The Art of Hypertuning: A Comparative Analysis of Different Optimization Techniques

Ana Lucia Carrizo[1], Amine Tourki[2], Mortadha Abderrahim[3]
Swiss Federal Institute of Technology Lausanne

Abstract—**Fine tuning hyperparameters is one of the most challenging tasks in deep learning. Learning rate tuning has been a major focus and methods like schedulers have been a major success. Over recent years, researchers have shown interest over other parameters such as momentum with optimizers like momentum SGD or Nesterov Accelerated Gradient (NAG). In this project, we further explore momentum tuning techniques by implementing two state-of-the-art optimizers: Demon and YellowFin. We compare their performances with already established optimizers like Adam and SGD. We conclude on the performance of momentum tuning techniques and their viability.**

Key words— **momentum tuning, hyperparameters, optimization, deep neural networks.**

I. Introduction

Optimizers rely on hyper-parameter tuning to achieve good performances. Learning rate plays a major role in hyperparameter tuning. Selection of a good learning rate, learning rate decay[1] and decay schedulers[2] can massively improve the performance of a network. The literature is full with amazing results, which are mostly due to spending immense resources, in terms of time and computing power, exploring the hyperparameter space.

Adaptive methods, such as Adam, make our life easier by tuning for individual variables and achieve remarkable results. Nevertheless, they still not manage to outperform hand-tuned methods, like SGDM, which suggests that adaptive methods can suffer from lack of generalization [3]. The importance of momentum [4] has been mostly ignored by the community, making it so that there is much to be done in terms of optimization. Momentum is designed to speed up the learning in direction of low curvatures while retaining stability in directions of high curvatures.

In this project we will analyze momentum tuning techniques through two state-of-the-art optimizers: Demon[5] and YellowFin[6]. We will study their performance and compare them with classical methods such as Adam[7] or SGD.

II. Optimization Methods

A. SGD

Stochastic Gradient Descent (SGD) is one of the standard optimization methods. It is an iterative algorithm that updates the model's weights θ based on the gradient of the loss function at a randomly sampled data point $(x^{(i)}, y^{(i)}) \sim D$.

The update rule is given by:

$$\theta = \theta - \eta \nabla_\theta J(\theta; x^{(i)}; y^{(i)})$$

where $J(\theta; x^{(i)}; y^{(i)})$ is the loss function, and η the learning rate.

SGD with momentum

SGDM [8] was born out of a need to learn how to navigate ravines, i.e. areas where the surface curves much more steeply in one dimension than in another, which are quite common around local optimum. The momentum term increases for dimensions whose gradients point in the same directions and reduces updates for dimensions whose gradients change directions. Resulting in faster convergence and reduced oscillation. The update rule is given by:

$$v_t = \gamma v_{t-1} + \eta \nabla_\theta J(\theta)$$
$$\theta = \theta - v_t$$

where γ, the momentum term, is usually set to 0.9.

Nesterov Accelerated Gradient

With SGDM we managed to accumulate momentum, so that it becomes faster and faster. Nevertheless, our optimizer still lacks a notion of where it is going so that it knows how to respond accordingly. NAG [9] is a way to give our momentum this kind of prescience. It gets a rough idea of where the parameters are going to be by calculating the gradient w.r.t the approximate future position of our parameters:

$$v_t = \gamma v_{t-1} + \eta \nabla_\theta J(\theta - \gamma v_{t-1})$$
$$\theta = \theta - v_t$$

This anticipatory update allows us to control the acceleration and results in greater responsiveness.

B. ADAM

Adaptive Moment Estimation was first published in 2014 [10] and remains to this day the most popular adaptive learning rate algorithm. It is a combination of the Stochastic Gradient Descent with momentum and the RMSProp algorithm, and was designed specifically for training deep neural networks. It inherits the strengths of both techniques, resulting in a more optimized adaptive learning rate method. It keeps an exponentially decaying averge of past gradients

[1]Msc. Data Science, Swiss Federal Institute of Technology Lausanne, e-mail: ana.carrizodelgado@epfl.ch.
[2]Msc. Robotics, Swiss Federal Institute of Technology Lausanne, e-mail: amine.tourki@epfl.ch.
[3]Msc. Data Science, Swiss Federal Institute of Technology Lausanne, e-mail: mortadha.aberrahim@epfl.ch.

with discount factor β_1: $\xi^g_{t+1} = \beta_1 \cdot \xi^g_t + (1-\beta_1)\cdot g_t$, leading to the recursion:

$$\theta_{t+1,i} = \theta_{t,i} - \frac{\eta}{\sqrt{\xi^{g\circ g}_{t+1,i} + \varepsilon}} \cdot \xi^g_{t+1,i}$$

where usually $\beta_1 = 0{,}9$ and $\beta_2 = 0{,}999$.

C. Learning Rate Schedulers

Further optimizations were done on the Adam and SGD models by implementing different learning rate schedulers, which seek to adjust the learning rate during training by reducing the learning rate according to a predefined schedule.

- **Lambda LR**: sets the learning rate of each parameter to the initial *lr* times a given function.

$$lr_{\text{epoch}} = lr_{\text{initial}} * \text{Lambda(epoch)}$$

- **Linear LR**: decays the learning rate of each parameter group by linearly changing small multiplicative factor until the number of epoch reaches a predefined milestone.
- **Cosine Annealing LR**: sets the learning rate of each parameter group using a cosine annealing schedule. It relies on the observation that we might not want to decrease the learning rate too drastically in the beginning and moreover, that we might want to refine"the solution in the end using a very small learning rate [11]. The learning rate at each step becomes:

$$\eta_t = \eta_{\text{mín}} + \frac{1}{2}(\eta_{\text{máx}} - \eta_{\text{mín}})\left(1 + \cos\left(\frac{T_{cur}}{T_{\text{máx}}}\pi\right)\right)$$

- **Exponential LR**: decays the learning rate of each parameter group by gamma every epoch.

$$lr_{\text{epoch}} = Gamma * lr_{\text{epoch}-1}$$

D. Demon Adam

Demon Adam is a stochastic optimizer obtained by applying the Demon momentum rule to the Adam optimizer.

The Demon momentum rule is a momentum decaying method that can be applied to any optimizer with a momentum. As described earlier, one of the most simple momentum methods, SGDM, usually set 0.9 as default value for the momentum parameter. However, papers such as [12] suggest that using said default value could lead to momentum overdose and performance degradation.

The motivation of using a decaying momentum is to reduce the impact of a gradient to the current and future updates. Applied to Adam, the Demon momentum rule is:

$$\begin{aligned}
\beta_t &= \beta_{init} \cdot \frac{(1-\frac{t}{T})}{(1-\beta_{init}) + \beta_{init}1 - \frac{t}{T})} \\
\xi^{g\circ g}_{t+1} &= \beta_2 \xi^{g\circ g}_t + (1-\beta_2)\cdot(g_t \circ g_t) \\
m_{t,i} &= g_{t,1} + \beta_t m_{t-1,i} \\
\theta_{t+1,i} &= \theta_{t,i} - \frac{\nu}{\sqrt{\xi^{g\circ g}_{t+1,i} + \epsilon}} \cdot m_{t,i}
\end{aligned}$$

with β_2 and ϵ hyperparameters.

E. YellowFin

YellowFin is an automatic hyperparameter tuner for momentum SGD. The algorithm simultaneously tunes the learning rate and momentum. YellowFin finds its motivation in the robustness of the momentum operator to learning rate misspecification, meaning a tolerance to a less-carefully-tuned learning rate. The algorithm is also inspired by the robustness of the momentum operator to curvature variation meaning empirical linear convergence on a class of non-convex objectives with varying curvatures.

The implementation of YellowFin follows the algorithm 1, where D is an estimate of the current model's distance to a local quadratic approximation's minimum. C denotes an estimate for gradient variance. h_{max}, h_{min} denote estimates for the largest and smallest generalized curvature respectively. YellowFin uses measure functions CurvatureRange, Variance, Distance to measure these quantities.

Algorithm 1 YellowFin

function `YellowFin`(g_t, β)**:**
  $c \longleftarrow CurvatureRange(g_t, \beta)$
  $C \longleftarrow Variance(g_t, \beta)$
  $D \longleftarrow Distance(g_t, \beta)$
  $\mu_t, \alpha_t \longleftarrow SingleStep(C, D, h_{max}, h_{min})$
  return μ_t, α_t
end function

III. Experimental Tests

A. Problem Setup

Our experiment consists in comparing the performance of the previously discussed methods on a simple image classification task, classifying handwritten digits. To run our experiments we took advantage of Google Colab's K80 GPUs.

B. Dataset

We evaluated all models on the MNIST dataset, which consists of 60,000 samples in the training set and 10,000 samples in the test set. Each sample is an image of 28x28 pixels representing a handwritten number.

C. Neural Net Architecture

In order fully grasp the contribution of the chosen optimization technique, we run our experiments on

a simple deep neural network architecture. Our net is a fully connected neural network with 3 layers, taking as initial input the image of size 28x28 and outputs a tensor of size 256. The second layer takes as input the output from the previous layer and returns a tensor of size 100, which will serve as input for the next layer. The third layer returns a tensor of size 10, and finally we apply a ReLU activation function.

D. Hypertuning and Cross-Validation

To obtain the best parameters for each model, we do a classic grid search. We select the configuration that yields the highest accuracies when running a K-Fold Cross-Validation. For all models we fix the number of epochs to 25, and the number of folds to 5. Then, we proceed to try out all possible configurations, tuning the learning rate, the batch size and other model-specific variables.

E. Results

Our results for all models are summarized in table I. Similarly, we can see the evolution of our results per model in graph 1. The Loss plot can be found in the appendix in figure 2

We can see that all optimizers achieve good results. When it comes to models that implement Adam, using a LambdaLR scheduler yields the best performance overall for our given task. Apart from the CosineAnnealingLR, all other Adam based optimizers perform very well on this architecture. Adam Demon does not outperform the best model on this dataset, but performs second best. YellowFin on the other hand, achieves good results overall but fails to outperform all the standard optimizers. SGD based models perform the best with this architecture. SGD Nesterov in particular achieves the best results followed by standard SGD. We can also observe that almost all the optimizers stabilize after a few epoches and hit a plateau with little improvement afterward, especially ADAM with ExponentialLR scheduler. YellowFin oscillate the most and have a hard time converging to a final loss.

Table I: Accuracy average over three runs. for each Optimizer. In bold the best accuracies for Adam based and SGD based models

Optimizer	Best	Last
SGD	0.9842	0.9842
SGDM	0.9829	0.9828
Nesterov SGD	**0.9853**	0.9853
Adam+LinearLR	0.9816	0.9823
Adam+ExponentialLR	0.9798	0.9798
Adam+CosineAnnealingLR	0.9682	0.9220
Adam+LambdaLR	**0.9841**	0.9841
YellowFin	0.9476	0.9208
Adam Demon	0.9814	0.98074

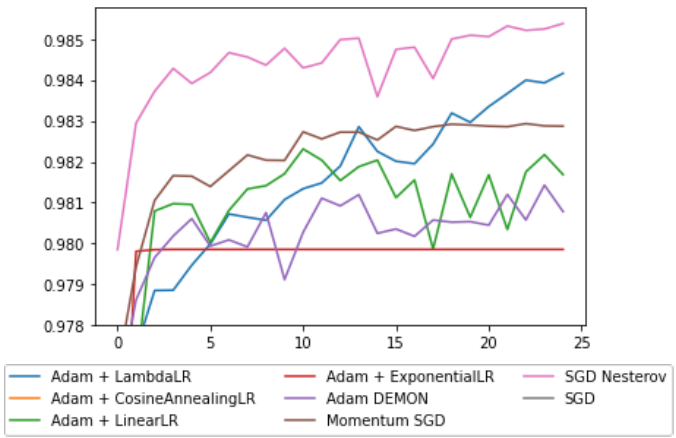

Fig. 1: Accuracy results of the models

IV. Discussion

Based on the results previously exhibited, we make the following observations:

- We observed that ADAM Demon performs well in general when compared to other models; while YellowFin performs worse on average. We conclude that hypertuning each model, while very time-consuming, can yield very good results with optimizers based on ADAM and SGD.
- YellowFin optimizes the hyperparameters to minimize the loss and does not require fine tuning which can save a lot of time. However, it cannot outperform finely tuned models, as shown in our experiment.
- Demon, on the other hand, can be easily tuned and performs strongly on this architecture with a near state-of-the-art performance.

In conclusion, we still recommend the use of YellowFin in cases where no budget for hypertuning is available as it can self-tune both the learning rate and the momentum. If budget is not a problem, Adam Demon can be a good choice since it can deliver top performances and maintain robustness. Classical optimizers with different schedulers are still always viable and can perform very well when finely tuned. Of course, the chosen architecture and the dataset play a major role and can ultimately be the deciding factor for which optimizer to choose.

V. Summary

In this project, we compared the performance of Adam and SGD based optimization techniques to two state-of-the-art momentum tuning optimizers: Demon and YellowFin. We used the same deep neural network architecture and trained the models on the same dataset. We fine tuned each model's hyperparameter using a gridsearch and evaluated their performances based on their accuracies and cross-entropy losses.

VI. Appendix

The best parameters per model can be summed up in table II.

Table II: Best Hyperparameters per Model

Model	Configuration
SGD	lr: 0.2, batch_size: 90
SGDM	lr: 0.02, momentum: 0.02, batch_size: 90
Nesterov SGD	lr: 0.05, momentum: 0.9, batch_size: 90
Adam + LinearLR	lr: 0.002, batch_size: 180, start_factor: 0.03
Adam + ExponentialLR	lr: 0.002, batch_size: 90, gamma: 0.03
Adam + CosineAnnealingLR	lr: 0.002, batch_size: 120, T_max: 5, eta_min: 0.02
Adam + LambdaLR	lr: 0.002, batch_size: 90, lr_lambda: 0.95 ** epoch
YellowFin	lr: 0.02, batch_size: 180, wd: 0, mu: 0, view_every: 1000
Adam Demon	lr: 0.002, batch_size: 120, wd: 0, view_every: 1000

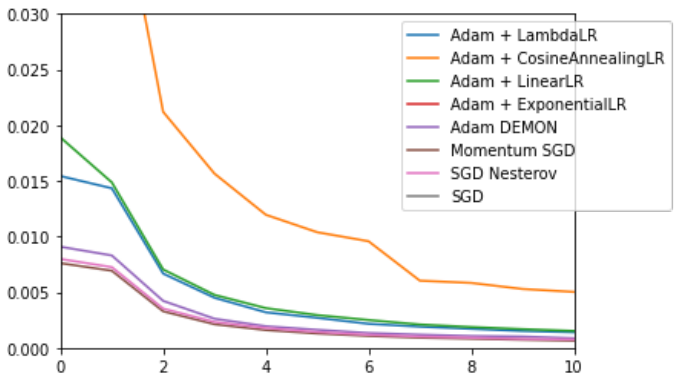

Fig. 2: Cross-Entropy Loss of the models over the epoches

References

[1] Kaichao You, Mingsheng Long, Jianmin Wang, and Michael I. Jordan, "How does learning rate decay help modern neural networks?," 2019.

[2] Zeke Xie, Issei Sato, and Masashi Sugiyama, "Understanding and scheduling weight decay," 2022.

[3] Ashia C Wilson, Rebecca Roelofs, Mitchell Stern, Nati Srebro, and Benjamin Recht, "The marginal value of adaptive gradient methods in machine learning," *Advances in neural information processing systems*, vol. 30, 2017.

[4] Ilya Sutskever, James Martens, George Dahl, and Geoffrey Hinton, "On the importance of initialization and momentum in deep learning," in *International conference on machine learning*. PMLR, 2013, pp. 1139–1147.

[5] John Chen, Cameron Wolfe, Zhao Li, and Anastasios Kyrillidis, "Demon: Improved neural network training with momentum decay," 2019.

[6] Jian Zhang and Ioannis Mitliagkas, "Yellowfin and the art of momentum tuning," 2017.

[7] Diederik P. Kingma and Jimmy Ba, "Adam: A method for stochastic optimization," 2014.

[8] Ning Qian, "On the momentum term in gradient descent learning algorithms," *Neural networks*, vol. 12, no. 1, pp. 145–151, 1999.

[9] Yurii Nesterov, "A method for unconstrained convex minimization problem with the rate of convergence o (1/kˆ 2)," in *Doklady an ussr*, 1983, vol. 269, pp. 543–547.

[10] Diederik P Kingma and Jimmy Ba, "Adam: A method for stochastic optimization," *arXiv preprint arXiv:1412.6980*, 2014.

[11] Ilya Loshchilov and Frank Hutter, "Sgdr: Stochastic gradient descent with warm restarts," *arXiv preprint arXiv:1608.03983*, 2016.

[12] Ioannis Mitliagkas, Ce Zhang, Stefan Hadjis, and Christopher Ré, "Asynchrony begets momentum, with an application to deep learning," in *2016 54th Annual Allerton Conference on Communication, Control, and Computing (Allerton)*, 2016, pp. 997–1004.