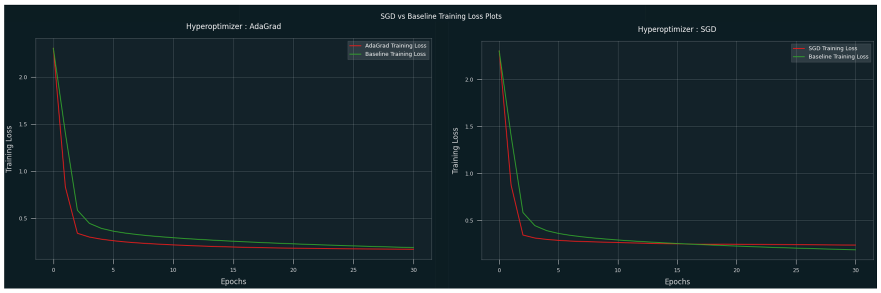


Figure 1: Graphs showcasing the comparison in loss between a hyperoptimised and baseline MLP, for varying hyperoptimiser functions.

Additionally, while not discussed in the original work, this assessment found that the optimisation of hyperparameters was more dependent on the optimiser than the hyperoptimiser. This can most likely be attributed to the fact that it is the optimiser that traverses the loss landscape, and not the hyperoptimiser.

Also investigated in this assessment was the effect of the hyperoptimiser on hyperparameter values over time. In experimentation, it was found that the hyperparameters often saw a major increase initially, before quickly reducing becoming slightly stochastic. This can likely be attributed to the initial hyper parameter value being far away from its optimal, and thus a large update being induced.

3.2 CNN

As done in Chandra et al. (2022), we train a ResNet-20 He et al. (2016) using an SGD optimizer with SGD hyperoptimization on the CIFAR-10 dataset Krizhevsky (2009) for 50 epochs. Chandra et al. (2022) aimed to investigate the robustness of the hyperoptimizer to varying choices of step size α and momentum μ. In our reproduction, we also aimed to examine the effects of hyperoptimizing hyperparameters that are not α.

Each variation of the Resnet-20 model was trained and evaluated on the same set of hyperparameter values used in the paper: $\{\alpha = [0.01, 0.1, 1.0], \mu = [0.09, 0.9, 0.99], \kappa = 0.01, \mu = 0\}$. The team could reproduce the scheduling-like performance for multiple variations of α and μ as identified in Chandra et al. (2022).

Our additional investigation of μ allowed us to identify an increase in the scale of the change of the μ hyperparameter when α_{init} is set to larger values. For small and well-defined α_{init}, the changes in μ occur over a small range. For large α_{init} values, however, μ is dramatically reduced and follows a similar scheduling scheme to that of the learning rate. This reinforces the idea that the hyperoptimizer makes the optimization scheme more robust to 'bad' initialisations of the hyperparameters and adjusts the hyperparameters accordingly.

Additional observations regarding the effects of hyperparameters during training were made. These include: (a) the closer μ_{init} and α_{init} are to 'good' values, the smaller the change in their values over time and the faster their convergence 2, (b) the use of a hyperoptimizer makes the training loss plot less stable due to changes in the alpha value, and (c) sudden reductions in α and μ align with sudden spikes in the loss that restabilise after a few epochs, however, this promotes more precise navigation of the loss landscape.

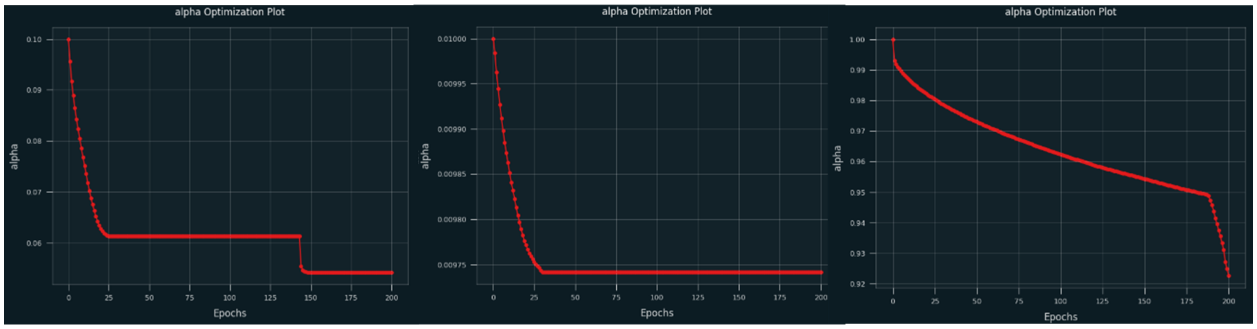
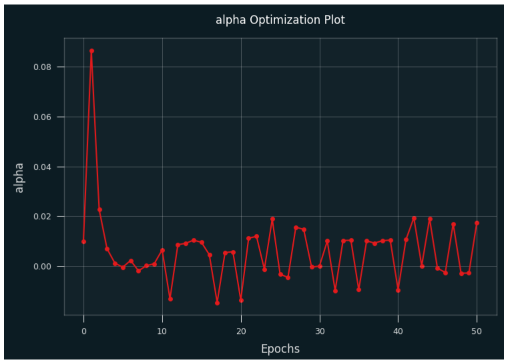
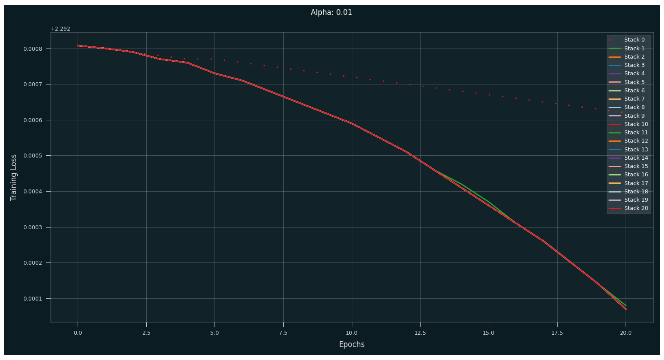


Figure 2: A set of graphs showcasing the change in hyperparameter values against epochs during the training of 3 ResNet-20 models He et al. (2016) with $\{\alpha = 0.01, \mu = 0.09\}, \{\alpha = 0.1, \mu = 0.9\}, \{\alpha = 1.0, \mu = 0.99\}$={left, bottom, right}. We can see the faster convergence and increase in stability for both α and μ when they are better initialised.

3.3 RNN



(a) Alpha vs Epochs for an Adam on Adam optimiser with alpha = 0.001.



(b) Graph showing the effect of various sizes of hyperoptimiser stacks on the loss.

Figure 3: Two figures side by side

Despite the authors of the paper showing the changes in alpha over time for the CNN, they didn't show the same for the RNN. As part of our study we thought it would be worthwhile investigating this. Looking at the behaviour of alpha for various different initial states, all hyperoptimisers cause a large initial jump in alpha, for example, a = 1e-4 had a nearly 500x increase in magnitude. An example of this can be seen in Figure 3a. This causes the steep drop in loss that is common for all iterations of CharRNN. However, past the initial 10 epochs the behaviour of alpha is very erratic, showing no real trend besides oscillating around 0. For each of the various initial alpha values, it seems that the kappa value used for the hyper optimiser is causing the system to overshoot the optimum value of alpha, causing the alpha value to jump from positive to negative repeatedly. This oscillatory behaviour suggests that there is an optimum value of Kappa, meaning the methodology has essentially replaced one parameter for another. None of the final loss values for hyperoptimisers out performed the baseline, although they were very close.

Experiments comparing loss between the baseline and various hyperoptimiser versions like the ones shown in the paper were also replicated. However, instead of recording loss after each optimiser step it was recorded after every epoch. As in the paper, the replicated experiments showed an acceleration of the decrease of the loss which, when looking at Figure 3a can be explained by the large increase in alpha initially. Thus, the hyper optimisers promote a faster convergence at the cost of a very slight increase in loss.

3.4 Stacking

The original paper also investigates the effects of stacking "towers" of hyperoptimisers. In the original paper, the authors found that the practical application of the towers was sensitive to the choice in step size: choosing too large of a step size could result in divergence. The replicated experiments agree with this finding, however, the provided schemes did not fix the issue and instead a new function to decide the step size for successive layers was applied $\alpha_{\text{layer}} = \frac{\alpha_{init}}{100(1+\text{layer})}$. Using a function like this had the added benefit of being able to define arbitrarily tall towers without having to define a learning parameter for each layer.

The experiments in this study used towers far taller than any of the ones used in the original paper. Figure 3b shows that there is a strong diminishing return for the improvement of performance for increasingly tall towers of hyperoptimisers, but the author's claim that this results in a minimal memory impact was empirically found to be true. Unfortunately, due to computing restraints, the overall claim that there is a minimal increase in time for required training could not be verified. Since the experiments were run on the University Alpha cluster, which is a shared resource, the time taken to run experiments depends on the demand for compute resources which could not be controlled.

4 Conclusion

This report aimed to reproduce and further investigate the use of hyperoptimizers by Chandra et al. (2022) to reduce the impact of hyperparameter tuning on the training process of deep learning models. The findings support the claims proposed by Chandra et al. (2022) and provided additional insight regarding the effects of hyperoptimizers on hyperparameter values during the training over a variety of deep learning models.

Similar or improved performance of all model varieties and hyperparameter initialisations against a baseline was observed, with common characteristics regarding changes in hyperparameter values during training that was not mentioned in Chandra et al. (2022) being identified.

The report additionally investigated the impact of using higher-order hyperoptimizers than those used in the paper, identifying diminishing returns in performance for every higher-order hyperoptimizer applied to the stack.

Future work should aim to further investigate the effect of these taller high-order hyperoptimizers; in particular, the temporal and robustness effects of very tall hyperoptimizers. There should also be work put towards the production of a better function for identifying κ_{layer}.

In conclusion, this report successfully reproduced the majority of findings in Chandra et al. (2022) and investigated additional features of interest in more depth.

References

Kartik Chandra, Audrey Xie, Jonathan Ragan-Kelley, and Erik Meijer. Gradient descent: The ultimate optimizer, 2022.

Li Deng. The mnist database of handwritten digit images for machine learning research. *IEEE Signal Processing Magazine*, 29(6):141–142, 2012.

Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pp. 770–778, 2016.

Alex Krizhevsky. Learning multiple layers of features from tiny images. 2009.