COCKPIT: A Practical Debugging Tool for the Training of Deep Neural Networks

Frank Schneider*
University of Tübingen
Maria-von-Linden-Straße 6
Tübingen, Germany
f.schneider@uni-tuebingen.de

Felix Dangel*
University of Tübingen
Maria-von-Linden-Straße 6
Tübingen, Germany
f.dangel@uni-tuebingen.de

Philipp Hennig
University of Tübingen &
MPI for Intelligent Systems
Tübingen, Germany
philipp.hennig@uni-tuebingen.de

Abstract

When engineers train deep learning models, they are very much "flying blind". Commonly used methods for real-time training diagnostics, such as monitoring the train/test loss, are limited. Assessing a network's training process solely through these performance indicators is akin to debugging software without access to internal states through a debugger. To address this, we present COCKPIT, a collection of instruments that enable a closer look into the inner workings of a learning machine, and a more informative and meaningful status report for practitioners. It facilitates the identification of learning phases and failure modes, like ill-chosen hyperparameters. These instruments leverage novel higher-order information about the gradient distribution and curvature, which has only recently become efficiently accessible. We believe that such a debugging tool, which we open-source for PYTORCH, is a valuable help in troubleshooting the training process. By revealing new insights, it also more generally contributes to explainability and interpretability of deep nets.

1 Introduction and motivation

Deep learning represents a new programming paradigm: instead of deterministic programs, users design models and "simply" train them with data. In this metaphor, deep learning is a meta-programming form, where *coding* is replaced by *training*. Here, we ponder the question how we can provide more insight into this process by building a *debugger* specifically designed for deep learning.

Debuggers are crucial for traditional software development. When things fail, they provide access to the internal workings of the code, allowing a look "into the box". This is much more efficient than re-running the program with different inputs. And yet, deep learning is arguably closer to the latter. If the attempt to train a deep net fails, a machine learning engineer faces various options: Should they change the training hyperparameters (how?); the optimizer (to which one?); the model (how?); or just re-run with a different seed? Machine learning toolboxes provide scant help to guide these decisions.

Of course, traditional debuggers can be applied to deep learning. They will give access to every single weight of a neural net, or the individual pixels of its training data. But this rarely yields insights

*Equal contribution

35th Conference on Neural Information Processing Systems (NeurIPS 2021).

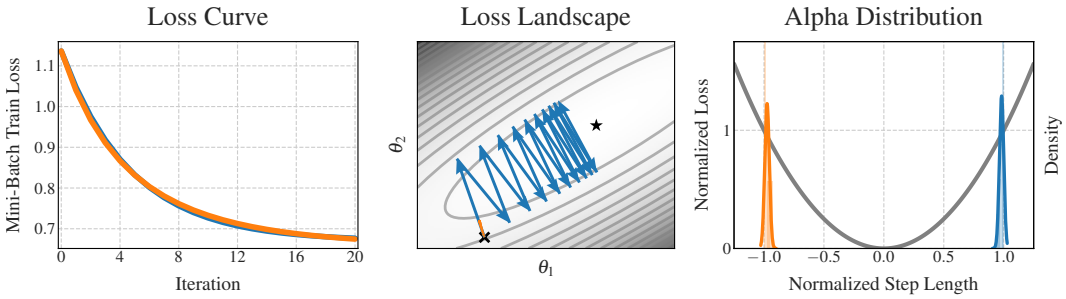


Figure 1: **Illustrative example: Learning curves do not tell the whole story**. Two different optimization runs (—/—) can lead to virtually the same loss curve (*left*). However, the actual optimization trajectories (*middle*), exhibit vastly different behaviors. In practice, the trajectories are intractably large and cannot be visualized directly. Recommendable actions for both scenarios (increase/decrease the learning rate) cannot be inferred from the loss curve. The α-distribution, one COCKPIT instrument (*right*), not only clearly distinguishes the two scenarios, but also allows for taking decisions regarding how the learning rate should be adapted. See Section 3.3 for further details.

towards successful training. Extracting meaningful information requires a statistical approach and distillation of the bewildering complexity into a manageable summary. Tools like TENSORBOARD [1] or WEIGHTS & BIASES [6] were built in part to streamline this visualization. Yet, the quantities that are widely monitored (mainly train/test loss & accuracy), provide only scant explanation for relative differences between multiple training runs, because *they do not show the network's internal state*. Figure 1 illustrates how such established learning curves can describe the *current* state of the model – whether it is performing well or not – while failing to inform about training state and dynamics. They tell the user *that* things are going well or badly, but not *why*. The situation is similar to flying a plane by sight, without instruments to provide feedback. It is not surprising, then, that achieving state-of-the-art performance in deep learning requires expert intuition, or plain trial & error.

We aim to enrich the deep learning pipeline with a visual and statistical debugging tool that uses newly proposed observables as well as several established ones (Section 2). We leverage and augment recent extensions to automatic differentiation (i.e. BACKPACK [12] for PYTORCH [33]) to efficiently access second-order statistical (e.g. gradient variances) and geometric (e.g. Hessian) information. We show how these quantities can aid the deep learning engineer in tasks, like learning rate selection, as well as detecting common bugs with data processing or model architectures (Section 3).

Concretely, we introduce COCKPIT, a flexible and efficient framework for online-monitoring these observables during training in carefully designed plots we call "instruments" (see Figure 2). To be of practical use, such visualization must have a manageable computational overhead. We show that COCKPIT scales well to real-world deep learning problems (see Figure 2 and Section 4). We also provide three different configurations of varying computational complexity and demonstrate that their instruments keep the computational cost *well below* a factor of 2 in run time (Section 5). It is available as open-source code,[2] extendable, and seamlessly integrates into existing PYTORCH training loops (see Appendix A).

2 COCKPIT's instruments

Setting: We consider supervised regression/classification with labeled data $(\boldsymbol{x}, \boldsymbol{y}) \in \mathbb{X} \times \mathbb{Y}$ generated by a distribution $P(\boldsymbol{x}, \boldsymbol{y})$. The training set $\mathcal{D} = \{(\boldsymbol{x}_n, \boldsymbol{y}_n) \mid n = 1, \dots, N\}$ consists of N i.i.d. samples from P and the deep model $f : \Theta \times \mathbb{X} \to \mathbb{Y}$ maps inputs $\boldsymbol{x}_n$ to predictions $\hat{\boldsymbol{y}}_n$ by parameters $\boldsymbol{\theta} \in \mathbb{R}^D$. This prediction is evaluated by a loss function $\ell : \mathbb{Y} \times \mathbb{Y} \to \mathbb{R}$ which compares to the label $\boldsymbol{y}_n$. The goal is minimizing an inaccessible expected risk $\mathcal{L}_P(\boldsymbol{\theta}) = \int \ell(f(\boldsymbol{\theta}, \boldsymbol{x}), \boldsymbol{y}) \, \mathrm{d}P(\boldsymbol{x}, \boldsymbol{y})$ by empirical approximation through $\mathcal{L}_\mathcal{D}(\boldsymbol{\theta}) = \frac{1}{N} \sum_{n=1}^{N} \ell(f(\boldsymbol{\theta}, \boldsymbol{x}_n), \boldsymbol{y}_n) := \frac{1}{N} \sum_{n=1}^{N} \ell_n(\boldsymbol{\theta})$, which

[2] https://github.com/f-dangel/cockpit

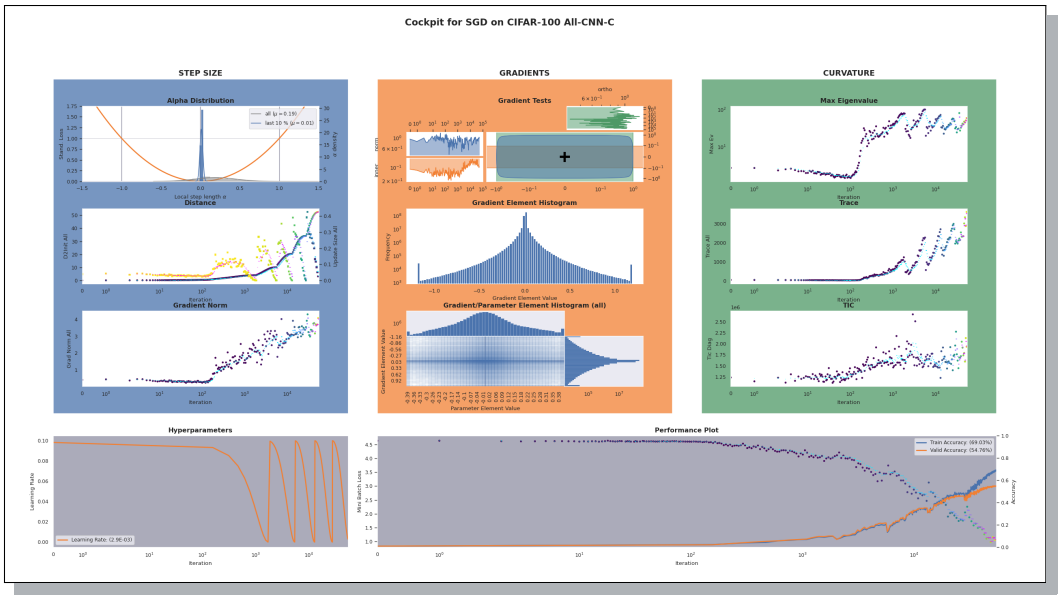


Figure 2: **Screenshot of COCKPIT's full view** while training the ALL-CNN-C [41] on CIFAR-100 with SGD using a cyclical learning rate schedule. This figure and its labels are not meant to be legible, but rather give an impression of COCKPIT's user experience. Gray panels (bottom row) show the information currently tracked by most practitioners. The individual instruments are discussed in Section 2, and observations are described in Section 4. An animated version can be found in the accompanying GitHub repository.

in practice though can only be stochastically sub-sampled on mini-batches $\mathcal{B} \subseteq \{1, \dots, N\}$,

$$\mathcal{L}_{\mathcal{B}}(\boldsymbol{\theta}) = \frac{1}{|\mathcal{B}|} \sum_{n \in \mathcal{B}} \ell_n(\boldsymbol{\theta}) \,. \tag{1}$$

As is standard practice, we use first- and second-order information of the mini-batch loss, described by its gradient $\boldsymbol{g}_{\mathcal{B}}(\boldsymbol{\theta})$ and Hessian $\boldsymbol{H}_{\mathcal{B}}(\boldsymbol{\theta})$,

$$\boldsymbol{g}_{\mathcal{B}}(\boldsymbol{\theta}) = \frac{1}{|\mathcal{B}|} \sum_{n \in \mathcal{B}} \underbrace{\nabla_{\boldsymbol{\theta}} \ell_n(\boldsymbol{\theta})}_{\boldsymbol{g}_n(\boldsymbol{\theta})} \,, \qquad \boldsymbol{H}_{\mathcal{B}}(\boldsymbol{\theta}) = \frac{1}{|\mathcal{B}|} \sum_{n \in \mathcal{B}} \nabla^2_{\boldsymbol{\theta}} \ell_n(\boldsymbol{\theta}) \,. \tag{2}$$

Design choices: To minimize computational and design overhead, we restrict the metrics to quantities that require no additional model evaluations. This means that, at training step $t \to t+1$ with mini-batches $\mathcal{B}_t, \mathcal{B}_{t+1}$ and parameters $\boldsymbol{\theta}_t, \boldsymbol{\theta}_{t+1}$, we may access information about the mini-batch losses $\mathcal{L}_{\mathcal{B}_t}(\boldsymbol{\theta}_t)$ and $\mathcal{L}_{\mathcal{B}_{t+1}}(\boldsymbol{\theta}_{t+1})$, but no cross-terms that would require additional forward passes.

Key point: $\mathcal{L}_{\mathcal{B}}(\boldsymbol{\theta}), \boldsymbol{g}_{\mathcal{B}}(\boldsymbol{\theta})$, and $\boldsymbol{H}_{\mathcal{B}}(\boldsymbol{\theta})$ are just expected values of a *distribution* over the batch. Only recently, this distribution has begun to attract attention [15] as its computation has become more accessible [8; 12]. Contemporary optimizers leverage only the *mean* gradient and neglect higher moments. One core point of our work is making extensive use of these distribution properties, trying to visualize them in various ways. This out-of-the-box support for the carefully selected and efficiently computed quantities distinguishes COCKPIT from tools like TENSORBOARD that offer visualizations as well. Leveraging these distributional quantities, we create instruments and show how they can help adapt hyperparameters (Section 2.1), analyze the loss landscape (Section 2.2), and track network dynamics (Section 2.3). Instruments can sometimes be built from already-computed information or are efficient variants of previously proposed observables. To keep the presentation concise, we highlight the instruments shown in Figure 2 and listed in Table 1. Appendix C defines them formally and contains more extensions, such as the mean GSNR [27], the early stopping [29] and CABS [4] criterion, which can all be used in COCKPIT.

Table 1: **Overview of COCKPIT quantities**. They range from cheap byproducts, to nonlinear transformations of first-order information and Hessian-based measures. Some quantities have already been proposed, others are first to be considered in this work. They are categorized into configurations *economy* $\subseteq$ *business* $\subseteq$ *full* based on their run time overhead (see Section 5 for a detailed evaluation).

Name	Short Description	Config	Pos. in Figure 2
`Alpha`	Normalized step size on a noisy quadratic interpolation between two iterates $\boldsymbol{\theta}_t$, $\boldsymbol{\theta}_{t+1}$	*economy*	top left
`Distance`	Distance from initialization $\|\boldsymbol{\theta}_t - \boldsymbol{\theta}_0\|_2$	*economy*	middle left
`UpdateSize`	Update size $\|\boldsymbol{\theta}_{t+1} - \boldsymbol{\theta}_t\|_2$	*economy*	middle left
`GradNorm`	Mini-batch gradient norm $\|\boldsymbol{g}_{\mathcal{B}}(\boldsymbol{\theta})\|_2$	*economy*	bottom left
`NormTest`	Normalized fluctuations of the residual norms $\|\boldsymbol{g}_{\mathcal{B}} - \boldsymbol{g}_n\|_2$, proposed in [9]	*economy*	top center
`InnerTest`	Normalized fluctuations of the $\boldsymbol{g}_n$'s parallel components along $\boldsymbol{g}_{\mathcal{B}}$, proposed in [7]	*economy*	top center
`OrthoTest`	Same as `InnerTest` but using the orthogonal components, proposed in [7]	*economy*	top center
`GradHist1d`	Histogram of individual gradient elements, $\{\boldsymbol{g}_n(\boldsymbol{\theta}_j)\}_{n\in\mathcal{B}}^{j=1,\dots,D}$	*economy*	middle center
`TICDiag`	Relation between (diagonal) curvature and gradient noise, inspired by [43]	*business*	bottom right
`HessTrace`	Exact or approximate Hessian trace, $\mathrm{Tr}(\boldsymbol{H}_{\mathcal{B}}(\boldsymbol{\theta}))$, inspired by [50]	*business*	middle right
`HessMaxEV`	Maximum Hessian eigenvalue, $\lambda_{\text{max}}(\boldsymbol{H}_{\mathcal{B}}(\boldsymbol{\theta}))$, inspired by [50]	*full*	top right
`GradHist2d`	Histogram of weights and individual gradient elements, $\{(\boldsymbol{\theta}_j, \boldsymbol{g}_n(\boldsymbol{\theta}_j))\}_{n\in\mathcal{B}}^{j=1,\dots,D}$	*full*	bottom center

Bug types: We distinguish three types of bugs encountered in deep learning. *Implementation bugs* are low-level software bugs that, for example, trigger syntax errors. *Training bugs* result in unnecessarily inefficient or even unsuccessful training. They can, for example, stem from erroneous data handling (see Section 3.1), the chosen model architecture (see Section 3.2), or ill-chosen hyperparameters (see Section 3.3). *Prediction bugs* describe incorrect predictions of a trained model on specific examples. Traditional debuggers are well-suited to find implementation bugs. COCKPIT focuses on efficiently identifying training bugs instead.

2.1 Adapting hyperparameters

One big challenge in deep learning is setting the hyperparameters correctly, which is currently mostly done by trial & error through parameter searches. We aim to augment this process with instruments that inform the user about the effect that the chosen parameters have on the current training process.

Alpha: Are we crossing the valley? Using individual loss and gradient observations at the start and end point of each iteration, we build a noise-informed univariate quadratic approximation along the step direction (i.e. the loss as a function of the step size), and assess to which point on this parabola our optimizer moves. We standardize this value α such that stepping to the valley-floor is assigned $\alpha = 0$, the starting point is at $\alpha = -1$ and updates to the point exactly opposite of the starting point have $\alpha = 1$ (see Appendix C.2 for a more detailed visual and mathematical description of α). Figure 1 illustrates the scenarios $\alpha = \pm 1$ and how monitoring the α-distribution (right panel) can help distinguish between two training runs with similar performance but distinct failure sources. By default, this COCKPIT instrument shows the α-distribution for the last 10 % of training and the entire training process (e.g. top left plot in Figure 2). In Section 3.3 we demonstrate empirically that, counter-intuitively, it is generally *not* a good idea to choose the step size such that α is close to zero.

Distances: Are we making progress? Another way to discern the trajectories in Figure 1 is by measuring the L_2 *distance from initialization* [31] and the *update size* [2; 16] in parameter space. Both are shown together in one COCKPIT instrument (see also middle left plot in Figure 2) and are far larger for the blue line in Figure 1. These distance metrics are also able to disentangle phases for the blue path. Using the same step size, it will continue to "jump back and forth" between the loss valley's walls but at some point cease to make progress. During this "surfing of the walls", the *distance from initialization* increases, ultimately though, it will stagnate, with the *update size* remaining non-zero, indicating diffusion. While the initial "surfing the wall"-phase benefits training (see Section 3.3), achieving stationarity may require adaptation once the optimizer reaches that diffusion.

Gradient norm: How steep is the wall? The *update size* will show that the orange trajectory is stuck. But why? Such slow-down can result from both a bad learning rate and from loss landscape plateaus. The *gradient norm* (bottom left panel in Figure 2) distinguishes these two causes.

Gradient tests: How noisy is the batch? The batch size trades off gradient accuracy versus computational cost. Recently, adaptive sampling strategies based on testing geometric constraints between mean and individual gradients have been proposed [9; 7]. The *norm*, *inner product*, and *orthogonality tests* use a standardized radius and two band widths (parallel and orthogonal to the gradient mean) that indicate how strongly individual gradients scatter around the mean. The original works use these values to adapt batch sizes. Instead, COCKPIT combines all three tests into a single gauge (top center plot of Figure 2) using the standardized noise radius and band widths for visualization. These noise signals can be used to guide batch size adaptation on- and offline, or to probe the influence of gradient alignment on training speed [37] and generalization [10; 11; 27].

2.2 Hessian properties for local loss geometry

An intuition for the local loss landscape helps in many ways. It can help diagnose whether training is stuck, to adapt the step size, and explain stability or regularization [18; 23]. The key challenge is the large number of weights: Low-dimensional projections of surfaces can behave unintuitively [30], but tracking the extreme or average behaviors may help in debugging, especially if first-order metrics fail.

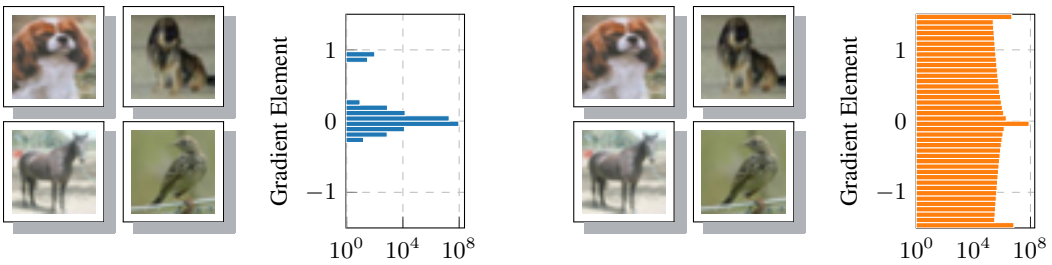
Hessian eigenvalues: A gorge or a lake? In convex optimization, the maximum Hessian eigenvalue crucially determines the appropriate step size [38]. Many works have studied the Hessian spectrum in machine learning [e.g. 17; 18; 30; 35; 36; 50]. In short: curvature matters. Established [34] and recent autodiff frameworks [12] can compute Hessian properties without requiring the full matrix. COCKPIT leverages this to provide the *Hessian's largest eigenvalue* and *trace* (top right and middle right plots in Figure 2). The former resembles the loss surface's sharpest valley and can thus hint at training instabilities [23]. The *trace* describes a notion of "average curvature", since the eigenvalues λ_i relate to it by $\sum_i \lambda_i = \mathrm{Tr}(\boldsymbol{H}_{\mathcal{B}}(\boldsymbol{\theta}))$, which might correlate with generalization [22].

TIC: How do curvature and gradient noise interact? There is an ongoing debate about curvature's link to generalization [e.g. 14; 21; 24]. The *Takeuchi Information Criterion (TIC)* [42; 43] estimates the generalization gap by a ratio between Hessian and non-central second gradient moment. It also provides intuition for changes in the objective function implied by gradient noise. Inspired by the approximations in [43], COCKPIT provides mini-batch TIC estimates (bottom right plot of Figure 2).

2.3 Visualizing internal network dynamics

Histograms are a natural visual compression of the high-dimensional $|\mathcal{B}| \times D$ individual gradient values. They give insights into the gradient *distribution* and hence offer a more detailed view of the learning signal. Together with the parameter associated to each individual gradient, the entire model status and dynamics can be visualized in a single plot and be monitored during training. This provides a more fine-grained view of training compared to tracking parameters and gradient norms [16].

Gradient and parameter histograms: What is happening in our network? COCKPIT offers a univariate *histogram of the gradient elements* $\{\boldsymbol{g}_n(\boldsymbol{\theta})_j\}_{n\in\mathcal{B}}^{j=1,\dots,D}$. Additionally, a combined *histogram of parameter-gradient pairs* $\{(\boldsymbol{\theta}_j, \boldsymbol{g}_n(\boldsymbol{\theta}_j)\}_{n\in\mathcal{B}}^{j=1,\dots,D}$ provides a two-dimensional look into the network's gradient and parameter values in a mini-batch. Section 3.1 shows an example use-case of the gradient histogram; Section 3.2 makes the case for the layer-wise variants of the instruments.



(a) Normalized Data (b) Raw Data

Figure 3: **Same inputs, different gradients; Catching data bugs with COCKPIT.** (a) *normalized* ($[0, 1]$) and (b) *raw* ($[0, 255]$) images look identical in auto-scaled front-ends like MATPLOTLIB's `imshow`. The gradient distribution on the 3C3D model, however, is crucially affected by this scaling.

3 Experiments

The diverse information provided by COCKPIT can help users and researchers in many ways, some of which, just like for a traditional debugger, only become apparent in practical use. In this section, we present a few motivating examples, selecting specific instruments and scenarios in which they are practically useful. Specifically, we show that COCKPIT can help the user discern between, and thus fix, common training bugs (Sections 3.1 and 3.2) that are otherwise hard to distinguish as they lead to the same failure: bad training. We demonstrate that COCKPIT can guide practitioners to choose efficient hyperparameters *within a single training run* (Sections 3.2 and 3.3). Finally, we highlight that COCKPIT's instruments can provide research insights about the optimization process (Section 3.3). Our empirical findings are demonstrated on problems from the DEEPOBS [39] benchmark collection.

3.1 Incorrectly scaled data

One prominent source of bugs is the data pipeline. To pick a relatively simple example: For standard optimizers to work at their usual learning rates, network inputs must be standardized (i.e. between zero and one, or have zero mean and unit variance [e.g. 5]). If the user forgets to do this, optimizer performance is likely to degrade. It can be difficult to identify the source of this problem as it does not cause obvious failures, `NaN` or `Inf` gradients, etc. We now construct a semi-realistic example, to show how using COCKPIT can help diagnose this problem upon observing slow training performance.

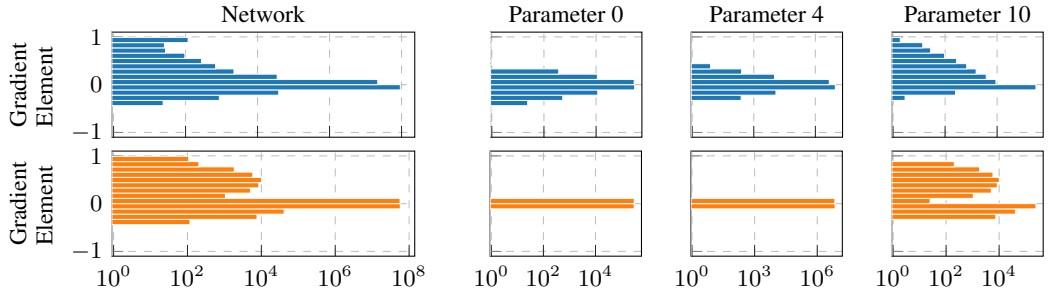
By default[3], the popular image data sets CIFAR-10/100 [25] are provided as NUMPY [20] arrays that consist of integers in the interval $[0, 255]$. This *raw* data, instead of the widely used version with floats in $[0, 1]$, changes the data scale by a factor of 255 and thus the gradients as well. Therefore, the optimizer's optimal learning rate is scaled as well. In other words, the default parameters of popular optimization methods may not work well anymore, or good hyperparameters may take extreme values. Even if the user directly inspects the training images, this may not be apparent (see Figure 3 and Figure 10 in the appendix for the same experiment with VGG16 on IMAGENET). But the gradient histogram instrument of COCKPIT, which has a deliberate default plotting range around $[-1, 1]$ to highlight such problems, immediately and prominently shows that there is an issue.

Of course, this particular data is only a placeholder for real practical data sets. While this problem may not frequently arise in the highly pre-processed, packaged CIFAR-10, it is not a rare problem for practitioners who work with their personal data sets. This is particularly likely in domains outside standard computer vision, e.g. when working with mixed-type data without obvious natural scales.

3.2 Vanishing gradients

The model architecture itself can be a source of training bugs. As before, such problems mostly arise with novel data sets, where well-working architectures are unknown. The following example shows how even small (in terms of code) architecture modifications may severely harm the training.

[3] https://www.cs.toronto.edu/~kriz/cifar.html



(a) Network Histogram (b) Layer-wise Histograms

Figure 4: **Gradient distributions of two similar architectures on the same problem**. (a) Distribution of individual gradient elements summarized over the entire network. Both seem similar. (b) Layer-wise histograms for a subset of layers. Parameter 0 is the layer closest to the network's input, parameter 10 closest to its output. Only the layer-wise view reveals that there are several degenerated gradient distributions for the orange network making training unnecessary hard.

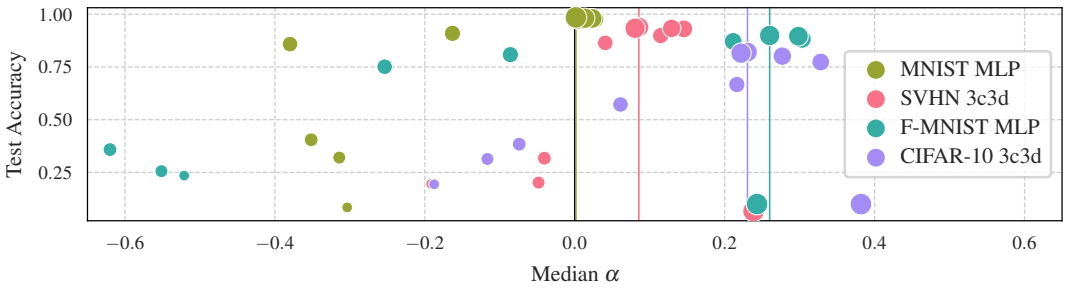


Figure 5: **Test accuracy as a function of standardized step size** α. For four DEEPOBS problems (see Appendix E), final test accuracy is shown versus the median α-value over the entire training. Marker size indicates the magnitude of the raw learning rate, marker color identifies tasks (see legend). For each problem, the best-performing setting is highlighted by a vertical colored line.

Figure 4a shows the distribution of gradient values of two different network architectures in blue and orange. Although the blue model trains considerably better than the orange one, their gradient distributions look quite similar. The difference becomes evident when inspecting the histogram *layer-wise*. We can see that multiple layers have a degenerated gradient distribution with many elements being practically zero (see Figure 4b, bottom row). Since the fully connected layers close to the output have far more parameters (a typical pattern of convolutional networks), they dominate the network-wide histogram. This obscures that a major part of the model is effectively unable to train.

Both the blue and orange networks follow DEEPOBS's 3C3D architecture. The only difference is the non-linearity: The blue network uses standard ReLU activations, while the orange one has sigmoid activations. Here, the layer-wise histogram instrument of COCKPIT highlights which part of the architecture makes training unnecessarily hard. Accessing information layer-wise is also essential due to the strong overparameterization in deep models where training can happen in small subspaces [19]. Once again, this is hard to do with common monitoring tools, such as the loss curve.

3.3 Tuning learning rates

Once the architecture is defined, the optimizer's learning rate is the most important hyperparameter to tune. Getting it right requires extensive hyperparameter searches at high resource costs. COCKPIT's instruments can provide intuition and information to streamline this process: In contrast to the raw learning rate, the curvature-standardized step size α-quantity (see Section 2.1) has a natural scale.

Across multiple optimization problems, we observe, perhaps surprisingly, that the best runs and indeed all good runs have a median $\alpha > 0$ (Figure 5). This illustrates a fundamental difference between stochastic optimization, as is typical for machine learning, and classic deterministic optimization. Instead of locally stepping "to the valley floor" (optimal in the deterministic case), stochastic

optimizers should *overshoot* the valley somewhat. This need to "surf the walls" has been hypothesized before [e.g. 47; 49] as a property of neural network training. Frequently, learning rates are adapted during training, which fits with our observation about positive α-values: "Overshooting" allows fast early progression towards areas of lower loss, but it does not yield convergence in the end. Real-time visualizations of the training state, as offered by COCKPIT, can augment these fine-tuning processes.

Figure 5 also indicates a major challenge preventing simple automated tuning solutions: The optimal α-value is problem-dependent, and simpler problems, such as a multi-layer perceptron (MLP) on MNIST [26], behave much more similar to classic optimization problems. Algorithmic research on small problems can thus produce misleading conclusions. The figure also shows that the α-gauge is not sufficient by itself: extreme overshooting with a too-large learning rate leads to poor performance, which however can be prevented by taking additional instruments into account. This makes the case for the cockpit metaphor of increasing interpretability from several instruments in conjunction. By combining the α-instrument with other gauges that capture the local geometry or network dynamics, the user can better identify good choices of the learning rate and other hyperparameters.

4 Showcase

Having introduced the tool, we can now return to Figure 2 for a closer look. The figure shows a snapshot from training the ALL-CNN-C [41] on CIFAR-100 using SGD with a cyclic learning rate schedule (see bottom left panel). Diagonal curvature instruments are configured to use an MC approximation in order to reduce the run time (here, $C = 100$, compare Section 5).

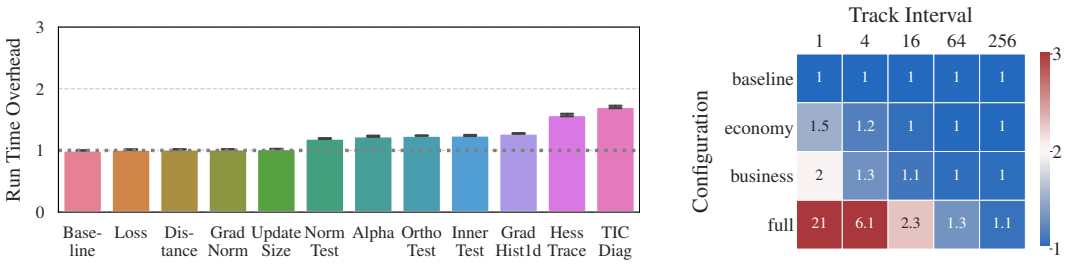
A glance at all panels shows that the learning rate schedule is reflected in the metrics. However, the instruments also provide insights into the early phase of training (first ~ 100 iterations), where the learning rate is still unaffected by the schedule: There, the loss plateaus and the optimizer takes relatively small steps (compared to later, as can be seen in the small gradient norms, and small distance from initialization). Based on these low-cost instruments, one may thus at first suspect that training was poorly initialized; but training indeed succeeds after iteration 100! Viewing COCKPIT entirely though, it becomes clear that optimization in these first steps is not stuck at all: While loss, gradient norms, and distance in parameter space remain almost constant, curvature changes, which expresses itself in a clear downward trend of the maximum Hessian eigenvalue (top right panel).

The importance of early training phases has recently been hypothesized [16], suggesting a logarithmic timeline. Not only does our showcase support this hypothesis, but it also provides an explanation from the curvature-based metrics, which in this particular case are the only meaningful feedback in the first few training steps. It also suggests monitoring training at log-spaced intervals. COCKPIT provides the flexibility to do so, indeed, Figure 2 has been created with log-scheduled tracking events.

As a final note, we recognize that the approach taken here promotes an amount of *manual* work (monitoring metrics, deliberately intervening, etc.) that may seem ironic and at odds with the paradigm of automation that is at the heart of machine learning. However, we argue that this might be what is needed at this point in the evolution of the field. Deep learning has been driven notably by scaling compute resources [44], and fully automated, one-shot training may still be some way out. To develop better training methods, researchers, not just users, need *algorithmic* interpretability and explainability: direct insights and intuition about the processes taking place "inside" neural nets. To highlight how COCKPIT might provide this, we contrast in Appendix F the COCKPIT view of two convex DEEPOBS problems: a noisy quadratic and logistic regression on MNIST. In both cases, the instruments behave differently compared to the deep learning problem in Figure 2. In particular, the gradient norm increases (left column, bottom panel) during training, and individual gradients become less scattered (center column, top panel). This is diametrically opposed to the convex problems and shows that deep learning differs even qualitatively from well-understood optimization problems.

5 Benchmark

Section 3 made a case for COCKPIT as an effective debugging and tuning tool. To make the library useful in practice, it must also have limited computational cost. We now show that it is possible to compute all quantities at reasonable overhead. The user can control the absolute cost along two dimensions, by reducing the number of instruments, or by reducing their update frequency.



(a) Overhead COCKPIT instruments

(b) Overhead COCKPIT configurations

Figure 6: **Run time overhead for individual COCKPIT instruments and configurations** as shown on CIFAR-10 3C3D on a GPU. *Left:* The run time overheads for individual instruments are shown as multiples of the *baseline* (no tracking). Most instruments add little overhead. This plot shows the overhead in one iteration, determined by averaging over multiple iterations and random seeds. *Right:* Overhead for different COCKPIT configurations. Adjusting the tracking interval and re-using the computation shared by multiple instruments can make the overhead orders of magnitude smaller. Blue fields mark settings that allow tracking without doubling the training time.

All benchmark results show SGD without momentum. COCKPIT's quantities, however, work for generic optimizers and can mostly be used identically without increased costs. One current exception is `Alpha` which can be computed more efficiently given the optimizer's update rule.[4]

Complexity analysis: Computing more information adds computational overhead, of course. However, recent work [12] has shown that first-order information, like distributional statistics on the batch gradients, can be computed on top of the mean gradient at little extra cost. Similar savings apply for most quantities in Table 1, as they are (non-)linear transformations of individual gradients. A subset of COCKPIT's quantities also uses second-order information from the Hessian diagonal. For ReLU networks on a classification task with C classes, the additional work is proportional to C gradient backpropagations (i.e. $C = 10$ for CIFAR-10, $C = 100$ for CIFAR-100). Parallel processing can, to some extent, process these extra backpropagations in parallel without significant overhead. If this is no longer possible, we can fall back to a Monte Carlo (MC) sampling approximation, which reduces the number of extra backprop passes to the number of samples (1 by default).[5]

While parallelization is possible for the gradient instruments, computing the maximum Hessian eigenvalue is inherently sequential. Similar to Yao et al. [50], we use matrix-free Hessian-vector products by automatic differentiation [34], where each product's costs are proportional to one gradient computation. Regardless of the underlying iterative eigensolver, multiple such products must be queried to compute the spectral norm (the required number depends on the spectral gap to the second-largest eigenvalue).

Run time benchmark: Figure 6a shows the wall-clock computational overhead for individual instruments (details in Appendix E).[6] As expected, byproducts are virtually free, and quantities that rely solely on first-order information add little overhead (at most roughly 25 % on this problem). Thanks to parallelization, the ten extra backward passes required for Hessian quantities reduce to less than 100 % overhead. Individual overheads also do not simply add up when multiple quantities are tracked, because quantities relying on the same information share computations.

To allow a rough cost control, COCKPIT currently offers three configurations, called *economy*, *business*, and *full*, in increasing order of cost (cf. Table 1). As a basic guideline, we consider a factor of two to be an acceptable limit for the increase in training time and benchmark the configurations'

[4] This is currently implemented for vanilla SGD. Otherwise, COCKPIT falls back to a less efficient scheme.

[5] An MC-sampled approximation of the Hessian/generalized Gauss-Newton has been used in Figure 2 to reduce the prohibitively large number of extra backprops on CIFAR-100 ($C = 100$).

[6] To improve readability, we exclude `HessMaxEV` here, because its overhead is large compared to other quantities. Surprisingly, we also observed significant cost for the 2D histogram on GPU. It is caused by an implementation bottleneck for histogram shapes observed in deep models. We thus also omit `GradHist2d` here, as we expect it to be eliminated with future implementations (see Appendix E.2 for a detailed analysis and further benchmarks). Both quantities, however, are part of the benchmark shown in Figure 6b.

run times for different tracking intervals. Figure 6b shows a run time matrix for the CIFAR-10 3C3D problem, where settings that meet this limit are set in blue (more problems including IMAGENET are shown in Appendix E). Speedups due to shared computations are easy to read off: Summing all the individual overheads shown in Figure 6a would result in a total overhead larger than 200 %, while the joint overhead (*business*) reduces to 140 %. The *economy* configuration can easily be tracked at every step of this problem and stay well below our threshold of doubling the execution time. COCKPIT's full view, shown in Figure 2, can be updated every 64-th iteration without a major increase in training time (this corresponds to about five updates per epoch). Finally, tracking any configuration about once per epoch – which is common in practice – adds overhead close to zero (rightmost column).

This good performance is largely due to the efficiency of the BACKPACK package [12], which we leverage with custom and optimized modification, that compacts information layer-wise and then discards unneeded buffers. Using layer-wise information (Section 3.2) scales better to large networks, where storing the entire model's individual gradients all at once becomes increasingly expensive (see Appendix E). To the best of our knowledge, many of the quantities in Table 1, especially those relying on individual gradients, have only been explored on rather small problems. With COCKPIT they can now be accessed at a reasonable rate for deep learning models outside the toy problem category.

6 Conclusion

Contemporary machine learning, in particular deep learning, remains a craft and an art. High dimensionality, stochasticity, and non-convexity require constant tracking and tuning, often resulting in a painful process of trial and error. When things fail, popular performance measures, like the training loss, do not provide enough information by themselves. These metrics only tell *whether* the model is learning, but not *why*. Alternatively, traditional debugging tools can provide access to individual weights and data. However, in models whose power only arises from possessing myriad weights, this approach is hopeless, like looking for the proverbial needle in a haystack.

To mitigate this, we proposed COCKPIT, a practical visual debugging tool for deep learning. It offers instruments to monitor the network's internal dynamics during training, in real-time. In its presentation, we focused on two crucial factors affecting user experience: Firstly, such a debugger must provide meaningful insights. To demonstrate COCKPIT's utility, we showed how it can identify bugs where traditional tools fail. Secondly, it must come at a feasible computational cost. Although COCKPIT uses rich second-order information, efficient computation keeps the necessary run time overhead cheap. The open-source PYTORCH package can be added to many existing training loops.

Obviously, such a tool is never complete. Just like there is no perfect universal debugger, the list of current instruments is naturally incomplete. Further practical experience with the tool, for example in the form of a future larger user study, could provide additional evidence for its utility. However, our analysis shows that COCKPIT provides useful tools and extracts valuable information presently not accessible to the user. We believe that this improves algorithmic interpretability – helping practitioners understand how to make their models work – but may also inspire new research. The code is designed flexibly, deliberately separating the computation and visualization. New instruments can be added easily and also be shown by the user's preferred visualization tool, e.g. TENSORBOARD. Of course, instead of just showing the data, the same information can be used by novel algorithms directly, side-stepping the human in the loop.

Acknowledgments and Disclosure of Funding

The authors gratefully acknowledge financial support by the European Research Council through ERC StG Action 757275 / PANAMA; the DFG Cluster of Excellence "Machine Learning - New Perspectives for Science", EXC 2064/1, project number 390727645; the German Federal Ministry of Education and Research (BMBF) through the Tübingen AI Center (FKZ: 01IS18039A); and funds from the Cyber Valley Initiative of the Ministry for Science, Research and Arts of the State of Baden-Württemberg. Moreover, the authors thank the International Max Planck Research School for Intelligent Systems (IMPRS-IS) for supporting Felix Dangel and Frank Schneider. Further, we are grateful to Agustinus Kristiadi, Alexandra Gessner, Christian Fröhlich, Filip de Roos, Jonathan Wenger, Julia Grosse, Lukas Tatzel, Marius Hobbhahn, and Nicholas Krämer for providing feedback to the manuscript.

References

[1] Abadi, M., Agarwal, A., Barham, P., Brevdo, E., Chen, Z., Citro, C., Corrado, G. S., Davis, A., Dean, J., Devin, M., Ghemawat, S., Goodfellow, I., Harp, A., Irving, G., Isard, M., Jia, Y., Jozefowicz, R., Kaiser, L., Kudlur, M., Levenberg, J., Mané, D., Monga, R., Moore, S., Murray, D., Olah, C., Schuster, M., Shlens, J., Steiner, B., Sutskever, I., Talwar, K., Tucker, P., Vanhoucke, V., Vasudevan, V., Viégas, F., Vinyals, O., Warden, P., Wattenberg, M., Wicke, M., Yu, Y., and Zheng, X. TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems, 2015. URL https://www.tensorflow.org/. 2

[2] Agrawal, A. M., Tendle, A., Sikka, H., Singh, S., and Kayid, A. Investigating Learning in Deep Neural Networks using Layer-Wise Weight Change. *arXiv preprint: 2011.06735*, 2020. 5

[3] Bahamou, A. and Goldfarb, D. A Dynamic Sampling Adaptive-SGD Method for Machine Learning. *arXiv preprint: 1912.13357*, 2019. 24

[4] Balles, L., Romero, J., and Hennig, P. Coupling adaptive batch sizes with learning rates. In *Conference on Uncertainty in Artificial Intelligence (UAI)*, 2017. 3, 17, 20, 21

[5] Bengio, Y. Practical recommendations for gradient-based training of deep architectures. *Neural Networks: Tricks of the Trade (2nd ed.)*, pp. 437–478, 2012. 6

[6] Biewald, L. Experiment Tracking with Weights and Biases, 2020. URL https://www.wandb.com/. Software available from wandb.com. 2

[7] Bollapragada, R., Byrd, R. H., and Nocedal, J. Adaptive Sampling Strategies for Stochastic Optimization. *SIAM Journal on Optimization*, 28:3312–3343, 2017. 4, 5, 17, 22, 23, 24

[8] Bradbury, J., Frostig, R., Hawkins, P., Johnson, M. J., Leary, C., Maclaurin, D., and Wanderman-Milne, S. JAX: composable transformations of Python+NumPy programs, 2018. URL http://github.com/google/jax. 3

[9] Byrd, R. H., Chin, G. M., Nocedal, J., and Wu, Y. Sample Size Selection in Optimization Methods for Machine Learning. *Math. Program.*, 134(1):127–155, 2012. ISSN 0025-5610. 4, 5, 17, 22

[10] Chatterjee, S. Coherent Gradients: An Approach to Understanding Generalization in Gradient Descent-based Optimization. In *International Conference on Learning Representations (ICLR)*, 2020. 5

[11] Chatterjee, S. and Zielinski, P. Making Coherence Out of Nothing At All: Measuring the Evolution of Gradient Alignment. *arXiv preprint: 2008.01217*, 2020. 5

[12] Dangel, F., Kunstner, F., and Hennig, P. BackPACK: Packing more into Backprop. In *International Conference on Learning Representations (ICLR)*, 2020. 2, 3, 5, 9, 10, 14

[13] Deng, J., Dong, W., Socher, R., Li, L.-J., Li, K., and Fei-Fei, L. ImageNet: A Large-Scale Hierarchical Image Database. In *IEEE conference on computer vision and pattern recognition*, pp. 248–255. IEEE, 2009. 26

[14] Dinh, L., Pascanu, R., Bengio, S., and Bengio, Y. Sharp Minima Can Generalize For Deep Nets. In *International Conference on Machine Learning (ICML)*, 2017. 5

[15] Faghri, F., Duvenaud, D., Fleet, D. J., and Ba, J. A Study of Gradient Variance in Deep Learning. *arXiv preprint: 2007.04532*, 2020. 3

[16] Frankle, J., Schwab, D. J., and Morcos, A. S. The Early Phase of Neural Network Training. In *International Conference on Learning Representations (ICLR)*, 2020. 5, 8

[17] Ghorbani, B., Krishnan, S., and Xiao, Y. An Investigation into Neural Net Optimization via Hessian Eigenvalue Density. In *International Conference on Machine Learning (ICML)*, 2019. 5

[18] Ginsburg, B. On regularization of gradient descent, layer imbalance and flat minima. *arXiv preprint: 2007.09286*, 2020. 5, 26, 27

[19] Gur-Ari, G., Roberts, D. A., and Dyer, E. Gradient Descent Happens in a Tiny Subspace. *arXiv preprint: 1812.04754*, 2018. 7

[20] Harris, C. R., Millman, K. J., van der Walt, S. J., Gommers, R., Virtanen, P., Cournapeau, D., Wieser, E., Taylor, J., Berg, S., Smith, N. J., Kern, R., Picus, M., Hoyer, S., van Kerkwijk, M. H., Brett, M., Haldane, A., del Río, J. F., Wiebe, M., Peterson, P., Gérard-Marchant, P., Sheppard, K., Reddy, T., Weckesser, W., Abbasi, H., Gohlke, C., and Oliphant, T. E. Array programming with NumPy. *Nature*, 585(7825):357–362, 2020. 6

[21] Hochreiter, S. and Schmidhuber, J. Flat Minima. *Neural Comput.*, 9(1):1–42, 1997. 5, 25

[22] Jastrzebski, S., Arpit, D., Astrand, O., Kerg, G., Wang, H., Xiong, C., Socher, R., Cho, K., and Geras, K. Catastrophic Fisher Explosion: Early Phase Fisher Matrix Impacts Generalization. *arXiv preprint:* `2012.14193`, 2020. 5

[23] Jastrzebski, S., Szymczak, M., Fort, S., Arpit, D., Tabor, J., Cho, K., and Geras, K. The Break-Even Point on Optimization Trajectories of Deep Neural Networks. In *International Conference on Learning Representations (ICLR)*, 2020. 5

[24] Keskar, N. S., Mudigere, D., Nocedal, J., Smelyanskiy, M., and Tang, P. T. P. On Large-Batch Training for Deep Learning: Generalization Gap and Sharp Minima. In *International Conference on Learning Representations (ICLR)*, 2017. 5

[25] Krizhevsky, A. Learning multiple layers of features from tiny images. Technical report, 2009. 6, 28

[26] Lecun, Y., Bottou, L., Bengio, Y., and Haffner, P. Gradient-based learning applied to document recognition. In *Proceedings of the IEEE*, volume 86, pp. 2278–2324, 1998. 8, 15, 28

[27] Liu, J., Bai, Y., Jiang, G., Chen, T., and Wang, H. Understanding Why Neural Networks Generalize Well Through GSNR of Parameters. In *International Conference on Learning Representations (ICLR)*, 2020. 3, 5, 17, 26

[28] Mahsereci, M. and Hennig, P. Probabilistic Line Searches for Stochastic Optimization. *Journal of Machine Learning Research*, 18(119):1–59, 2017. 20

[29] Mahsereci, M., Balles, L., Lassner, C., and Hennig, P. Early Stopping without a Validation Set. *arXiv preprint:* `1703.09580`, 2017. 3, 17, 21

[30] Mulayoff, R. and Michaeli, T. Unique Properties of Flat Minima in Deep Networks. In *International Conference on Machine Learning (ICML)*, 2020. 5, 26, 27

[31] Nagarajan, V. and Kolter, J. Z. Generalization in Deep Networks: The Role of Distance from Initialization. *arXiv preprint:* `1901.01672`, 2019. 5

[32] Netzer, Y., Wang, T., Coates, A., Bissacco, A., Wu, B., and Ng, A. Y. Reading Digits in Natural Images with Unsupervised Feature Learning. NIPS Workshop on Deep Learning and Unsupervised Feature Learning, 2011. 28

[33] Paszke, A., Gross, S., Massa, F., Lerer, A., Bradbury, J., Chanan, G., Killeen, T., Lin, Z., Gimelshein, N., Antiga, L., Desmaison, A., Kopf, A., Yang, E., DeVito, Z., Raison, M., Tejani, A., Chilamkurthy, S., Steiner, B., Fang, L., Bai, J., and Chintala, S. PyTorch: An Imperative Style, High-Performance Deep Learning Library. In *Neural Information Processing Systems (NeurIPS)*, 2019. 2

[34] Pearlmutter, B. A. Fast Exact Multiplication by the Hessian. *Neural Computation*, 6(1):147–160, 1994. 5, 9, 24, 25

[35] Sagun, L., Bottou, L., and LeCun, Y. Eigenvalues of the Hessian in Deep Learning: Singularity and Beyond. *arXiv preprint:* `1611.07476`, 2017. 5

[36] Sagun, L., Evci, U., Guney, V. U., Dauphin, Y., and Bottou, L. Empirical Analysis of the Hessian of Over-Parametrized Neural Networks. *arXiv preprint:* `1706.04454`, 2018. 5

[37] Sankararaman, K. A., De, S., Xu, Z., Huang, W. R., and Goldstein, T. The Impact of Neural Network Overparameterization on Gradient Confusion and Stochastic Gradient Descent. In *International Conference on Machine Learning (ICML)*, 2020. 5

[38] Schmidt, M. Convergence rate of stochastic gradient with constant step size. 2014. URL `https://www.cs.ubc.ca/~schmidtm/Documents/2014_Notes_ConstantStepSG.pdf`. 5, 25

[39] Schneider, F., Balles, L., and Hennig, P. DeepOBS: A Deep Learning Optimizer Benchmark Suite. In *International Conference on Learning Representations (ICLR)*, 2019. 6, 14, 28

[40] Simonyan, K. and Zisserman, A. Very Deep Convolutional Networks for Large-Scale Image Recognition. In *3rd International Conference on Learning Representations, ICLR*, 2015. 26

[41] Springenberg, J. T., Dosovitskiy, A., Brox, T., and Riedmiller, M. Striving for Simplicity: The All Convolutional Net. *arXiv preprint:* `1412.6806`, 2015. 3, 8, 28

[42] Takeuchi, K. The distribution of information statistics and the criterion of goodness of fit of models. *Mathematical Science*, 153:12–18, 1976. 5, 25

[43] Thomas, V., Pedregosa, F., van Merriënboer, B., Manzagol, P.-A., Bengio, Y., and Roux, N. L. On the interplay between noise and curvature and its effect on optimization and generalization. In *International Conference on Artificial Intelligence and Statistics (AISTATS)*, 2020. 4, 5, 17, 25

[44] Thompson, N. C., Greenewald, K., Lee, K., and Manso, G. F. The Computational Limits of Deep Learning. *arXiv preprint: 2007.05558*, 2020. 8

[45] Vaswani, S., Mishkin, A., Laradji, I., Schmidt, M., Gidel, G., and Lacoste-Julien, S. Painless Stochastic Gradient: Interpolation, Line-Search, and Convergence Rates. In *Neural Information Processing Systems (NeurIPS)*, 2019. 20

[46] Warsa, J., Wareing, T., Morel, J., Mcghee, J., and Lehoucq, R. Krylov Subspace Iterations for Deterministic k-Eigenvalue Calculations. *Nuclear Science and Engineering - NUCL SCI ENG*, 147, 05 2004. doi: 10.13182/NSE04-1. 24

[47] Wu, Y., Ren, M., Liao, R., and Grosse, R. B. Understanding short-horizon bias in stochastic meta-optimization. *International Conference on Learning Representations (ICLR 2018)*, 2018. 8

[48] Xiao, H., Rasul, K., and Vollgraf, R. Fashion-MNIST: a Novel Image Dataset for Benchmarking Machine Learning Algorithms. *arXiv preprint: 1708.07747*, 2017. 28

[49] Xing, C., Arpit, D., Tsirigotis, C., and Bengio, Y. A Walk with SGD. *arXiv preprint: 1802.08770*, 2018. 8, 20

[50] Yao, Z., Gholami, A., Keutzer, K., and Mahoney, M. W. PyHessian: Neural Networks Through the Lens of the Hessian. In *IEEE International Conference on Big Data (Big Data)*, 2020. 4, 5, 9, 17, 24, 25

COCKPIT: A Practical Debugging Tool for the Training of Deep Neural Networks

Supplementary Material

Checklist

1. For all authors...
   (a) Do the main claims made in the abstract and introduction accurately reflect the paper's contributions and scope? [Yes]
   (b) Did you describe the limitations of your work? [Yes] In Section 5 we detail the additional costs of each instrument, also showing that two of them come with a large overhead (more details and how it can be mitigated in Appendix E.2). Section 6 acknowledges that while we believe this tool to be an important step, it is understandably incomplete.
   (c) Did you discuss any potential negative societal impacts of your work? [N/A] The paper proposes an algorithmic debugging tool that is of a foundational nature. Ethical questions are thus not sufficiently prominent in this work to warrant a dedicated discussion section. In general, we believe, this work will have an overall positive impact as it can help shed light into the black-box that is deep learning. As a longer-term side-effect of this work, this could help the explainability and interpretability of neural networks.
   (d) Have you read the ethics review guidelines and ensured that your paper conforms to them? [Yes]
2. If you are including theoretical results...
   (a) Did you state the full set of assumptions of all theoretical results? [N/A]
   (b) Did you include complete proofs of all theoretical results? [N/A]
3. If you ran experiments...
   (a) Did you include the code, data, and instructions needed to reproduce the main experimental results (either in the supplemental material or as a URL)? [Yes] All experimental results, as well as the complete code base to reproduce them can be found at the linked GitHub repository at `https://github.com/fsschneider/cockpit-experiments`. The COCKPIT package is available open source at `https://github.com/f-dangel/cockpit`.
   (b) Did you specify all the training details (e.g., data splits, hyperparameters, how they were chosen)? [Yes] All training details are given at `https://github.com/fsschneider/cockpit-experiments`. If not stated otherwise, we use the defaults suggested by the DEEPOBS benchmark suite which are summarized in Appendix E.
   (c) Did you report error bars (e.g., with respect to the random seed after running experiments multiple times)? [Yes] whenever applicable, we report error bars (e.g. left subplot of Figure 6 shows error bars from averages over ten random seeds).
   (d) Did you include the total amount of compute and the type of resources used (e.g., type of GPUs, internal cluster, or cloud provider)? [Yes] listed in Appendix E
4. If you are using existing assets (e.g., code, data, models) or curating/releasing new assets...
   (a) If your work uses existing assets, did you cite the creators? [Yes] We make extensive use of both the BACKPACK [12] and the DEEPOBS [39] packages. Both are cited throughout the text. Whenever applicable, we also cited the used data sets and models. We explicitly mention the authors of the used histogram code in Appendix E.3 and have asked them for permissions.

(b) Did you mention the license of the assets? [Yes] The library has been released open source under the MIT License. COCKPIT's GitHub repository includes the full license.
(c) Did you include any new assets either in the supplemental material or as a URL? [Yes] The full library can be found at `https://github.com/f-dangel/cockpit`.
(d) Did you discuss whether and how consent was obtained from people whose data you're using/curating? [N/A]
(e) Did you discuss whether the data you are using/curating contains personally identifiable information or offensive content? [N/A] No new data was collected with our experiments relying on established and published data sets such as MNIST [26].

5. If you used crowdsourcing or conducted research with human subjects...
   (a) Did you include the full text of instructions given to participants and screenshots, if applicable? [N/A]
   (b) Did you describe any potential participant risks, with links to Institutional Review Board (IRB) approvals, if applicable? [N/A]
   (c) Did you include the estimated hourly wage paid to participants and the total amount spent on participant compensation? [N/A]

A Code example

One design principle of COCKPIT is its easy integration with conventional PYTORCH training loops. Figure 7 shows a working example of a standard training loop with COCKPIT integration. More examples and tutorials are described in COCKPIT's documentation. COCKPIT's syntax is inspired by BACKPACK: It can be used interchangeably with the library responsible for most back-end computations. Changes to the code are straightforward:

- **Importing** (*Lines 5, 7* and *8*): Besides importing COCKPIT we also need to import BACKPACK which is required for extending (parts of) the model (see next step).
- **Extending** (*Lines 11* and *12*): When defining the model and the loss function, we need to *extend* both of them using BACKPACK. This is as trivial as wrapping them in the `extend()` function provided by BACKPACK and lets BACKPACK know that additional quantities (such as the individual gradients) should be computed for them. Note, that while applying BACKPACK is easy, it currently does not support all possible model architectures and layer types. Specifically, *batch norm* layers are not supported since using them results in ill-defined individual gradients.
- **Individual losses** (*Line 13*): For the `Alpha` quantity, COCKPIT also requires the individual loss values (to estimate the variance of the loss estimate). This can be computed cheaply but is not usually part of a conventional training loop. Creating this loss is done analogously to creating any other loss, with the only exception of setting `reduction="none"`. Since we don't differentiate this loss, we don't need to extend it.
- **Cockpit configuration** (*Line 16* and *17*): Initializing the COCKPIT requires passing them (extended) model parameters as well as a list of quantities that should be tracked. Table 1 provides an overview of all possible quantities. In this example, we use one of the predefined configurations offered by COCKPIT. Separately, we initialize the plotting part of COCKPIT. We deliberately detached the visualization from the tracking to allow greater flexibility.
- **Quantity computation** (*Line 27* and *38*): Performing the training is very similar to a regular training loop, with the only difference being that the backward pass should be surrounded by the COCKPIT context (`with cockpit():`). Additionally to the `global_step` we also pass a few additional information to the COCKPIT that are computed anyway and can be re-used by the COCKPIT, such as the batch size, the individual losses, or the optimizer itself. After the backward pass (when the context is left) all COCKPIT quantities are automatically computed.
- **Logging and visualizing** (*Line 46* and *47*): At any point during the training, here we do it at the end, we can write all quantities to a log file. We can use this log file, or alternatively the COCKPIT directly, to visualize all quantities which would result in a status screen similar to Figure 2.

```python
"""Example: Training Loop using Cockpit."""

import torch
from _utils_examples import cnn, fmnist_data, get_logpath
from backpack import extend
from cockpit import Cockpit, CockpitPlotter
from cockpit.utils.configuration import configuration as config

fmnist_data = fmnist_data()
model = extend(cnn())
loss_fn = extend(torch.nn.CrossEntropyLoss(reduction="mean"))
losses_fn = torch.nn.CrossEntropyLoss(reduction="none")
opt = torch.optim.SGD(model.parameters(), lr=1e-2)

cockpit = Cockpit(model.parameters(), quantities=config("full"))
plotter = CockpitPlotter()

max_steps, global_step = 50, 0
for inputs, labels in iter(fmnist_data):
    opt.zero_grad()

    outputs = model(inputs)
    loss = loss_fn(outputs, labels)
    losses = losses_fn(outputs, labels)

    with cockpit(
        global_step,
        info={
            "batch_size": inputs.shape[0],
            "individual_losses": losses,
            "loss": loss,
            "optimizer": opt,
        },
    ):
        loss.backward(
            create_graph=cockpit.create_graph(global_step),
        )

    opt.step()
    global_step += 1

    if global_step >= max_steps:
        break

cockpit.write(get_logpath())
plotter.plot(get_logpath())
```

Figure 7: **Complete training loop with COCKPIT** in PYTORCH. Line changes are highlighted in light orange ( ).

B COCKPIT instruments overview

Table 2 lists all quantities available in the first public release of COCKPIT. If necessary, we provide references to their mathematical definition. This table contains additional quantities, compared to Table 1 in the main text. To improve the presentation of this work, we decided to not describe every quantity available in COCKPIT in the main part and instead focus on the investigated metrics. Custom quantities can be added easily without having to understand the inner-workings.

Table 2: **Overview of all COCKPIT quantities** with a short description and, if necessary, a reference to mathematical definition.

Name	Description	Math
`Loss`	Mini-batch training loss at current iteration, $\mathcal{L}_{\mathcal{B}}(\boldsymbol{\theta})$	(1)
`Parameters`	Parameter values $\boldsymbol{\theta}_t$ at the current iteration	-
`Distance`	L_2 distance from initialization $\|\boldsymbol{\theta}_t - \boldsymbol{\theta}_0\|_2$	-
`UpdateSize`	Update size of the current iteration $\|\boldsymbol{\theta}_{t+1} - \boldsymbol{\theta}_t\|_2$	
`GradNorm`	Mini-batch gradient norm $\|\boldsymbol{g}_{\mathcal{B}}(\boldsymbol{\theta})\|_2$	-
`Time`	Time of the current iteration (e.g. used in benchmark of Appendix E)	-
`Alpha`	Normalized step on a noisy quadratic interpolation between two iterates $\boldsymbol{\theta}_t, \boldsymbol{\theta}_{t+1}$	(9)
`CABS`	Adaptive batch size for SGD, optimizes expected objective gain per cost, adapted from [4]	(11)
`EarlyStopping`	Evidence-based early stopping criterion for SGD, proposed in [29]	(13d)
`GradHist1d`	Histogram of individual gradient elements, $\{\boldsymbol{g}_n(\boldsymbol{\theta}_j)\}_{n\in\mathcal{B}}^{j=1,\dots,D}$	(14)
`GradHist2d`	Histogram of weights and individual gradient elements, $\{(\boldsymbol{\theta}_j, \boldsymbol{g}_n(\boldsymbol{\theta}_j))\}_{n\in\mathcal{B}}^{j=1,\dots,D}$	(15)
`NormTest`	Normalized fluctuations of the residual norms $\|\boldsymbol{g}_{\mathcal{B}} - \boldsymbol{g}_n\|$, proposed in [9]	(18c)
`InnerTest`	Normalized fluctuations of $\boldsymbol{g}_n$'s parallel components along $\boldsymbol{g}_{\mathcal{B}}$, proposed in [7]	(21c)
`OrthoTest`	Normalized fluctuations of $\boldsymbol{g}_n$'s orthogonal components along $\boldsymbol{g}_{\mathcal{B}}$, proposed in [7]	(24b)
`HessMaxEV`	Maximum Hessian eigenvalue, $\lambda_{\text{max}}(\boldsymbol{H}_{\mathcal{B}}(\boldsymbol{\theta}))$, inspired by [50]	(25)
`HessTrace`	Exact or approximate Hessian trace, $\text{Tr}(\boldsymbol{H}_{\mathcal{B}}(\boldsymbol{\theta}))$, inspired by [50]	-
`TICDiag`	Relation between (diagonal) curvature and gradient noise, inspired by [43]	(28)
`TICTrace`	Relation between curvature and gradient noise trace, inspired by [43]	(27)
`MeanGSNR`	Average gradient signal-to-noise-ratio (GSNR), inspired by [27]	(30b)

C Mathematical details

In this section, we want to provide the mathematical background for each instrument described in Table 2. This complements the more informal description presented in Section 2 in the main text, which focused more on the expressiveness of the individual quantities. We will start by setting up the necessary notation in addition to the one introduced in Section 2.

C.1 Additional notation

Population properties: The population risk $\mathcal{L}_P(\boldsymbol{\theta}) \in \mathbb{R}$ and its variance $\Lambda(\boldsymbol{\theta}) \in \mathbb{R}$ are given by

$$\mathcal{L}_P(\boldsymbol{\theta}) = \mathbb{E}_{(\boldsymbol{x},\boldsymbol{y})\sim P}\left[\ell(f(\boldsymbol{\theta},\boldsymbol{x}),\boldsymbol{y})\right] = \int \ell(f(\boldsymbol{\theta},\boldsymbol{x}),\boldsymbol{y}) P(\boldsymbol{x},\boldsymbol{y})\, d\boldsymbol{x}\, d\boldsymbol{y}\,, \tag{3a}$$

$$\Lambda_P(\boldsymbol{\theta}) = \mathrm{Var}_{(\boldsymbol{x},\boldsymbol{y})\sim P}\left[\ell(f(\boldsymbol{\theta},\boldsymbol{x}),\boldsymbol{y})\right] = \int \left(\ell(f(\boldsymbol{\theta},\boldsymbol{x}),\boldsymbol{y}) - \mathcal{L}_P(\boldsymbol{\theta})\right)^2 P(\boldsymbol{x},\boldsymbol{y})\, d\boldsymbol{x}\, d\boldsymbol{y}\,. \tag{3b}$$

The population gradient $\boldsymbol{g}_P(\boldsymbol{\theta}) \in \mathbb{R}^D$ and its variance $\boldsymbol{\Sigma}_P(\boldsymbol{\theta}) \in \mathbb{R}^{D\times D}$ are given by

$$\boldsymbol{g}_P(\boldsymbol{\theta}) = \mathbb{E}_{(\boldsymbol{x},\boldsymbol{y})\sim P}\left[\nabla_{\boldsymbol{\theta}}\ell(f(\boldsymbol{\theta},\boldsymbol{x}),\boldsymbol{y})\right] = \int \nabla_{\boldsymbol{\theta}}\ell(f(\boldsymbol{\theta},\boldsymbol{x}),\boldsymbol{y}) P(\boldsymbol{x},\boldsymbol{y})\, d\boldsymbol{x}\, d\boldsymbol{y}\,, \tag{4a}$$

$$\begin{aligned}\boldsymbol{\Sigma}_P(\boldsymbol{\theta}) &= \mathrm{Var}_{(\boldsymbol{x},\boldsymbol{y})\sim P}\left[\nabla_{\boldsymbol{\theta}}\ell(f(\boldsymbol{\theta},\boldsymbol{x}),\boldsymbol{y})\right] \\ &= \int \left(\nabla_{\boldsymbol{\theta}}\ell(f(\boldsymbol{\theta},\boldsymbol{x}),\boldsymbol{y}) - \boldsymbol{g}_P(\boldsymbol{\theta})\right)\left(\nabla_{\boldsymbol{\theta}}\ell(f(\boldsymbol{\theta},\boldsymbol{x}),\boldsymbol{y}) - \boldsymbol{g}_P(\boldsymbol{\theta})\right)^\top P(\boldsymbol{x},\boldsymbol{y})\, d\boldsymbol{x}\, d\boldsymbol{y}\,.\end{aligned} \tag{4b}$$

Empirical approximations: Let $\mathcal{S}$ denote a set of samples drawn i.i.d. from P, i.e. $\mathcal{S} = \{(\boldsymbol{x}_i,\boldsymbol{y}_i) \mid i = 1,\dots,|\mathcal{S}|\}$. With a slight abuse of notation the empirical risk approximated with $\mathcal{S}$ is

$$\mathcal{L}_{\mathcal{S}}(\boldsymbol{\theta}) = \frac{1}{|\mathcal{S}|}\sum_{n\in\mathcal{S}} \ell_n(\boldsymbol{\theta}) \tag{5a}$$

(later, $\mathcal{S}$ will represent either a mini-batch $\mathcal{B}$, or the train set $\mathcal{D}$). The empirical risk gradient $\boldsymbol{g}_{\mathcal{S}}(\boldsymbol{\theta}) \in \mathbb{R}^D$ on $\mathcal{S}$ is

$$\boldsymbol{g}_{\mathcal{S}}(\boldsymbol{\theta}) = \nabla_{\boldsymbol{\theta}}\mathcal{L}_{\mathcal{S}}(\boldsymbol{\theta}) = \frac{1}{|\mathcal{S}|}\sum_{n\in\mathcal{S}} \nabla_{\boldsymbol{\theta}}\ell_n(\boldsymbol{\theta}) = \frac{1}{|\mathcal{S}|}\sum_{n\in\mathcal{S}} \boldsymbol{g}_n(\boldsymbol{\theta})\,, \tag{5b}$$

with individual gradients $\boldsymbol{g}_n(\boldsymbol{\theta}) = \nabla_{\boldsymbol{\theta}}\ell_n(\boldsymbol{\theta}) \in \mathbb{R}^D$ implied by a sample n. Population risk and gradient variances $\Lambda_P(\boldsymbol{\theta}), \boldsymbol{\Sigma}_P(\boldsymbol{\theta})$ can be empirically estimated on $\mathcal{S}$ with the sample variances $\hat{\Lambda}_{\mathcal{S}}(\boldsymbol{\theta}) \in \mathbb{R}, \hat{\boldsymbol{\Sigma}}_{\mathcal{S}}(\boldsymbol{\theta}) \in \mathbb{R}^{D\times D}$, given by

$$\Lambda_P(\boldsymbol{\theta}) \approx \frac{1}{|S|-1}\sum_{n\in\mathcal{S}} \left(\ell_n(\boldsymbol{\theta}) - \mathcal{L}_{\mathcal{S}}(\boldsymbol{\theta})\right)^2 := \hat{\Lambda}_{\mathcal{S}}(\boldsymbol{\theta})\,, \tag{6a}$$

$$\begin{aligned}\boldsymbol{\Sigma}_P(\boldsymbol{\theta}) &\approx \frac{1}{|\mathcal{S}|-1}\sum_{n\in\mathcal{S}} \left(\boldsymbol{g}_n(\boldsymbol{\theta}) - \boldsymbol{g}_{\mathcal{S}}(\boldsymbol{\theta})\right)\left(\boldsymbol{g}_n(\boldsymbol{\theta}) - \boldsymbol{g}_{\mathcal{S}}(\boldsymbol{\theta})\right)^\top := \hat{\boldsymbol{\Sigma}}_{\mathcal{S}}(\boldsymbol{\theta}) \\ &\approx \frac{1}{|\mathcal{S}|-1}\left[\left(\sum_{n\in\mathcal{S}} \boldsymbol{g}_n(\boldsymbol{\theta})\boldsymbol{g}_n(\boldsymbol{\theta})^\top\right) - |\mathcal{S}|\boldsymbol{g}_{\mathcal{S}}(\boldsymbol{\theta})\boldsymbol{g}_{\mathcal{S}}(\boldsymbol{\theta})^\top\right]\,.\end{aligned} \tag{6b}$$

Often, gradient elements are assumed independent and hence their variance is diagonal ($^{\odot 2}$ denotes element-wise square),

$$\mathrm{diag}(\boldsymbol{\Sigma}_P(\boldsymbol{\theta})) \approx \frac{1}{|S|-1}\sum_{n\in\mathcal{S}} \left(\boldsymbol{g}_n(\boldsymbol{\theta}) - \boldsymbol{g}_{\mathcal{S}}(\boldsymbol{\theta})\right)^{\odot 2} = \mathrm{diag}\Big(\hat{\boldsymbol{\Sigma}}_{\mathcal{S}}(\boldsymbol{\theta})\Big) \in \mathbb{R}^D\,. \tag{7}$$

Slicing: To avoid confusion between $\boldsymbol{\theta}_t$ (parameter at iteration t) and $\boldsymbol{\theta}_j$ (j-th parameter entry), we denote the latter as $[\boldsymbol{\theta}]_j$.

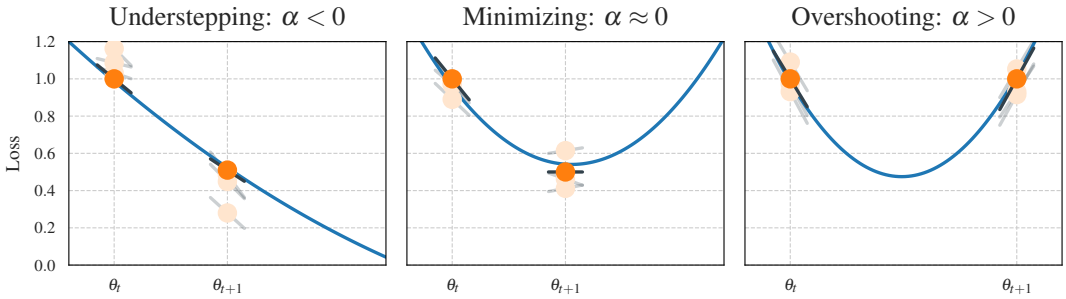


Figure 8: **Motivational sketch for the α quantity.** In each iteration of the optimizer we observe the loss function at two positions $\boldsymbol{\theta}_t$ and $\boldsymbol{\theta}_{t+1}$ (shown in ●). The black lines (—) show the observed slope at this position, which we can get from projecting the gradients onto the current step direction $\boldsymbol{\theta}_{t+1} - \boldsymbol{\theta}_t$. Note, that all four observations (two loss and two slope values) are noisy, due to being computed on a mini-batch. With access to the individual losses and gradients (some samples shown in ●/—), we can estimate their noise level and build a noise-informed quadratic fit (—). Using this fit, we determine whether the optimizer minimizes the local univariate loss (*middle plot*), or whether we understep (*left plot*) or overshoot (*right plot*) the minimum.

C.2 Normalized Step Length (`Alpha`)

Motivation: The goal of the α-quantity is to estimate and quantify the effect that a selected learning rate has on the optimizer's steps. Let's consider the step that the optimizer takes at training iteration t. This parameter update from $\boldsymbol{\theta}_t$ to $\boldsymbol{\theta}_{t+1}$ happens in a one-dimensional space, defined by the update direction $\boldsymbol{\theta}_{t+1} - \boldsymbol{\theta}_t = \boldsymbol{s}_t$. The update direction depends on the update rule of the optimizer, e.g. for SGD with learning rate η it is simply $\boldsymbol{s}_t = -\eta \boldsymbol{g}_{\mathcal{B}_t}(\boldsymbol{\theta}_t)$.

We build a noise-informed univariate quadratic approximation along this update step ($\boldsymbol{\theta}_t \to \boldsymbol{\theta}_{t+1}$) based on the two noisy loss function observations at $\boldsymbol{\theta}_t$ and $\boldsymbol{\theta}_{t+1}$ and the two noisy slope observation at these two points. Examining this quadratic fit, we are able to determine where on this parabola our optimizer steps. Standardizing this, we express a step to the minimum of the loss in the update direction as $\alpha = 0$. Analogously, steps that end short of this minimum result in $\alpha < 0$, and a step over the minimum in $\alpha > 0$. These three different scenarios are illustrated in Figure 8 also showing the underlying observations that would lead to them. Figure 1 shows the distribution of α-values for two very different optimization trajectories.

Noisy observations: In order to build an approximation for the loss function in the update direction, we leverage the four observations of the function (and its derivative) that are available in each iteration. Due to the stochasticity of deep learning optimization, we also take into account the noise-level of all observations by estimating them. The first two observations are the mini-batch training losses $\mathcal{L}_{\mathcal{B}_t}(\boldsymbol{\theta}_t), \mathcal{L}_{\mathcal{B}_{t+1}}(\boldsymbol{\theta}_{t+1})$ at point $\boldsymbol{\theta}_t$ and $\boldsymbol{\theta}_{t+1}$, which are computed in every standard training loop. The mini-batch losses are averages over individual losses,

$$\mathcal{L}_{\mathcal{B}_t}(\boldsymbol{\theta}_t) = \mathbb{E}_{\mathcal{B}_t}\left[\ell(\boldsymbol{\theta}_t)\right] = \frac{1}{|\mathcal{B}_t|} \sum_{n \in \mathcal{B}_t} \ell_n(\boldsymbol{\theta}_t)\,,$$

$$\mathcal{L}_{\mathcal{B}_{t+1}}(\boldsymbol{\theta}_{t+1}) = \mathbb{E}_{\mathcal{B}_{t+1}}\left[\ell(\boldsymbol{\theta}_{t+1})\right] = \frac{1}{|\mathcal{B}_{t+1}|} \sum_{n \in \mathcal{B}_{t+1}} \ell_n(\boldsymbol{\theta}_{t+1})\,,$$

and using these individual losses, we can also compute the variances to estimate the noise-level of our loss observation,

$$\mathrm{Var}_{\mathcal{B}_t}\left[\ell(\boldsymbol{\theta}_t)\right] = \left(\frac{1}{B_t} \sum_{n \in \mathcal{B}_t} \ell_n(\boldsymbol{\theta}_t)^2\right) - \left(\frac{1}{B_t} \sum_{n \in \mathcal{B}_t} \ell_n(\boldsymbol{\theta}_t)\right)^2,$$

$$\mathrm{Var}_{\mathcal{B}_{t+1}}\left[\ell(\boldsymbol{\theta}_{t+1})\right] = \left(\frac{1}{|\mathcal{B}_{t+1}|} \sum_{n \in \mathcal{B}_{t+1}} \ell_n(\boldsymbol{\theta}_{t+1})^2\right) - \left(\frac{1}{|\mathcal{B}_{t+1}|} \sum_{n \in \mathcal{B}_{t+1}} \ell_n(\boldsymbol{\theta}_{t+1})\right)^2.$$

Similarly, we proceed with the slope in the update direction. To compute the slope of the loss function in the direction of the optimizer's update $\boldsymbol{s}_t$, we project the current gradient along this update direction

$$\mathbb{E}_{\mathcal{B}_t}\left[\frac{\boldsymbol{s}_t^\top \boldsymbol{g}(\boldsymbol{\theta}_t)}{\|\boldsymbol{s}_t\|^2}\right] = \frac{1}{|\mathcal{B}_t|}\sum_{n\in\mathcal{B}_t}\frac{\boldsymbol{s}_t^\top \boldsymbol{g}_n(\boldsymbol{\theta}_t)}{\|\boldsymbol{s}_t\|^2}\,,$$
$$\mathbb{E}_{\mathcal{B}_{t+1}}\left[\frac{\boldsymbol{s}_t^\top \boldsymbol{g}(\boldsymbol{\theta}_{t+1})}{\|\boldsymbol{s}_t\|^2}\right] = \frac{1}{|\mathcal{B}_{t+1}|}\sum_{n\in\mathcal{B}_{t+1}}\frac{\boldsymbol{s}_t^\top \boldsymbol{g}_n(\boldsymbol{\theta}_{t+1})}{\|\boldsymbol{s}_t\|^2}\,.$$

Just like before, we can also compute the variance of this slope, by leveraging individual gradients,

$$\mathrm{Var}_{\mathcal{B}_t}\left[\frac{\boldsymbol{s}_t^\top \boldsymbol{g}(\boldsymbol{\theta}_t)}{\|\boldsymbol{s}_t\|^2}\right] = \frac{1}{|\mathcal{B}_t|}\sum_{n\in B_t}\left(\frac{\boldsymbol{s}_t^\top \boldsymbol{g}_n(\boldsymbol{\theta}_t)}{\|\boldsymbol{s}_t\|^2}\right)^2 - \left(\frac{1}{|\mathcal{B}_t|}\sum_{n\in\mathcal{B}_t}\frac{\boldsymbol{s}_t^\top \boldsymbol{g}_n(\boldsymbol{\theta}_t)}{\|\boldsymbol{s}_t\|^2}\right)^2,$$
$$\mathrm{Var}_{\mathcal{B}_{t+1}}\left[\frac{\boldsymbol{s}_t^\top \boldsymbol{g}(\boldsymbol{\theta}_{t+1})}{\|\boldsymbol{s}_t\|^2}\right] = \frac{1}{|\mathcal{B}_{t+1}|}\sum_{n\in\mathcal{B}_{t+1}}\left(\frac{\boldsymbol{s}_t^\top \boldsymbol{g}_n(\boldsymbol{\theta}_{t+1})}{\|\boldsymbol{s}_t\|^2}\right)^2 - \left(\frac{1}{|\mathcal{B}_{t+1}|}\sum_{n\in\mathcal{B}_{t+1}}\frac{\boldsymbol{s}_t^\top \boldsymbol{g}_n(\boldsymbol{\theta}_{t+1})}{\|\boldsymbol{s}_t\|^2}\right)^2.$$

Quadratic fit & normalization: Using our (noisy) observations, we are now ready to build an approximation for the loss as a function of the step size, which we will denote as $f(\tau)$. We assume a quadratic function for f, which follows recent reports for the loss landscape of neural networks [49], i.e. a function $f(\tau) = w_0 + w_1\tau + w_2\tau^2$ parameterized by $\boldsymbol{w} \in \mathbb{R}^3$. We further assume a Gaussian likelihood of the form

$$p\left(\tilde{\boldsymbol{f}}|\boldsymbol{w}, \boldsymbol{\Phi}\right) = \mathcal{N}\left(\tilde{\boldsymbol{f}}; \boldsymbol{\Phi}^\top \boldsymbol{w}, \boldsymbol{\Lambda}\right) \tag{8}$$

for observations $\tilde{\boldsymbol{f}}$ of the loss and its slope. The observation matrix $\boldsymbol{\Phi}$ and the noise matrix of the observations $\boldsymbol{\Lambda}$ are

$$\boldsymbol{\Phi} = \begin{pmatrix} 1 & 1 & 0 & 0 \\ \tau_1 & \tau_2 & 1 & 1 \\ \tau_1^2 & \tau_2^2 & 2\tau_1 & 2\tau_2 \end{pmatrix}, \qquad \boldsymbol{\Lambda} = \begin{pmatrix} \sigma_{\tilde{f}_1} & 0 & 0 & 0 \\ 0 & \sigma_{\tilde{f}_2} & 0 & 0 \\ 0 & 0 & \sigma_{\tilde{f}_1'} & 0 \\ 0 & 0 & 0 & \sigma_{\tilde{f}_2'} \end{pmatrix},$$

where τ denotes the position and σ denotes the noise-level estimate of the observation. The maximum likelihood solution of Equation (8) for the parameters of our quadratic fit is given by

$$\boldsymbol{w} = \left(\boldsymbol{\Phi}\boldsymbol{\Lambda}^{-1}\boldsymbol{\Phi}^\top\right)\boldsymbol{\Phi}\boldsymbol{\Lambda}^{-1}\tilde{\boldsymbol{f}}\,. \tag{9}$$

Once we have the quadratic fit of the univariate loss function in the update direction, we normalize the scales such that the resulting α-value expresses the effective step taken by the optimizer sketched in Figure 8.

Usage: The α-quantity is related to recent line search approaches [28; 45]. However, instead of searching for an acceptable step by repeated attempts, we instead report the effect of the current step size selection. This could, for example, be used to disentangle the two optimization runs in Figure 1. Additionally, this information could also be used to automatically adapt the learning rate during the training process. But, as discussed in Section 3.3, it isn't trivial what the "correct" decision is, as it might depend on the optimization problem, the training phase, and other factors. Having this α-quantity can, however, provide more insight into what kind of steps are used in well-tuned runs with traditional optimizers such as SGD.

C.3 CABS criterion: Coupling adaptive batch sizes with learning rates (`CABS`)

The CABS criterion, proposed by Balles et al. [4], can be used to adapt the mini-batch size during training with SGD. It relies on the gradient noise and approximately optimizes the objective's expected gain per cost. The adaptation rule is (with learning rate η)

$$|\mathcal{B}| \leftarrow \eta\frac{\mathrm{Tr}(\boldsymbol{\Sigma}_P(\boldsymbol{\theta}))}{\mathcal{L}_P(\boldsymbol{\theta})}\,, \tag{10}$$

and the practical implementation approximates $\mathcal{L}_P(\boldsymbol{\theta}) \approx \mathcal{L}_{\mathcal{B}}(\boldsymbol{\theta}), \operatorname{Tr}(\boldsymbol{\Sigma}_P(\boldsymbol{\theta})) \approx \frac{|\mathcal{B}|-1}{|\mathcal{B}|} \operatorname{Tr}(\hat{\boldsymbol{\Sigma}}_{\mathcal{B}}(\boldsymbol{\theta}))$ (compare equations (10, 22) and first paragraph of Section 4 in [4]). This yields the quantity computed in cockpit's `CABS` instrument,

$$|\mathcal{B}| \leftarrow \eta \frac{\frac{1}{|\mathcal{B}|} \sum_{j=1}^{D} \sum_{n \in \mathcal{B}} \left[\boldsymbol{g}_n(\boldsymbol{\theta}) - \boldsymbol{g}_{\mathcal{B}}(\boldsymbol{\theta})\right]_j^2}{\mathcal{L}_{\mathcal{B}}(\boldsymbol{\theta})} \,. \tag{11}$$

Usage: The CABS criterion suggests a batch size which is optimal under certain assumptions. This suggestion can support practitioners in the batch size selection for their deep learning task.

C.4 Early-stopping criterion for SGD (`EarlyStopping`)

The empirical risk $\mathcal{L}_{\mathcal{D}}(\boldsymbol{\theta})$, and the mini-batch loss $\mathcal{L}_{\mathcal{B}}(\boldsymbol{\theta})$ are only estimators of the target objective $\mathcal{L}_P(\boldsymbol{\theta})$. Mahsereci et al. [29] motivate $p(\boldsymbol{g}_{\mathcal{B},\mathcal{D}}(\boldsymbol{\theta}) \mid \boldsymbol{g}_P(\boldsymbol{\theta}) = \boldsymbol{0})$ as a measure for detecting noise in the finite data sets $\mathcal{B}, \mathcal{D}$ due to sampling from P. They propose an evidence-based (EB) criterion for early stopping the training procedure based on mini-batch statistics, and model $p(\boldsymbol{g}_{\mathcal{B}}(\boldsymbol{\theta}))$ with a sampled diagonal variance approximation (compare Equation (7)),

$$p(\boldsymbol{g}_{\mathcal{B}}(\boldsymbol{\theta})) \approx \prod_{j=1}^{D} \mathcal{N}\left(\left[\boldsymbol{g}_P(\boldsymbol{\theta})\right]_j ; \frac{\left[\hat{\boldsymbol{\Sigma}}_{\mathcal{B}}(\boldsymbol{\theta})\right]_{j,j}}{|\mathcal{B}|} \right) \,. \tag{12}$$

Their SGD stopping criterion is

$$\frac{2}{D} \left[\log p(\boldsymbol{g}_{\mathcal{B}}(\boldsymbol{\theta})) - \mathbb{E}_{\boldsymbol{g}_{\mathcal{B}}(\boldsymbol{\theta}) \sim p(\boldsymbol{g}_{\mathcal{B}}(\boldsymbol{\theta}))} \left[\log p(\boldsymbol{g}_{\mathcal{B}}(\boldsymbol{\theta}))\right]\right] > 0 \,, \tag{13a}$$

and translates into

$$1 - \frac{|\mathcal{B}|}{D} \sum_{j=1}^{D} \frac{\left[\boldsymbol{g}_{\mathcal{B}}(\boldsymbol{\theta})\right]_j^2}{\left[\hat{\boldsymbol{\Sigma}}_{\mathcal{B}}(\boldsymbol{\theta})\right]_{j,j}} > 0 \,, \tag{13b}$$

$$1 - \frac{|\mathcal{B}|}{D} \sum_{d=1}^{D} \frac{\left[\boldsymbol{g}_{\mathcal{B}}(\boldsymbol{\theta})\right]_d^2}{\frac{1}{|\mathcal{B}|-1} \sum_{n \in \mathcal{B}} \left[\boldsymbol{g}_n(\boldsymbol{\theta}) - \boldsymbol{g}_{\mathcal{B}}(\boldsymbol{\theta})\right]_d^2} > 0 \,, \tag{13c}$$

$$1 - \frac{|\mathcal{B}|(|\mathcal{B}|-1)}{D} \sum_{d=1}^{D} \frac{\left[\boldsymbol{g}_{\mathcal{B}}(\boldsymbol{\theta})\right]_d^2}{\left(\sum_{n \in \mathcal{B}} \left[\boldsymbol{g}_n(\boldsymbol{\theta})\right]_d^2\right) - |\mathcal{B}| \left[\boldsymbol{g}_{\mathcal{B}}(\boldsymbol{\theta})\right]_d^2} > 0 \,. \tag{13d}$$

COCKPIT's `EarlyStopping` quantity computes the left-hand side of Equation (13d).

Usage: The `EarlyStopping` quantity of COCKPIT can inform the practitioner that training is about to be completed and the model might be at risk of overfitting.

C.5 Individual gradient element histograms (`GradHist1d`, `GradHist2d`)

For the $|\mathcal{B}| \times D$ individual gradient elements, COCKPIT's `GradHist1d` instrument displays a histogram of

$$\{\boldsymbol{g}_n(\boldsymbol{\theta}_j)\}_{n \in \mathcal{B}, j=1,\dots,D} \,. \tag{14}$$

COCKPIT's `GradHist2d` instrument displays a two-dimensional histogram of the $|\mathcal{B}| \times D$ tuples

$$\{(\boldsymbol{\theta}_j, \boldsymbol{g}_n(\boldsymbol{\theta}_j))\}_{n \in \mathcal{B}, j=1,\dots,D} \tag{15}$$

and the marginalized one-dimensional histograms over the parameter and gradient axes.

Usage: Sections 3.1 and 3.2 provide use cases (identifying data pre-processing issues and vanishing gradients) for both the gradient histogram as well as its layer-wise extension.

C.6 Gradient tests (`NormTest`, `InnerTest`, `OrthoTest`)

Bollapragada et al. [7] and Byrd et al. [9] propose batch size adaptation schemes based on the gradient noise. They formulate geometric constraints between population and mini-batch gradient and accessible approximations that can be probed to decide whether the mini-batch size should be increased. Because mini-batches are i.i.d. from P, it holds that

$$\mathbb{E}\left[\boldsymbol{g}_{\mathcal{B}}(\boldsymbol{\theta})\right] = \boldsymbol{g}_P(\boldsymbol{\theta}), \tag{16a}$$

$$\mathbb{E}\left[\boldsymbol{g}_{\mathcal{B}}(\boldsymbol{\theta})^\top \boldsymbol{g}_P(\boldsymbol{\theta})\right] = \|\boldsymbol{g}_P(\boldsymbol{\theta})\|^2. \tag{16b}$$

The above works propose enforcing other weaker similarity in expectation during optimization. These geometric constraints reduce to basic vector geometry (see Figure 9 (a) for an overview of the relevant vectors). We recall their formulation here for consistency and derive the practical versions, which can be computed from training observables and are used in COCKPIT (consult Figure 9 (b) for the visualization).

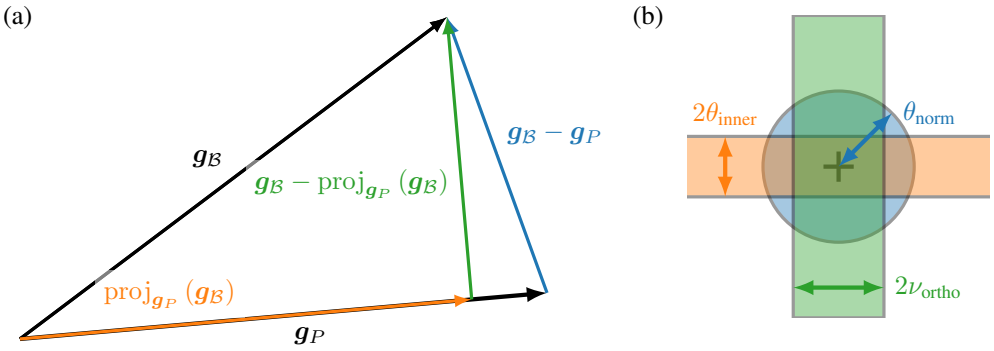


Figure 9: **Conceptual sketch for gradient tests.** *(a)* Relevant vectors to formulate the geometric constraints between population and mini-batch gradient probed by the gradient tests. *(b)* Gradient test visualization in COCKPIT.

Usage: All three gradient tests describe the noise level of the gradients. Bollapragada et al. [7] and Byrd et al. [9] adapt the batch size so that the proposed geometric constraints are fulfilled. Practitioners can use the combined gradient test plot, i.e. top center plot in Figure 2, to monitor gradient noise during training and adjust hyperparameters such as the batch size.

C.6.1 Norm test (`NormTest`)

The norm test [9] constrains the residual norm $\|\boldsymbol{g}_{\mathcal{B}}(\boldsymbol{\theta}) - \boldsymbol{g}_P(\boldsymbol{\theta})\|$, rescaled by $\|\boldsymbol{g}_P(\boldsymbol{\theta})\|$. This gives rise to a standardized ball of radius $\theta_{\text{norm}} \in (0, \infty)$ around the population gradient, where the mini-batch gradient should reside. Byrd et al. [9] set $\theta_{\text{norm}} = 0.9$ in their experiments and increase the batch size if (in the practical version, see below) the following constraint is not fulfilled

$$\mathbb{E}\left[\frac{\|\boldsymbol{g}_{\mathcal{B}}(\boldsymbol{\theta}) - \boldsymbol{g}_P(\boldsymbol{\theta})\|^2}{\|\boldsymbol{g}_P(\boldsymbol{\theta})\|^2}\right] \leq \theta_{\text{norm}}^2 \,. \tag{17a}$$

Instead of taking the expectation over mini-batches, Byrd et al. [9] note that the above will be satisfied if

$$\frac{1}{|\mathcal{B}|}\mathbb{E}\left[\frac{\|\boldsymbol{g}_n(\boldsymbol{\theta}) - \boldsymbol{g}_P(\boldsymbol{\theta})\|^2}{\|\boldsymbol{g}_P(\boldsymbol{\theta})\|^2}\right] \leq \theta_{\text{norm}}^2 \,. \tag{17b}$$

They propose a practical form of this test,

$$\frac{1}{|\mathcal{B}|(|\mathcal{B}|-1)}\frac{\sum_{n\in\mathcal{B}}\|\boldsymbol{g}_n(\boldsymbol{\theta}) - \boldsymbol{g}_{\mathcal{B}}(\boldsymbol{\theta})\|^2}{\|\boldsymbol{g}_{\mathcal{B}}(\boldsymbol{\theta})\|^2} \leq \theta_{\text{norm}}^2 \,, \tag{18a}$$

which can be computed from mini-batch statistics. Rearranging

$$\sum_{n\in\mathcal{B}} \left\|\boldsymbol{g}_n(\boldsymbol{\theta}) - \boldsymbol{g}_{\mathcal{B}}(\boldsymbol{\theta})\right\|^2 = \left(\sum_{n\in\mathcal{B}} \left\|\boldsymbol{g}_n(\boldsymbol{\theta})\right\|^2\right) - |\mathcal{B}| \left\|\boldsymbol{g}_{\mathcal{B}}(\boldsymbol{\theta})\right\|^2 , \tag{18b}$$

we arrive at

$$\frac{1}{|\mathcal{B}|(|\mathcal{B}|-1)} \left[\frac{\sum_{n\in\mathcal{B}} \left\|\boldsymbol{g}_n(\boldsymbol{\theta})\right\|^2}{\left\|\boldsymbol{g}_{\mathcal{B}}(\boldsymbol{\theta})\right\|^2} - |\mathcal{B}|\right] \leq \theta_{\text{norm}}^2 \tag{18c}$$

that leverages the norm of both the mini-batch and the individual gradients, which can be aggregated over parameters during a backward pass. COCKPIT's `NormTest` corresponds to the maximum radius θ_{norm} for which the above inequality holds.

C.6.2 Inner product test (`InnerTest`)

The inner product test [7] constrains the projection of $\boldsymbol{g}_{\mathcal{B}}(\boldsymbol{\theta})$ onto $\boldsymbol{g}_P(\boldsymbol{\theta})$ (compare Figure 9 (a)),

$$\text{proj}_{\boldsymbol{g}_P(\boldsymbol{\theta})}\left(\boldsymbol{g}_{\mathcal{B}}(\boldsymbol{\theta})\right) = \frac{\boldsymbol{g}_{\mathcal{B}}(\boldsymbol{\theta})^\top \boldsymbol{g}_P(\boldsymbol{\theta})}{\left\|\boldsymbol{g}_P(\boldsymbol{\theta})\right\|^2} \boldsymbol{g}_P(\boldsymbol{\theta}) , \tag{19}$$

rescaled by $\|\boldsymbol{g}_P(\boldsymbol{\theta})\|$. This restricts the mini-batch gradient to reside in a standardized band of relative width $\theta_{\text{inner}} \in (0, \infty)$ around the population risk gradient. Bollapragada et al. [7] use $\theta_{\text{inner}} = 0.9$ (in the practical version, see below) to adapt the batch size if the parallel component's variance does not satisfy the condition

$$\text{Var}\left(\frac{\boldsymbol{g}_{\mathcal{B}}(\boldsymbol{\theta})^\top \boldsymbol{g}_P(\boldsymbol{\theta})}{\left\|\boldsymbol{g}_P(\boldsymbol{\theta})\right\|^2}\right) = \mathbb{E}\left[\left(\frac{\boldsymbol{g}_{\mathcal{B}}(\boldsymbol{\theta})^\top \boldsymbol{g}_P(\boldsymbol{\theta})}{\left\|\boldsymbol{g}_P(\boldsymbol{\theta})\right\|^2} - 1\right)^2\right] \leq \theta_{\text{inner}}^2 \tag{20a}$$

(note that by Equation (16) we have $\mathbb{E}\left[\frac{\boldsymbol{g}_{\mathcal{B}}(\boldsymbol{\theta})^\top \boldsymbol{g}_P(\boldsymbol{\theta})}{\left\|\boldsymbol{g}_P(\boldsymbol{\theta})\right\|^2}\right] = 1$). Bollapragada et al. [7] bound Equation (20a) by the individual gradient variance,

$$\frac{1}{|\mathcal{B}|}\text{Var}\left(\frac{\boldsymbol{g}_n(\boldsymbol{\theta})^\top \boldsymbol{g}_P(\boldsymbol{\theta})}{\left\|\boldsymbol{g}_P(\boldsymbol{\theta})\right\|^2}\right) = \frac{1}{|\mathcal{B}|}\mathbb{E}\left[\left(\frac{\boldsymbol{g}_n(\boldsymbol{\theta})^\top \boldsymbol{g}_P(\boldsymbol{\theta})}{\left\|\boldsymbol{g}_P(\boldsymbol{\theta})\right\|^2} - 1\right)^2\right] \leq \theta_{\text{inner}}^2 . \tag{20b}$$

They then propose a practical form of Equation (20b), which uses the mini-batch sample variance,

$$\frac{1}{|\mathcal{B}|}\text{Var}\left(\frac{\boldsymbol{g}_n(\boldsymbol{\theta})^\top \boldsymbol{g}_{\mathcal{B}}(\boldsymbol{\theta})}{\left\|\boldsymbol{g}_{\mathcal{B}}(\boldsymbol{\theta})\right\|^2}\right) = \frac{1}{|\mathcal{B}|(|\mathcal{B}|-1)}\left[\sum_{n\in\mathcal{B}}\left(\frac{\boldsymbol{g}_n(\boldsymbol{\theta})^\top \boldsymbol{g}_{\mathcal{B}}(\boldsymbol{\theta})}{\left\|\boldsymbol{g}_{\mathcal{B}}(\boldsymbol{\theta})\right\|^2} - 1\right)^2\right] \leq \theta_{\text{inner}}^2 . \tag{21a}$$

Expanding

$$\sum_{n\in\mathcal{B}}\left(\frac{\boldsymbol{g}_n(\boldsymbol{\theta})^\top \boldsymbol{g}_{\mathcal{B}}(\boldsymbol{\theta})}{\left\|\boldsymbol{g}_{\mathcal{B}}(\boldsymbol{\theta})\right\|^2} - 1\right)^2 = \frac{\sum_{n\in\mathcal{B}}\left(\boldsymbol{g}_n(\boldsymbol{\theta})^\top \boldsymbol{g}_{\mathcal{B}}(\boldsymbol{\theta})\right)^2}{\left\|\boldsymbol{g}_{\mathcal{B}}(\boldsymbol{\theta})\right\|^4} - |\mathcal{B}| \tag{21b}$$

and inserting Equation (21b) into Equation (21a) yields

$$\frac{1}{|\mathcal{B}|(|\mathcal{B}|-1)}\left[\frac{\sum_{n\in\mathcal{B}}\left(\boldsymbol{g}_n(\boldsymbol{\theta})^\top \boldsymbol{g}_{\mathcal{B}}(\boldsymbol{\theta})\right)^2}{\left\|\boldsymbol{g}_{\mathcal{B}}(\boldsymbol{\theta})\right\|^4} - |\mathcal{B}|\right] \leq \theta_{\text{inner}}^2 . \tag{21c}$$

It relies on pairwise scalar products between individual gradients, which can be aggregated over layers during backpropagation. COCKPIT's `InnerTest` quantity computes the maximum band width θ_{inner} that satisfies Equation (21c).

C.6.3 Orthogonality test (`OrthoTest`)

In contrast to the inner product test (Appendix C.6.2) which constrains the projection (Equation (19)), the orthogonality test [7] constrains the orthogonal part (see Figure 9 (a))

$$\boldsymbol{g}_{\mathcal{B}}(\boldsymbol{\theta}) - \operatorname{proj}_{\boldsymbol{g}_P(\boldsymbol{\theta})}\left(\boldsymbol{g}_{\mathcal{B}}(\boldsymbol{\theta})\right) , \tag{22}$$

rescaled by $\|\boldsymbol{g}_P(\boldsymbol{\theta})\|$. This restricts the mini-batch gradient to a standardized band of relative width $\nu_{\text{ortho}} \in (0, \infty)$ parallel to the population gradient. Bollapragada et al. [7] use $\nu = \tan(80°) \approx 5.84$ (in the practical version, see below) to adapt the batch size if the following condition is violated,

$$\mathbb{E}\left[\left\| \frac{\boldsymbol{g}_{\mathcal{B}}(\boldsymbol{\theta}) - \operatorname{proj}_{\boldsymbol{g}_P(\boldsymbol{\theta})}\left(\boldsymbol{g}_{\mathcal{B}}(\boldsymbol{\theta})\right)}{\|\boldsymbol{g}_P(\boldsymbol{\theta})\|} \right\|^2\right] \leq \nu_{\text{ortho}}^2 . \tag{23a}$$

Expanding the norm, and inserting Equation (19), this simplifies to

$$\begin{aligned} \mathbb{E}\left[\left\| \frac{\boldsymbol{g}_{\mathcal{B}}(\boldsymbol{\theta})}{\|\boldsymbol{g}_P(\boldsymbol{\theta})\|} - \frac{\boldsymbol{g}_{\mathcal{B}}(\boldsymbol{\theta})^\top \boldsymbol{g}_P(\boldsymbol{\theta})}{\|\boldsymbol{g}_P(\boldsymbol{\theta})\|^2} \frac{\boldsymbol{g}_P(\boldsymbol{\theta})}{\|\boldsymbol{g}_P(\boldsymbol{\theta})\|} \right\|^2\right] &\leq \nu_{\text{ortho}}^2 , \\ \mathbb{E}\left[\frac{\|\boldsymbol{g}_{\mathcal{B}}(\boldsymbol{\theta})\|^2}{\|\boldsymbol{g}_P(\boldsymbol{\theta})\|^2} - \frac{\left(\boldsymbol{g}_{\mathcal{B}}(\boldsymbol{\theta})^\top \boldsymbol{g}_P(\boldsymbol{\theta})\right)^2}{\|\boldsymbol{g}_P(\boldsymbol{\theta})\|^4}\right] &\leq \nu_{\text{ortho}}^2 . \end{aligned} \tag{23b}$$

Bollapragada et al. [7] bound this inequality using individual gradients instead,

$$\frac{1}{|\mathcal{B}|}\mathbb{E}\left[\left\| \frac{\boldsymbol{g}_n(\boldsymbol{\theta})}{\|\boldsymbol{g}_P(\boldsymbol{\theta})\|^2} - \frac{\boldsymbol{g}_n(\boldsymbol{\theta})^\top \boldsymbol{g}_P(\boldsymbol{\theta})}{\|\boldsymbol{g}_P(\boldsymbol{\theta})\|^2} \frac{\boldsymbol{g}_P(\boldsymbol{\theta})}{\|\boldsymbol{g}_P(\boldsymbol{\theta})\|} \right\|^2\right] \leq \nu_{\text{ortho}}^2 . \tag{23c}$$

They propose the practical form

$$\frac{1}{|\mathcal{B}|(|\mathcal{B}|-1)}\mathbb{E}\left[\left\| \frac{\boldsymbol{g}_n(\boldsymbol{\theta})}{\|\boldsymbol{g}_{\mathcal{B}}(\boldsymbol{\theta})\|} - \frac{\boldsymbol{g}_n(\boldsymbol{\theta})^\top \boldsymbol{g}_{\mathcal{B}}(\boldsymbol{\theta})}{\|\boldsymbol{g}_{\mathcal{B}}(\boldsymbol{\theta})\|^2} \frac{\boldsymbol{g}_{\mathcal{B}}(\boldsymbol{\theta})}{\|\boldsymbol{g}_{\mathcal{B}}(\boldsymbol{\theta})\|} \right\|^2\right] \leq \nu_{\text{ortho}}^2 , \tag{24a}$$

which simplifies to

$$\frac{1}{|\mathcal{B}|(|\mathcal{B}|-1)} \sum_{n \in \mathcal{B}} \left(\frac{\|\boldsymbol{g}_n(\boldsymbol{\theta})\|^2}{\|\boldsymbol{g}_{\mathcal{B}}(\boldsymbol{\theta})\|^2} - \frac{\left(\boldsymbol{g}_n(\boldsymbol{\theta})^\top \boldsymbol{g}_{\mathcal{B}}(\boldsymbol{\theta})\right)^2}{\|\boldsymbol{g}_{\mathcal{B}}(\boldsymbol{\theta})\|^4} \right) \leq \nu_{\text{ortho}}^2 . \tag{24b}$$

It relies on pairwise scalar products between individual gradients which can be aggregated over layers during a backward pass.COCKPIT's `OrthTest` quantity computes the maximum band width ν_{ortho} which satisfies Equation (24b).

Relation to acute angle test: Recently, a novel "acute angle test" was proposed by Bahamou & Goldfarb [3]. While the theoretical constraint between $\boldsymbol{g}_{\mathcal{B}}(\boldsymbol{\theta})$ and $\boldsymbol{g}_P(\boldsymbol{\theta})$ differs from the orthogonality test, the practical versions coincide. Hence, we do not incorporate the acute angle here.

C.7 Hessian maximum eigenvalue (`HessMaxEV`)

The Hessian's maximum eigenvalue $\lambda_{\text{max}}(\boldsymbol{H}_{\mathcal{B}}(\boldsymbol{\theta}))$ is computed with an iterative eigensolver from Hessian-vector products through PYTORCH's automatic differentiation [34]. Like Yao et al. [50], we employ power iterations with similar default stopping parameters (stop after at most 100 iterations, or if the iterate does converged with a relative and absolute tolerance of $10^{-3}, 10^{-6}$, respectively) to compute $\lambda_{\text{max}}(\boldsymbol{H}_{\mathcal{B}}(\boldsymbol{\theta}))$ through the `HessMaxEV` quantity in COCKPIT.

In principle, more sophisticated eigensolvers (for example Arnoldi's method) could be applied to converge in fewer iterations or compute eigenvalues other than the leading ones. Warsa et al. [46] empirically demonstrate that the FLOP ratio between power iteration and implicitly restarted Arnoldi method can reach values larger than 100. While we can use such a beneficial method on a CPU through `scipy.sparse.linalg.eigsh` we are restricted to the GPU-compatible power iteration for GPU training. We expect that extending the support of popular machine learning libraries like PYTORCH for such iterative eigensolvers on GPUs can help to save computation time.

$$\lambda_{\text{max}}(\boldsymbol{H}_{\mathcal{B}}(\boldsymbol{\theta})) = \max_{\|\boldsymbol{v}\|=1} \|\boldsymbol{H}_{\mathcal{B}}(\boldsymbol{\theta})\boldsymbol{v}\| = \max_{\boldsymbol{v} \in \mathbb{R}^D} \frac{\boldsymbol{v}^\top \boldsymbol{H}_{\mathcal{B}}(\boldsymbol{\theta})\boldsymbol{v}}{\boldsymbol{v}^\top \boldsymbol{v}} . \tag{25}$$

Usage: The Hessian's maximum eigenvalue describes the loss surface's sharpest direction and thus provides an understanding of the current loss landscape. Additionally, in convex optimization, the largest Hessian eigenvalue crucially determines the appropriate step size [38]. In Section 4, we can observe that although training seems stuck in the very first few iterations progress is visible when looking at the maximum Hessian eigenvalue.

C.8 Hessian trace (`HessTrace`)

In comparison to Yao et al. [50], who leverage Hessian-vector products [34] to estimate the Hessian trace, we compute the exact value $\mathrm{Tr}(\boldsymbol{H}_{\mathcal{B}}(\boldsymbol{\theta}))$ with the `HessTrace` quantity in COCKPIT by aggregating the output of BACKPACK's `DiagHessian` extension, which computes the diagonal entries of $\boldsymbol{H}_{\mathcal{B}}(\boldsymbol{\theta})$. Alternatively, the trace can also be estimated from the generalized Gauss-Newton matrix, or an MC-sampled approximation thereof.

Usage: The Hessian trace equals the sum of the eigenvalues and thus provides a notion of "average curvature" of the current loss landscape. It has long been theorized and discussed that curvature and generalization performance may be linked [21, e.g.].

C.9 Takeuchi Information Criterion (TIC) (`TICDiag`, `TICTrace`)

Recent work by Thomas et al. [43] suggests that optimizer convergence speed and generalization is mainly influenced by curvature and gradient noise; and hence their interaction is crucial to understand the generalization and optimization behavior of deep neural networks. They reinvestigate the Takeuchi Information criterion [42], an estimator for the generalization gap in overparameterized maximum likelihood estimation. At a local minimum $\boldsymbol{\theta}^\star$, the generalization gap is estimated by the TIC

$$\frac{1}{|\mathcal{D}|} \mathrm{Tr}\left(\boldsymbol{H}_P(\boldsymbol{\theta}^\star)^{-1} \boldsymbol{C}_P(\boldsymbol{\theta}^\star)\right) , \tag{26}$$

where $\boldsymbol{H}_P(\boldsymbol{\theta}^\star)$ is the population Hessian and $\boldsymbol{C}_P(\boldsymbol{\theta}^\star)$ is the gradient's uncentered second moment,

$$\boldsymbol{C}_P(\boldsymbol{\theta}^\star) = \int \nabla_{\boldsymbol{\theta}} \ell(f(\boldsymbol{\theta}^\star, \boldsymbol{x}), \boldsymbol{y}) \left(\nabla_{\boldsymbol{\theta}} \ell(f(\boldsymbol{\theta}^\star, \boldsymbol{x}), \boldsymbol{y})\right)^\top P(\boldsymbol{x}, \boldsymbol{y}) \, d\boldsymbol{x} \, d\boldsymbol{y}.$$

Both matrices are inaccessible in practice. In their experiments, Thomas et al. [43] propose the approximation $\mathrm{Tr}(\boldsymbol{C}) / \mathrm{Tr}(\boldsymbol{H})$ for $\mathrm{Tr}(\boldsymbol{H}^{-1}\boldsymbol{C})$. They also replace the Hessian by the Fisher as it is easier to compute. With these practical simplifications, they investigate the TIC of trained neural networks where the curvature and noise matrix are evaluated on a large data set.

The TIC provided in COCKPIT differs from this setting, since by design we want to observe quantities during training, while avoiding additional model predictions. Also, BACKPACK provides access to the Hessian; hence we don't need to use the Fisher. We propose the following two approximations of the TIC from a mini-batch:

- `TICTrace`: Uses the approximation of Thomas et al. [43] which replaces the matrix-product trace by the product of traces,

$$\frac{\mathrm{Tr}\left(\boldsymbol{C}_{\mathcal{B}}(\boldsymbol{\theta})\right)}{\mathrm{Tr}\left(\boldsymbol{H}_{\mathcal{B}}(\boldsymbol{\theta})\right)} = \frac{\frac{1}{|\mathcal{B}|}\sum_{n\in\mathcal{B}} \|\boldsymbol{g}_n(\boldsymbol{\theta})\|^2}{\mathrm{Tr}\left(\boldsymbol{H}_{\mathcal{B}}(\boldsymbol{\theta})\right)} . \tag{27}$$

- `TICDiag`: Uses a diagonal approximation of the Hessian, which is cheap to invert,

$$\mathrm{Tr}\left(\mathrm{diag}\left(\boldsymbol{H}_{\mathcal{B}}(\boldsymbol{\theta})\right)^{-1} \boldsymbol{C}_{\mathcal{B}}(\boldsymbol{\theta})\right) = \frac{1}{|\mathcal{B}|} \sum_{j=1}^{D} [\boldsymbol{H}_{\mathcal{B}}(\boldsymbol{\theta})]_{j,j}^{-1} \left[\sum_{n\in\mathcal{B}} \boldsymbol{g}_n(\boldsymbol{\theta})^{\odot 2}\right]_j . \tag{28}$$

Usage: The TIC is a proxy for the model's generalization gap, see Thomas et al. [43].

C.10 Gradient signal-to-noise-ratio (`MeanGSNR`)

The gradient signal-to-noise-ratio $\mathrm{GSNR}([\boldsymbol{\theta}]_j) \in \mathbb{R}$ for a single parameter $[\boldsymbol{\theta}]_j$ is defined as

$$\mathrm{GSNR}([\boldsymbol{\theta}]_j) = \frac{\mathbb{E}_{(\boldsymbol{x},\boldsymbol{y})\sim P}\left[[\nabla_{\boldsymbol{\theta}} \ell(f(\boldsymbol{\theta}, \boldsymbol{x}), \boldsymbol{y})]_j\right]^2}{\mathrm{Var}_{(\boldsymbol{x},\boldsymbol{y})\sim P}\left[[\nabla_{\boldsymbol{\theta}} \ell(f(\boldsymbol{\theta}, \boldsymbol{x}), \boldsymbol{y})]_j\right]} = \frac{[\boldsymbol{g}_P(\boldsymbol{\theta})]_j^2}{[\boldsymbol{\Sigma}_P(\boldsymbol{\theta})]_{j,j}} \,. \tag{29}$$

Liu et al. [27] use it to explain generalization properties of models in the early training phase. We apply their estimation to mini-batches,

$$\mathrm{GSNR}([\boldsymbol{\theta}]_j) \approx \frac{[\boldsymbol{g}_{\mathcal{B}}(\boldsymbol{\theta})]_j^2}{\frac{|\mathcal{B}|-1}{|\mathcal{B}|}\left[\hat{\boldsymbol{\Sigma}}_{\mathcal{B}}(\boldsymbol{\theta})\right]_{j,j}} = \frac{[\boldsymbol{g}_{\mathcal{B}}(\boldsymbol{\theta})]_j^2}{\frac{1}{|\mathcal{B}|}\left(\sum_{n\in\mathcal{B}}[\boldsymbol{g}_n(\boldsymbol{\theta})]_j^2\right) - [\boldsymbol{g}_{\mathcal{B}}(\boldsymbol{\theta})]_j^2} \,. \tag{30a}$$

Inspired by Liu et al. [27], COCKPIT's `MeanGSNR` computes the average GSNR over all parameters,

$$\frac{1}{D}\sum_{j=1}^{D} \mathrm{GSNR}([\boldsymbol{\theta}]_j) \,. \tag{30b}$$

Usage: The GSNR describes the gradient noise level which is influenced, among other things, by the batch size. Using the GSNR, perhaps in combination with the gradient tests or the CABS criterion could provide practitioners a clearer picture of suitable batch sizes for their particular problem. As shown by Liu et al. [27], the GSNR is also linked to generalization of neural networks.

D Additional experiments

In this section, we present additional experiments and use cases that showcase COCKPIT's utility. Appendix D.1 shows that COCKPIT is able to scale to larger data sets by running the experiment with incorrectly scaled data (see Section 3.1) on IMAGENET instead of CIFAR-10. Appendix D.2 provides another concrete use case similar to Figure 1: detecting regularization during training.

D.1 Incorrectly scaled data for IMAGENET

We repeat the experiment of Section 3.1 on the IMAGENET [13] data set instead of CIFAR-10. We also use a larger neural network model, switching from 3C3D to VGG16 [40]. This demonstrates that COCKPIT is able to scale to both larger models and data sets. The input size of the images is almost fifty times larger (224×224 instead of 32×32). The model size increased by roughly a factor of 150 (VGG16 for IMAGENET has roughly 138 million parameters, 3C3D has less than a million).

Similar to the example shown in the main text, the gradients are affected by the scaling introduced via the input images, albeit less drastically (see Figure 10). Due to the gradient scaling, default optimization hyperparameters might not work well anymore for the model using the raw input data.

D.2 Detecting implicit regularization of the optimizer

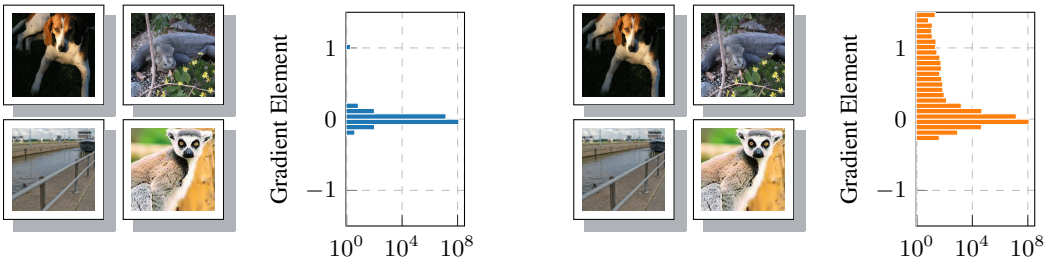
In non-convex optimization, optimizers can converge to local minima with different properties. Here, we illustrate this by investigating the effect of sub-sampling noise on a simple task from [30; 18].

We generate synthetic data $\mathcal{D} = \{(x_n, y_n) \in \mathbb{R} \times \mathbb{R}\}_{n=1}^{N=100}$ for a regression task with $x \sim \mathcal{N}(0; 1)$ with noisy observations $y = 1.4x + \epsilon$ where $\epsilon \sim \mathcal{N}(0; 1)$. The model is a scalar network with parameters $\boldsymbol{\theta} = (w_1 \quad w_2)^\top \in \mathbb{R}^2$, initialized at $\boldsymbol{\theta}_0 = (0.1 \quad 1.7)^\top$, that produces predictions via $f(\boldsymbol{\theta}, x) = w_2 w_1 x$. We seek to minimize the mean squared error

$$\mathcal{L}_{\mathcal{D}}(\boldsymbol{\theta}) = \frac{1}{N}\sum_{n=1}^{N} \left(f(\theta, x_n) - y_n\right)^2$$

and compare SGD ($|\mathcal{B}| = 95$) with GD ($|\mathcal{B}| = N = 100$) at a learning rate of 0.1 (see Figure 11).

We observe that the loss of both SGD and GD is almost identical. Using a noisy gradient regularizes the Hessian's maximum eigenvalue though. It decreases in later stages where the loss curve suggests that training has converged. This regularization effect constitutes an important phenomenon that cannot be observed by monitoring only the loss.



(a) Normalized Data (b) Raw Data

Figure 10: **Same inputs, different gradients on IMAGENET.** This is structurally the same plot as Figure 3, but using IMAGENET and VGG16. (a) *normalized* ($[0, 1]$) and (b) *raw* ($[0, 255]$) images look identical in auto-scaled front-ends like MATPLOTLIB's `imshow`. The gradient distribution on the VGG16 model, however, is affected by this scaling.

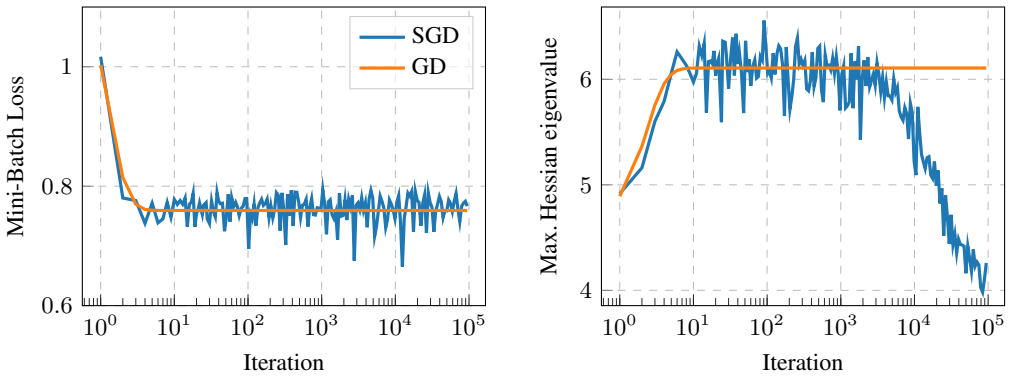


Figure 11: **Observing implicit regularization of the optimizer with COCKPIT** through a comparison of SGD and GD on a synthetic problem inspired by [30; 18] (details in the text). *Left:* The mini-batch loss of both optimizers looks similar. *Right:* Noise due to mini-batching regularizes the Hessian's maximum eigenvalue in stages where the loss suggests that training has converged.

E Implementation details and additional benchmarks

In this section, we provide more details about our implementation (Appendix E.1) to access the desired quantities with as little overhead as possible. Additionally, we present more benchmarks for individual instruments (Appendix E.2.1) and COCKPIT configurations (Appendix E.2.2). These are similar but extended versions of the ones presented in Figures 6a and 6b in the main text. Lastly, we benchmark different implementations of computing the two-dimensional gradient histogram (Appendix E.3), identifying a computational bottleneck for its current GPU implementation.

Hardware details: Throughout this paper, we conducted benchmarks on the following setup

- **CPU:** Intel Core i7-8700K CPU @ 3.70 GHz × 12 (32 GB)
- **GPU:** NVIDIA GeForce RTX 2080 Ti (11 GB)

Test problem details: The experiments in this paper rely mostly on optimization problems provided by the DEEPOBS benchmark suite [39]. If not stated otherwise, we use the default training details suggested by DEEPOBS, that are summarized below. For more details see the original paper.

- **Quadratic Deep:** A stochastic quadratic problem with an eigenspectrum similar to what has been reported for neural nets. Default batch size 128, default number of epochs 100.
- **MNIST Log. Reg.:** Multinomial logistic regression on MNIST [26]. Default batch size 128, default number of epochs 50.
- **MNIST MLP:** Multi-layer perceptron neural network on MNIST. Default batch size 128, default number of epochs 100.
- **FASHION-MNIST MLP:** Multi-layer perceptron neural network on FASHION-MNIST [48]. Default batch size 128, default number of epochs 100.
- **FASHION-MNIST 2C2D:** A two convolutional and two dense layered neural network on FASHION-MNIST. Default batch size 128, default number of epochs 100.
- **CIFAR-10 3C3D:** A three convolutional and three dense layered neural network on CIFAR-10 [25]. Default batch size 128, default number of epochs 100.
- **CIFAR-100 ALL-CNN-C:** All Convolutional Neural Network C (ALL-CNN-C [41]) on CIFAR-100 [25]. Default batch size 256, default number of epochs 350.
- **SVHN 3C3D:** A three convolutional and three dense layered neural network on SVHN [32]. Default batch size 128, default number of epochs 100.

E.1 Hooks & Memory benchmarks

To improve memory consumption, we compact information during the backward pass by adding hooks to the neural network's layers. These are executed after BACKPACK extensions and have access to the quantities computed therein. They compress information to what is requested by a quantity and free the memory occupied by BACKPACK buffers. Such savings primarily depend on the parameter distribution over layers, and are bigger for more balanced architectures (compare Figure 12).

Example: Say, we want to compute a histogram over the $|\mathcal{B}| \times D$ individual gradient elements of a network. Suppose that $|\mathcal{B}| = 128$ and the model is DEEPOBS's CIFAR-10 3C3D test problem with $895,210$ parameters. Given that every parameter is stored in single precision, the model requires $895,210 \times 4\,\text{Bytes} \approx 3.41\,\text{MB}$. Storing the individual gradients will require $128 \times 895,210 \times 4\,\text{Bytes} \approx 437\,\text{MB}$ (for larger networks this quickly exceeds the available memory as the individual gradients occupy $|\mathcal{B}|$ times the model size). If instead, the layer-wise individual gradients are condensed into histograms of negligible size and immediately freed afterwards during backpropagation, the maximum memory overhead reduces to storing the individual gradients of the largest layer. For our example, the largest layer has $589,824$ parameters, and the associated individual gradients will require $128 \times 589,824 \times 4\,\text{Bytes} \approx 288\,\text{MB}$, saving roughly $149\,\text{MB}$ of RAM. In practice, we observe these expected savings, see Figure 12c.

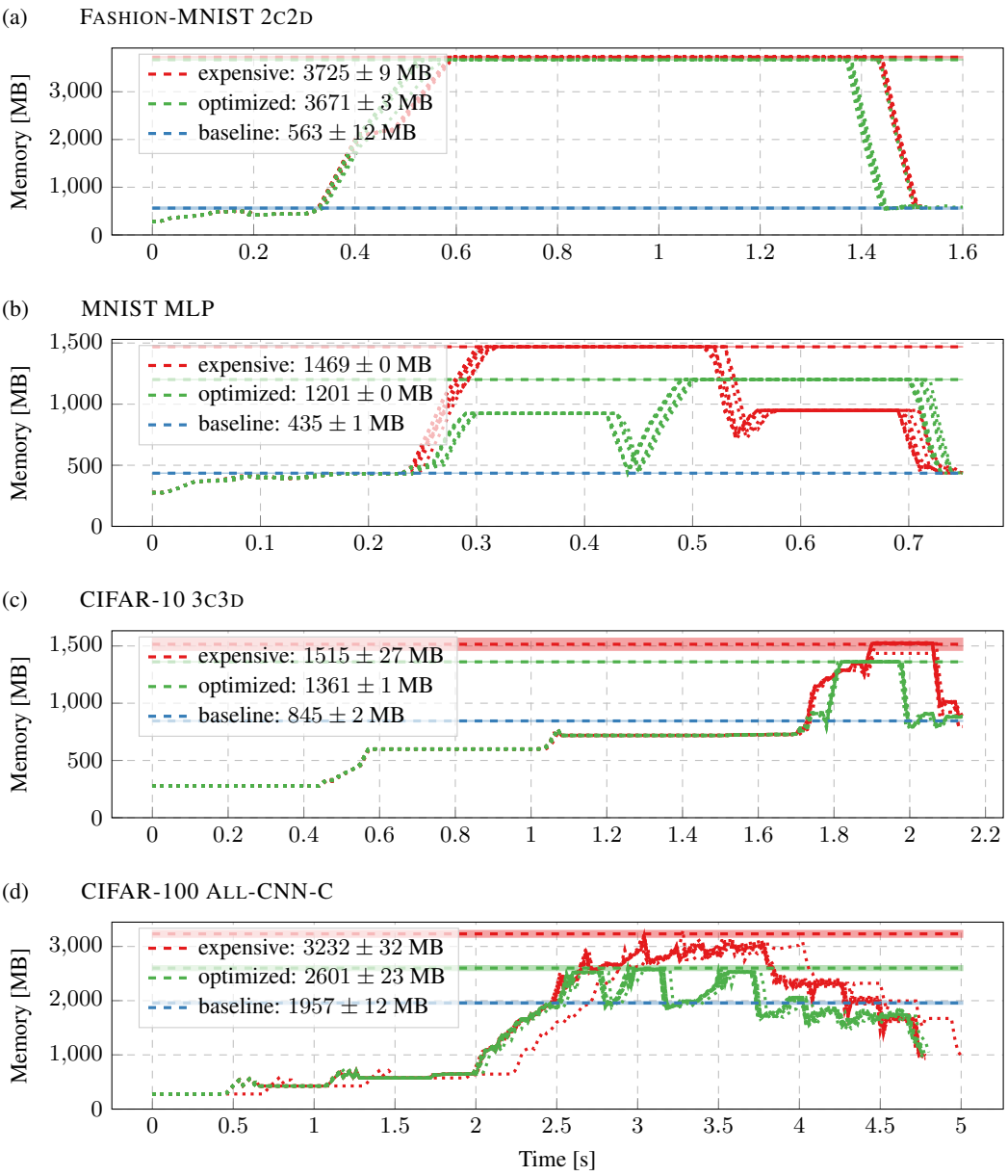


Figure 12: **Memory consumption and savings with hooks** during one forward-backward step on a CPU for different DEEPOBS problems. We compare three settings; i) without COCKPIT (baseline); ii) COCKPIT with `GradHist1d` with BACKPACK (expensive); iii) COCKPIT with `GradHist1d` with BACKPACK and additional hooks (optimized). Peak memory consumptions are highlighted by horizontal dashed bars and shown in the legend. Shaded areas, if visible, fill two standard deviations above and below the mean value, all of them result from ten independent runs. Dotted lines indicate individual runs. Our optimized approach allows to free obsolete tensors during backpropagation and thereby reduces memory consumption. From top to bottom: the effect is less pronounced for architectures that concentrate the majority of parameters in a single layer ((a) $3,274,634$ total, $3,211,264$ largest layer) and increases for more balanced networks ((b) $1,336,610$ total, $784,000$ largest layer, (c): $895,210$ total, $589,824$ largest layer).

E.2 Additional run time benchmarks

E.2.1 Individual instrument overhead

To estimate the computational overhead for individual instruments, we run COCKPIT with that instrument for 32 iterations, tracking at every step. Training proceeds with the default batch size specified by the DEEPOBS problem and uses SGD with learning rate 10^{-3}. We measure the time between iterations 1 and 32, and average for the overhead per step. Every such estimate is repeated over 10 random seeds to obtain mean and error bars as reported in Figure 6a.

Note that this protocol does *not* include initial overhead for setting up data loading and also does *not* include the time for evaluating train/test loss on a larger data set, which is usually done by practitioners. Hence, we even expect the shown overheads to be smaller in a conventional training loop which includes the above steps.

Individual overhead on GPU versus CPU: Figure 13 and Figure 14 show the individual overhead for four different DEEPOBS problems on GPU and CPU, respectively. The left part of Figure 13 (c) corresponds to Figure 6a. Right panels show the expensive quantities, which we omitted in the main text as they were expected to be expensive due to their computational work (`HessMaxEV`) or bottlenecks in the implementation (`GradHist2d`, see Appendix E.3 for details). We see that they are in many cases equally or more expensive than computing all other instruments. Another expected feature of the GPU-to-CPU comparison is that parallelism on the CPU is significantly less pronounced. Hence, we observe an increased overhead for all quantities that contain non-linear transformations and contractions of the high-dimensional individual gradients, or require additional backpropagations (curvature).

E.2.2 Configuration overhead

For the estimation of different COCKPIT configuration overheads, we use almost the same setting as described above, training for 512 iterations and tracking only every specified interval.

Configuration overhead on GPU versus CPU: Figure 15 and Figure 16 show the configuration overhead for four different DEEPOBS problems. The bottom left part of Figure 15 corresponds to Figure 6b. In general, we observe that increased parallelism can be exploited on a GPU, leading to smaller overheads in comparison to a CPU.

COCKPIT can even scale to significantly larger problems, such as a RESNET-50 on IMAGENET-like data. Figure 17 shows the computational overhead for different tracking intervals on such a large-scale problem. Using the *economy* configuration, we can achieve our self-imposed goal of at most doubling the run time even when tracking every fourth step. More extensive configurations (such as the *full* set) would indeed have almost prohibitively large costs associated. However, these costs could be dramatically reduced when one decides to only inspect a part of the network using COCKPIT. Note, individual gradients are not properly defined when using batch norm, therefore, we replaced these batch norm layers with identity layers when using the RESNET-50.

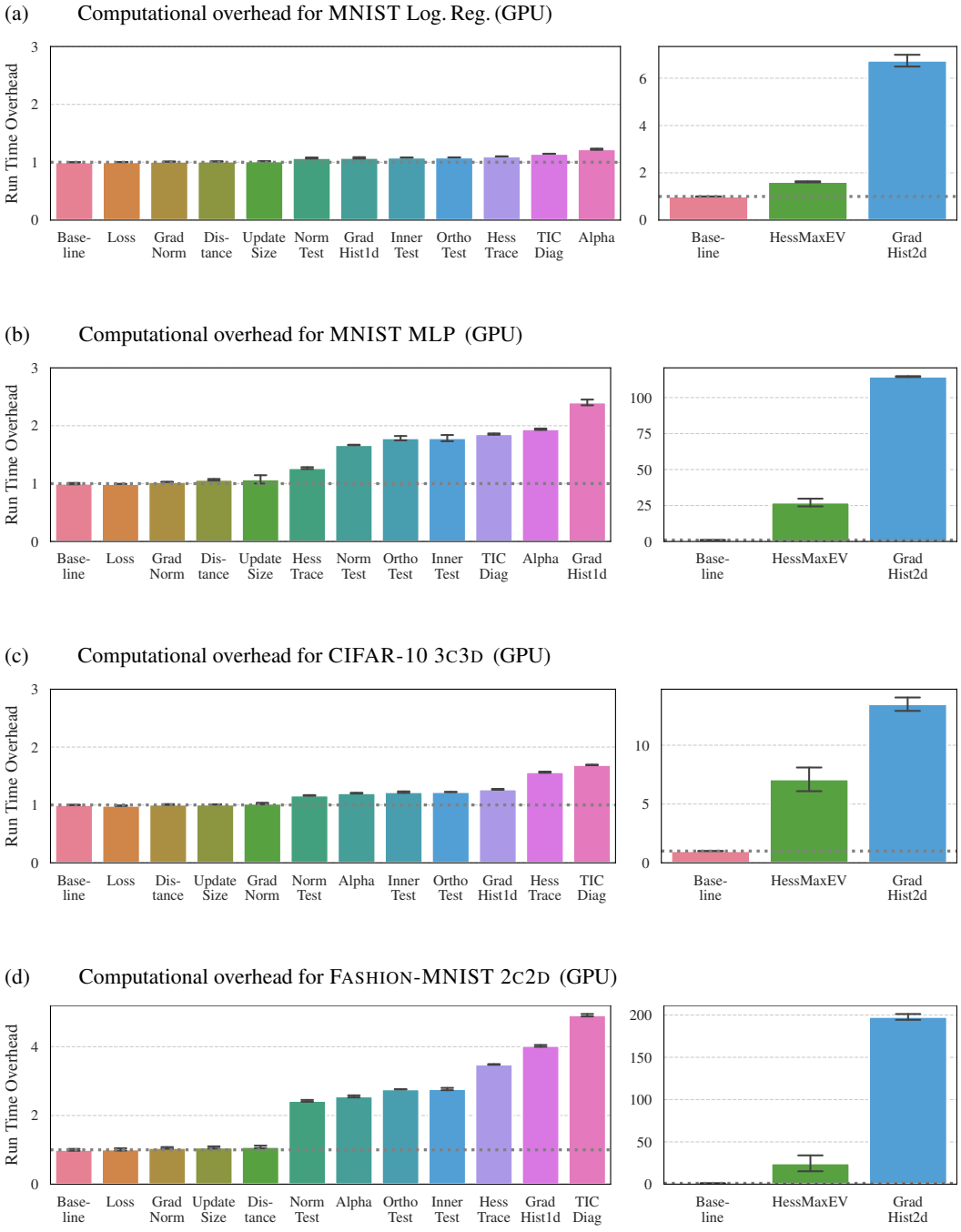


Figure 13: **Individual overhead of COCKPIT's instruments on GPU for four different problems.** All run times are shown as multiples of the *baseline* without tracking. Expensive quantities are displayed in separate panels on the right. Experimental details in the text.

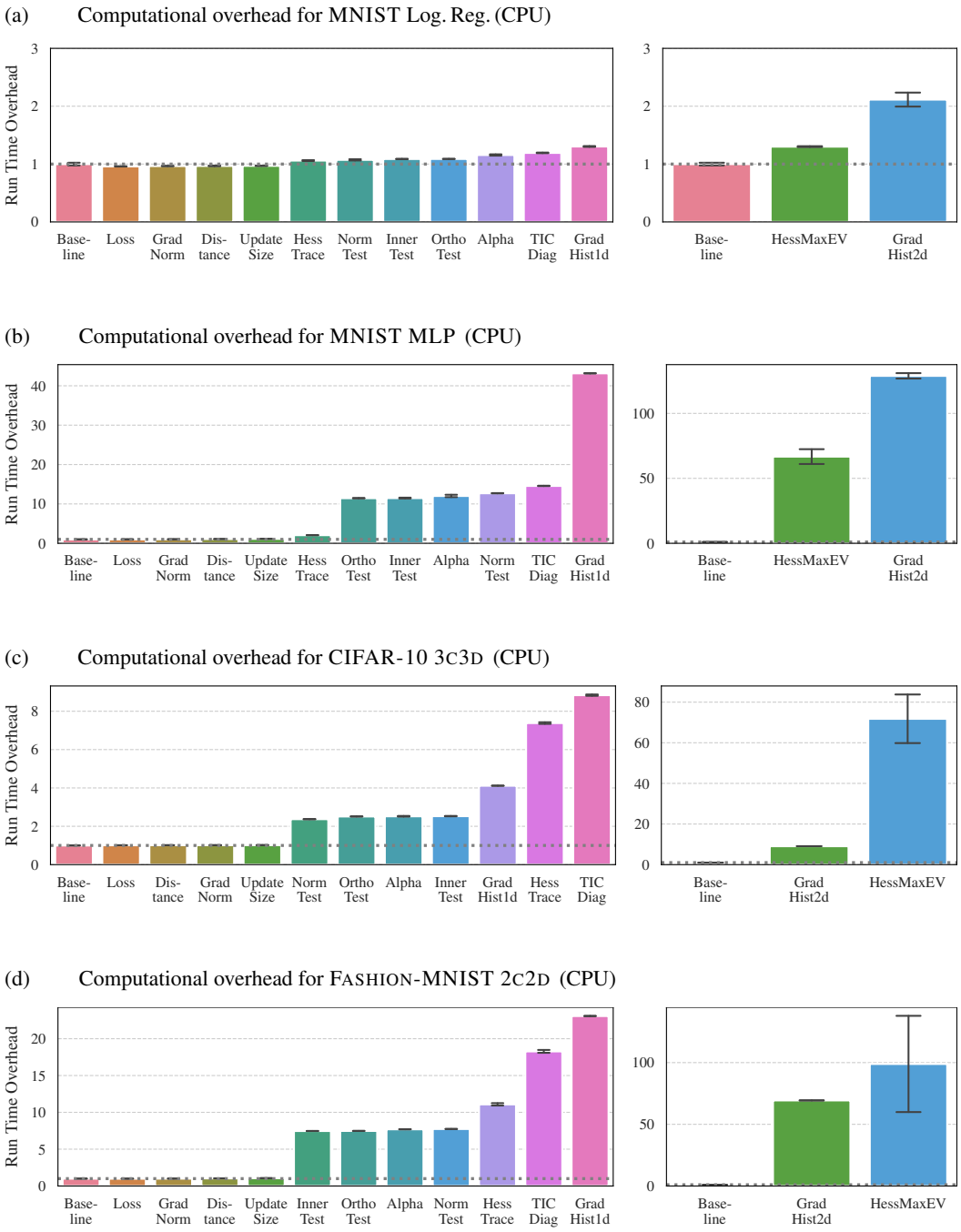


Figure 14: **Individual overhead of COCKPIT's instruments on CPU for four different problems.** All run times are shown as multiples of the *baseline* without tracking. Expensive quantities are displayed in separate panels on the right. Experimental details in the text.

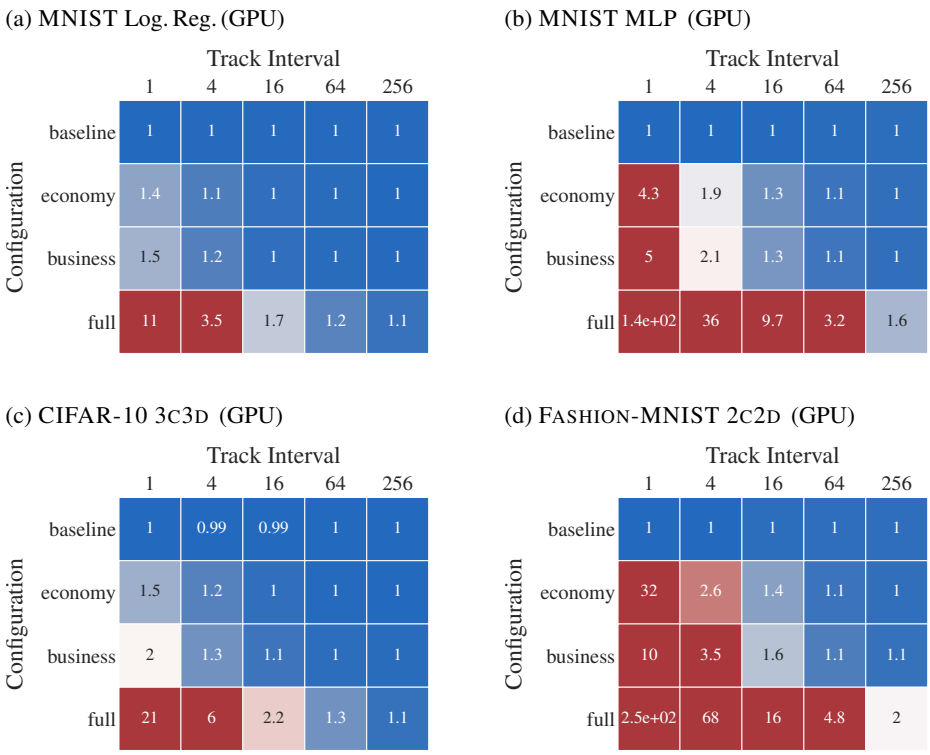


Figure 15: **Overhead of COCKPIT configurations on GPU for four different problems with varying tracking interval.** Color bar is the same as in Figure 6.

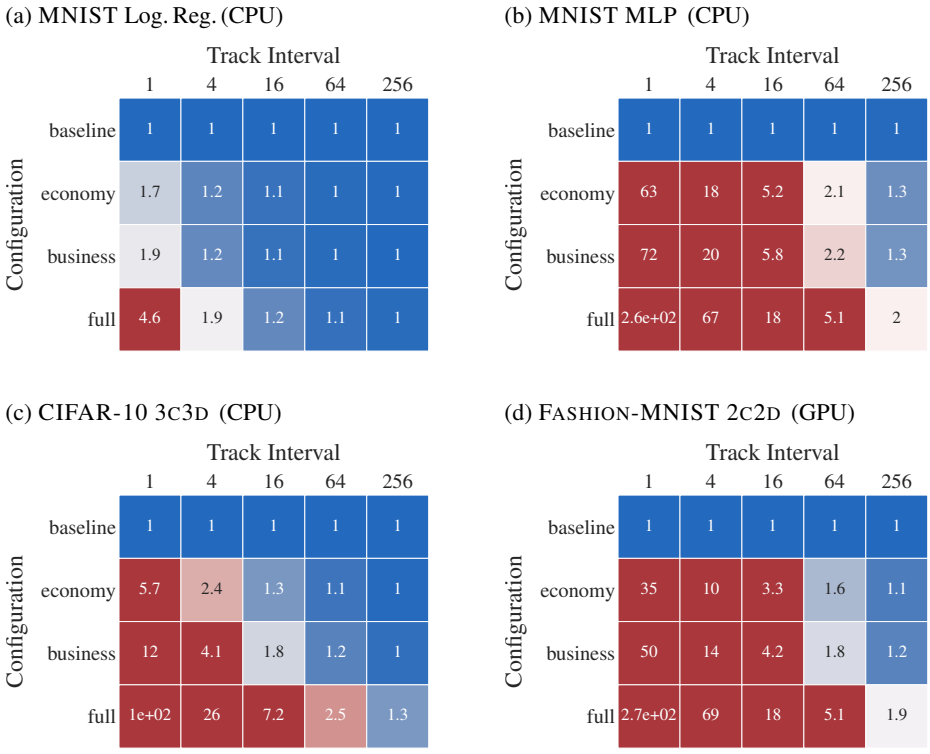


Figure 16: **Overhead of COCKPIT configurations on CPU for four different problems with varying tracking interval.** Color bar is the same as in Figure 6.

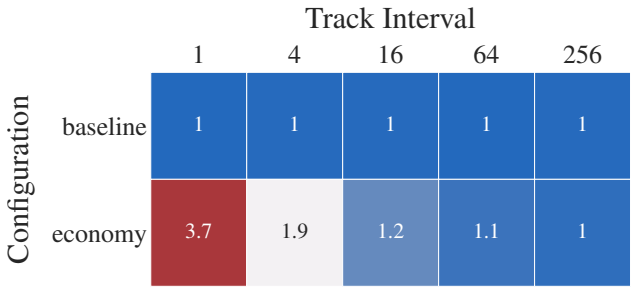


Figure 17: **Overhead of COCKPIT configurations on GPU for RESNET-50 on IMAGENET.** COCKPIT's instruments scale efficiently even to very large problems (here: 1000 classes, $(3, 224, 224)$-sized inputs, and a batch size of 64. For individual gradients to be defined, we replaced the batch norm layers of the RESNET-50 model with identities.) Color bar is the same as in Figure 6.

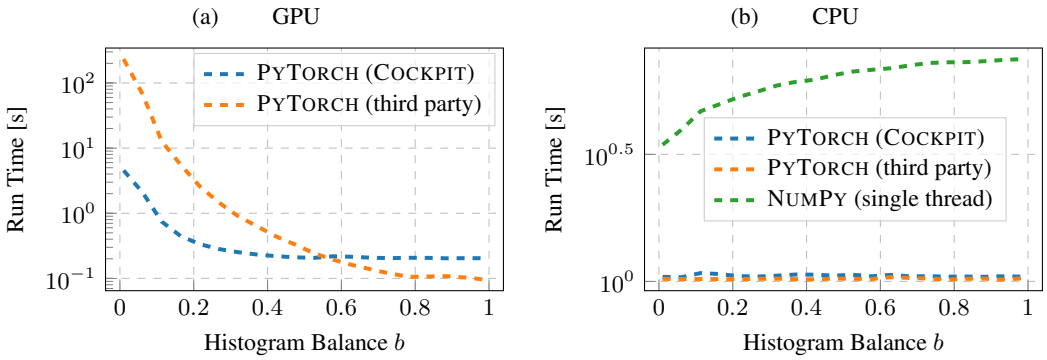


Figure 18: **Performance of two-dimensional histogram GPU implementations depends on the data.** (a) Run time for two different GPU implementations with histograms of different imbalance. Cockpit's implementation outperforms the third party solution by more than one order of magnitude in the deep learning regime ($b \ll 1$). (b) On CPU, performance is robust to histogram balance. The run time difference between NumPy and PyTorch is due to multi-threading. Data has the same size as DeepOBS's CIFAR-10 3c3d problem ($D = 895,210, |\mathcal{B}| = 128$). Curves represent averages over 10 independent runs. Error bars are omitted to improve legibility.

E.3 Performance of two-dimensional histograms:

Both one- and two-dimensional histograms require $|\mathcal{B}| \times D$ elements be accessed, and hence perform similarly. However, we observed different behavior on GPU and decided to omit the two-dimensional histogram's run time in the main text. As explained here, this performance lack is not fundamental, but a shortcoming of the GPU implementation. PyTorch provides built-in functionality for computing one-dimensional histograms at the time of writing, but is not yet featuring multi-dimensional histograms. We experimented with three implementations:

- PyTorch **(third party):** A third party implementation[7] under review for being integrated into PyTorch[8]. It relies on `torch.bincount`, which uses `atomicAdd`s that represent a bottleneck for histograms where most counts are contained in one bin.[9] This occurs often for over-parameterized deep models, as most of the gradient elements are zero.
- PyTorch **(Cockpit):** Our implementation uses a suggested workaround, computes bin indices and scatters the counts into their associated bins with `torch.Tensor.put_`. This circumvents `atomicAdd`s, but has poor memory locality.
- NumPy**:** The single-threaded `numpy.histogram2d` serves as baseline, but does not run on GPUs.

To demonstrate the strong performance dependence on the data, we generate data from a uniform distribution over $[0, b] \times [0, b]$, where $b \in (0, 1)$ parametrizes the histogram's balance, and compute two-dimensional histograms on $[0, 1] \times [0, 1]$. Figure 18 (a) shows a clear increase in run time of both GPU implementations for more imbalanced histograms. Note that even though our implementation outperforms the third party by more than one order of magnitude in the deep neural network regime ($b \ll 1$), it is still considerably slower than a one-dimensional histogram (see Figure 13 (c)), and even slower on GPU than on CPU (Figure 18 (b)). As expected, the CPU implementations do not significantly depend on the data (Figure 18 (b)). The performance difference between PyTorch and NumPy is likely due to multi-threading versus single-threading.

Although a carefully engineered histogram GPU implementation is currently not available, we think it will reduce the computational overhead to that of a one-dimensional histogram in future releases.

[7] Permission granted by the authors of `github.com/miranov25/.../histogramdd_pytorch.py`.
[8] See `https://github.com/pytorch/pytorch/pull/44485`.
[9] See `https://discuss.pytorch.org/t/torch-bincount-1000x-slower-on-cuda/42654`

F COCKPIT view of convex stochastic problems

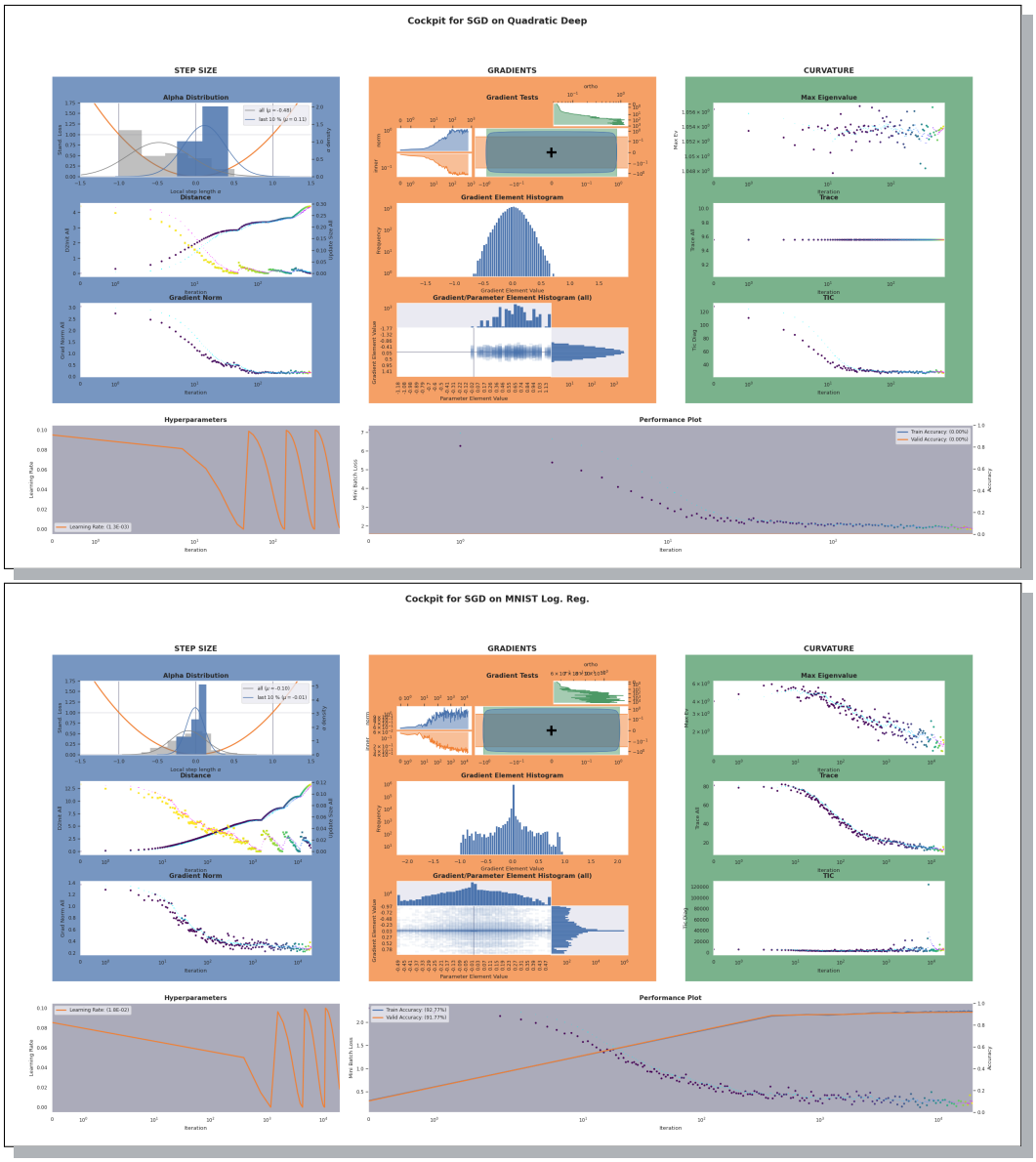


Figure 19: **Screenshot of COCKPIT's full view for convex DEEPOBS problems.** Top COCKPIT shows training on a noisy quadratic loss function. Bottom shows training on logistic regression on MNIST. Figure and labels are not meant to be legible. It is evident, that there is a fundamental difference in the optimization process, compared to training deep networks, i.e. Figure 2. This is, for example, visible when comparing the gradient norms, which converge to zero for convex problems but not for deep learning.