Neural Nets with a Newton Conjugate Gradient Method on Multiple GPUs

Severin Reiz[(✉)] , Tobias Neckel , and Hans-Joachim Bungartz

School of Computation, Information and Technology, Technical University of Munich (TUM), Munich, Germany
`s.reiz@tum.de`

Abstract. Training deep neural networks consumes increasing computational resource shares in many compute centers. Often, a brute force approach to obtain hyperparameter values is employed. Our goal is (1) to enhance this by enabling second-order optimization methods with fewer hyperparameters for large-scale neural networks and (2) to compare optimizers for specific tasks to suggest users the best one for their problem. We introduce a novel second-order optimization method that requires the effect of the Hessian on a vector only and avoids the huge cost of explicitly setting up the Hessian for large-scale networks.

We compare the proposed second-order method with two state-of-the-art optimizers on five representative neural network problems, including regression and very deep networks from computer vision or variational autoencoders. For the largest setup, we efficiently parallelized the optimizers with Horovod and applied it to a 8 GPU NVIDIA A100 (DGX-1) machine with 80% parallel efficiency.

Keywords: Numerical methods · Machine learning · Deep learning · Second-order optimization · Data parallelism

1   Introduction

Machine Learning (ML) is widely used in todays software world: regression or classification problems are solved to obtain efficient models of input-output relationships learning from measured or simulated data. In the context of scientific computing, the goal of ML frequently is to create surrogate models of similar accuracy than existing models but with evaluation runtimes of much cheaper computational costs. Applying an ML technique typically results in an online vs. an offline phase. While the offline phase comprises all computational steps to create the ML model from given data (the so-called training data), the online phase is associated to obtaining desired answers for new data points (typically called validation points). Different types of ML techniques exist: Neural networks in various forms, Gaussian processes which incorporate uncertainty, etc. [1].

For almost all methods, numerical optimization is necessary to tune parameters or hyperparameters of the corresponding method in the offline phase such

that good/accurate results can be obtained in the online phase. Even though numerical optimization is a comparably mature field that offers many solution approaches, the optimization problem associated with real-world large-scale ML scenarios is non-trivial and computationally very demanding: The dimensionality of the underlying spaces is high, the amount of parameters to be optimized is large to enormous, and the cost function (the loss) is typically mathematically complicated being non-convex and possessing many local optima and saddle points in general. Additionally, the performance of a method typically depends not only on the ML approach but also on the scenario of application.

Of the zoo of different optimization techniques, certain first-order methods such as the stochastic gradient descent (SGD) have been very popular and represent the de-facto fallback in many cases. Higher-order methods provide generally nice convergence features since they include more derivative information of the loss function. These methods, however, come at the price of evaluating the Hessian of the problem, which typically is way too costly for real-world large-scale ML scenarios, both w.r.t. setting up and storing the matrix and w.r.t. evaluating the matrix-vector product with standard implementations (e.g., the ResNet50 scenario discussed below has about 16 million degrees of freedom in form of corresponding weights).

In this paper, we analyze a second-order Newton-based optimization method w.r.t. accuracy and computational performance in the context of large-scale neural networks of different type. To cope with challenging costs in such scenarios, we implemented a special variant of a regularized Newton method using the Pearlmutter scheme together with a matrix-free conjugate gradient method to evaluate the effect of the Hessian on a given vector with about twice the costs of a backpropagation itself. All implementations are publicly available and easy to integrate since they rely on TensorFlow Keras code[1]. We compare our proposed solution with existing TensorFlow implementations of the prominent SGD and Adam method for five representative ML scenarios of different categories. In particular, we exploit parallelisation in the optimization process on two different levels: a parallel execution of runs the as well as data parallelism by treating several chunks of data (the so-called batches or mini batches) in parallel. The latter results in a quasi-Newton method where the effect of the Hessian is kept constant for a couple of data points before the next update is computed. Our approach, thus, represents a combination of usability, accuracy and efficiency.

The remainder of this paper is organized as follows. Section 2 lists work in the community that is related to our approach. In Sect. 3, basic aspects of deep neural networks are briefly stated to fix the nomenclature for the algorithmic building blocks we combine for our method. The detailed neural network structures and architectures for the five scenarios to be discussed are discussed in Sect. 4. We briefly describe aspects of the implementation in Sect. 5 and show results for the five neural network scenarios in Sect. 6. Section 7 finally concludes the discussion.

---

[1] https://github.com/severin617/Newton-CG.

2   Related Work

Hessian multiplication for neural networks without forming the matrix was introduced very early [2]; while there are multiple optimization techniques around [3], it gained importance again with *Deep Learning via Hessian-free optimization* [4]. Later, the Kronecker-factored approximate curvature (KFAC) of the Fisher matrix(similar to Gauss-Newton Hessian) was introduced [5]; for *high performance computing*, chainerkfac was introduced [6]. AdaHessian uses the Hutchinson method for adapting learning rate [7], other work involves inexact newton methods for neural networks [8] or a comparison of optimizers [9]. With GOFMM, we performed initial studies on Hessian matrices [10], where later we looked at the fast approximation for a multilayer perceptron [11].

3   Methods

In this section, we first briefly describe the basics of deep neural networks[2] and the peculiarities of the variants we are going to use in the five different scenarios in Sect. 6. Afterwards, we highlight the basic algorithmic ingredients of the reference implementations (SGD and Adam) [1]. Finally, we explain the building blocks of our approach: The Pearlmutter trick and the Newton-CG step.

3.1   Scientific Computing for Deep Learning

Consider a feed-forward deep neural network defined as the parameterized function $f(X, \mathbf{W})$. The function f is composed by vector-valued functions $f^{(i)}$, $i = 1, \ldots, D$, which represent each one layer in the network of depth D, in the following way: $f(x) = f^{(D)}(\ldots f^{(2)}(f^{(1)}(x)))$

The function corresponding to a network layer (d) and the output of the j-th neuron are computed via

$$f^{(d)} = \begin{bmatrix} z_1^{(d)} \\ z_2^{(d)} \\ \vdots \\ z_{M^{(d)}}^{(d)} \end{bmatrix} \text{ and } z_j^{(d)} = \phi\left(\sum_{i=1}^{M^{(d-1)}} (w_{ji}^{(d)} f_i^{(d-1)}) + w_j^0 \right)$$

with activation function ϕ and weights w. All weights w are comprised in a large vector $\mathbf{W} \in \mathbb{R}^n$ which represents a parameter for f. The optimization problem consists now of finding weights $\mathbf{W}$ a given loss function l will be minimized for given training samples X, Y: $\min_{\mathbf{W}} l(X, Y, \mathbf{W})$.

A prominent example of a loss function is the categorical cross-entropy:
$$l_{entr}(X, Y, \mathbf{W}) := - \sum_{l=1}^{\mathbb{N}} y_i \log(f^{(D)}(X, \mathbf{W})) \ .$$

---

[2] For a brief introduction on deep NN, cf. [12].

Note that only the last layer function $f^{(D)}$ of the network directly shows up in the loss, but all layers are indirectly relevant due to the optimization for all weights in all layers.

Optimizers look at stochastic *mini batches* of data, i.e. disjoint collections of data points. The union of all mini batches will represent the whole training data set. The reason for considering data in chunks of mini batches and not in total is that the backpropagation in larger neural networks will face severe issues w.r.t. memory. Hence, the mini batch loss function, where the mini batch is varied in each optimization step in a round-robin manner, is now defined by

$$L(x, y, \mathbf{W}) := -\sum_{i=1}^{\text{batch-size}} y_i \log(f^{(D)}(X, \mathbf{W})) \ .$$

3.2   State-of-the-Art Optimization Approaches

In order to solve the optimization problem (3.1), different first-order methods exist (for a survey, see [1], e.g.). The pure *gradient descent* without momentum computes weights $\mathbf{W}_k$ in iteration k via $\mathbf{W}_k = \mathbf{W}_{k-1} - a_{k-1}\nabla l(\mathbf{W}_{k-1})$ where $\nabla l(\mathbf{W}_{k-1})$ denotes the gradient of the total loss l w.r.t. the weights $\mathbf{W}$.

The *stochastic gradient descent* method (SGD) includes stochasticity by changing the loss function to the input of a specific mini batch of data, i.e. using L instead of l. Each mini batch of data provides a noisy estimator of the average gradient over all data points, hence the term stochastic. Technically, this is realised by switching the mini batches in a round-robin manner to reach over the full dataset (one full sweep is called an epoch; frequently, more than one epoch of iterations is necessary to achieve quality in the optimization).

The family of *Adam* methods updates weight values by enhancing averages of the gradient s_k with estimates of the 1st moment (the mean) and the 2nd raw moment (the uncentered variance). The approach called *AdaGrad* is directly using these estimators:

$$W_{k+1} = W_k - \alpha_k \frac{s_k}{\delta + \sqrt{r_k}} \tag{1}$$

The Adam method corrects for the biases in the estimators by using the estimators $\hat{s}_k = \frac{s_k}{1-\beta_1^k}$ and $\hat{r}_k = \frac{r_k}{1-\beta_2^k}$ instead of s_k and r_k. Good default settings for the tested machine learning problems described in this paper are $a_0 = 0.001$, $\beta_1 = 0.9$, $\beta_2 = 0.999$, and $\delta = 10^{-8}$.

3.3   Proposed 2nd-Order Optimizer

The second-order optimizer implemented and used for the results of this work consists of a Newton scheme with a matrix-free conjugate gradient (CG) solver for the linear systems of equations arising in each Newton step. The effect of the Hessian on a given vector (i.e. a matrix-vector multiplication result) is realised via the so-called Pearlmutter approach and avoids setting up the Hessian explicitly.

Pearlmutter Approach. The explicit setup of the Hessian is memory-expensive due to the quadratic dendence on the problem size; e.g., a 16M×16M matrix requires about 1 TB of memory. We can obtain "cheap" access to the problems curvature information by computing the Hessian-vector product. This method is called Fast Exact Multiplication by the Hessian H (see [2], e.g.). Specifically, it computes the Hessian vector product Hs for any s in just **two** (instead of the number of weights n) backpropagations (i.e. automatic differentiations for 1st derivative components). For our formulation of the problem this is defined as:

$$H_L(\mathbf{W})s = \begin{pmatrix} \sum_{i=1}^{n} s_i \frac{\delta^2}{\delta w_1 \delta w_i} L(\mathbf{W}) \\ \sum_{i=1}^{n} s_i \frac{\delta^2}{\delta w_2 \delta w_i} L(\mathbf{W}) \\ \vdots \\ \sum_{i=1}^{n} s_i \frac{\delta^2}{\delta w_n \delta w_i} L(\mathbf{W}) \end{pmatrix} = \begin{pmatrix} \frac{\delta}{\delta w_1} \sum_{i=1}^{n} s_i \frac{\delta}{\delta w_i} L(\mathbf{W}) \\ \frac{\delta}{\delta w_2} \sum_{i=1}^{n} s_i \frac{\delta}{\delta w_i} L(\mathbf{W}) \\ \vdots \\ \frac{\delta}{\delta w_n} \sum_{i=1}^{n} s_i \frac{\delta}{\delta w_i} L(\mathbf{W}) \end{pmatrix} = \nabla_w(\nabla_w L(\mathbf{W}) \cdot s)$$

The resulting formula is both efficient and numerically stable [2]. This results in the Algorithm 1, denoted **Pearlmutter** in the implementation.

Newton's Method. Recall the Newton equation

$$H_L(\mathbf{W}^k)d^k = -\nabla L(\mathbf{W}^k)$$

for the network loss function $L : \mathbb{R}^n \to \mathbb{R}$, where $\mathbf{W}$ is the vector of network weights and $\mathbf{W}^k$ the current iterate of Newton's method to solve for the update vector d^k. The size of the Hessian is $\mathbb{R}^{n \times n}$ which becomes infeasible to store with state-of-the-art weight parameter ranges of ResNets (or similar).

Since frequently the Hessian has a high condition number, which implies near-singularity and provokes imprecision, one would apply regularization techniques to counteract a bad condition. A common choice is Tikhonov regularization. To this end, a multiple of the unit matrix is added to the Hessian of the loss function such that the regularized system is given by

$$(H(\mathbf{W}^k) + \tau I)d^k = H(\mathbf{W}^k)d^k + \tau I d^k = -\nabla L(\mathbf{W}^k) \tag{2}$$

Note that for large τ the solution will converge to a fraction of the negative gradient $\nabla L(\mathbf{W}^k)$, similar to a stochastic gradient descent method. The regularized Newton method is summarized in Algorithm 2.

Algorithm 1. Pearlmutter

Require: $X, Y, \mathbf{W}, s$:   Compute $H_L s = \nabla_w(\nabla_w L(\mathbf{W}) \cdot s)$
Require: W_0:   Initial estimate for $\mathbf{W}$.
1: $g_0 \leftarrow \text{gradient}(L(\mathbf{W}))$ ▷ Back-Prop
2: intermediate $\leftarrow \text{matmul}(g_0, s)$ ▷ Matrix-Multiplication
3: $Hs \leftarrow \text{gradient}(\text{intermediate})$ ▷ Back-Prop
4: **return** Hs

Algorithm 2. Newton Step

Require: $\mathbf{W}_0$:   Starting point
Require: τ: Tikhonov regularization/damping factor
1: $k \leftarrow 0$
2: **while** $\mathbf{W}_k$ not converged **do**
3:     $k \leftarrow k + 1$
4:     $p_k \leftarrow \text{CG}((H + \tau I), -\nabla L(\mathbf{W}_k))$ # Approx $(H + \tau I) p_k = -\nabla L(\mathbf{W}_k)$
5:     **if** $\nabla L(\mathbf{W}_k)^\top p_k > \tau$ **then** $p_k \leftarrow -\nabla L(\mathbf{W}_k)$ # Feasibility check.
6:     **end if**
7:     $\alpha_k \leftarrow \alpha$ # Compute with lr-scheduler or use a given step size.
8:     $\mathbf{W}_k \leftarrow \mathbf{W}_{k-1} + \alpha_k p_k$
9: **end while**

Conjugate Gradient Step. Since **Pearlmutter** realises a matrix-vector product without setting up the full matrix, we employ an iterative solver that requires matrix products only. We therefore employ a few (inaccurate) CG-iterations to solve Newton's regularized Eq. (2), resulting in an approximated Newton method. The standard CG-algorithm is e.g. described in [13]; note that no direct matrix-access is required since CG relies only on products of vectors.

Complexity: The method described above requires $O(bn)$ operations for the evaluation of the gradient, where n is the number of network weights and b is the size of the mini batch. In addition, for the evaluation of the Hessian product and the solution of the Newton-like equation $O(2mbn)$ is needed, where m is the number of iterations conducted by the CG solver until a sufficient approximation to the solution is reached. Although the second-order optimizer requires more work than ordinary gradient descent, it may still be beneficial since, under the conditions that it promises local q-superlinear convergence, where $\mathbf{W}^\star$ is a local minimizer (see [14]), i.e. $\exists \gamma \in (0,1), l \geq 0$, such that

$$||\mathbf{W}^{k+1} - \mathbf{W}^*|| \leq \gamma ||\mathbf{W}^k - \mathbf{W}^*|| \quad \forall k > l .$$

4   Scenarios and Neural Network Architectures

In this section, we briefly outline the different neural network structures for the five different ML scenarios used in Sect. 6. Those networks share the general structure outlined in Sect. 3.1 but differ in details considerably.

Regressional Analysis: Most regression models connect the input X with some parametric function f to the output Y, including some error ϵ, i.e. $Y = f(X, \beta) + \epsilon$. The goal is find $\mathbf{W}$ to minimize the loss function which here is the sum of the squared error for all samples i in the training data set

$$l = \sum_{i=1}^{N} (y_i - f(x_i, \mathbf{W}))^2 .$$

Variational Autoencoder: A variational autoencoder (VAE) consists of two coupled but independently parametrized components: The encoder compresses the sampled input X into the latent space. The decoder receives as input the information sampled from the latent space and produces x' as close as possible to X. In a variational autoencoder, encoder and decoder are trained simultaneously such that output X' minimizes a reconstruction error to X by the Kullback-Leibler divergence. For details on VAEs, see [15], e.g.

Bayesian Neural Network: One of the biggest challenges in all areas of machine learning is deciding on an appropriate model complexity. Models with too low complexity will not fit the data well, while models possessing high complexity will generalize poorly and provide bad prediction results on unseen data, a phenomenon widely known as overfitting. Two commonly deployed strategies to counteract this problem are hold-out or cross-validation on one hand, where part of the data is kept from training in order to optimize hyperparameters of the respective model that correspond to model complexity, and controlling the effective complexity of the model by inducing a penalty term on the loss function on the other hand. The latter approach is known as regularization and can be implemented by applying Bayesian techniques on neural networks [16].

Let $\theta, \epsilon \sim N(0, 1)$ be random variables, $w = t(\theta, \epsilon)$, where t is a deterministic function. Moreover, let $w \sim q(w|\theta)$ be normally distributed. Then our optimization task where the loss function l is the log-likelihood reads [17]

$$l(w, \theta) = \log q(w|\theta) - \log P(D|w) - \log P(w) .$$

Convolutional Neural Network (CNN): In general, the convolution is an operation on two functions I, K, defined by

$$S(t) = (I * K)(t) = \int I(a)K(t - a)da .$$

If we use a 2D image I as input with a 2D kernel K, we obtain a two-dimensional discrete convolution $S(i, j) = (I * K)(i, j) = \sum_x \sum_y I(x, y)K(i - x, j - y)$.

Color images additionally have at least a channel for red, blue and green intensity at each pixel position. Assume that each image is a 3D-tensor and $V_{i,j,k}$ describes the value of channel i at row j and column k. Then let our kernel be a 4D-tensor with $K_{l,i,j,k}$ denoting the connection strength (weight) between a unit in input channel i and output channel l at an offset of k rows and l columns between input and output. CNNs apply, besides other incredients, convolution kernels of different size in different layers in a sliding window approach to extract features. For a brief introduction to CNN, see [12], e.g. As an example, the prominent ResNet 50 network structure consists of 50 layers of convolutions or other layers, with *skip connections* to avoid the problem of diminishing gradients.

Transfer Learning: Transfer learning (TL) deals with applying already gained knowledge for generalization to a different, but related domain [18]. Creating a separate, labeled dataset of sufficient size for a specific task of interest in the context of image classification is a time-consuming and resource-intensive process. Consequently, we find ourselves working with sets of training data that

are significantly smaller than other renowned datasets, such as CIFAR and ImageNet [19]. Moreover, the training process itself is time-consuming too and relies on dedicated hardware. Since modern CNNs take around 2–3 weeks to train on ImageNet in a professional environment, starting this process from scratch for every single model is hardly efficient. Therefore, general pretrained networks are typically used which are then tailored to specific inputs.

5   Implementation

5.1   Automatic Differentiation Framework

The `Newton-CG` optimization strategy is independent of the implementation, and of course, is suitable in any setting where second-order is beneficial **(1)** and storing Hessians is infeasible w.r.t. memory consumption **(2)**. However, one needs a *differentiation* framework. During the course of the work, a custom auto-encoder (and similar) implementation with optimized matrix operations [10] became difficult w.r.t. the automatic differentiation (especially with convolutions), so we decided to move to a prominent framework, *TensorFlow*. The TensorFlow programming model consists of two main steps: (1) Define computations in form of a "stateful dataflow graph" and (2) execute this graph. At the heart of model training in TensorFlow lies the optimizer; like `Adam` or `SGD`, `newton-cg` uses inheritance from the class `tf.python.keras.optimizer_v2.Optimizer_v2`. The base class handles the two main steps of optimization: `compute_gradients()` and `apply_gradients()`. When applying the gradients, for each variable that is optimized, the method `resource_compute_dense(grad, var)` is called with the variable and its (earlier computed) gradient. In this method, the algorithm update step for this variable is computed. It has to be overwritten by any subclassing optimizer. We implemented two versions of our optimizer: one inheriting from the optimizer in tf.train and one inheriting from the Keras Optimizer_v2. The constructor accepts the learning rate as well as the `Newton-CG` hyperparameters: regularization factor τ, the `CG`-convergence-tolerance and the maximum number of CG iterations. Internally, the parameters are converted to tensors and stored as python object attributes. The main logic happens in the above mentioned `resource_compute_dense(grad, var)` method[3]. Table 1 lists the five ML scenarios and their implementation used to generate the results below.

5.2   Data Parallelism

In order to show the applicability of the proposed second-order optimizer for real-world large-scale networks, it was necessary to parallelize optimization computations to obtain suitable runtimes. We decided to use the comparably simple and prominent strategy of data parallelism. Data-parallel strategies t distribute data across different compute units, and each unit operates on the data in parallel. So in our setting, we compute different Newton-CG steps on i different

---

[3] See the implementation in https://github.com/severin617/Newton-CG/blob/main/newton_cg/newton_cg.py#L127.

Table 1. Five ML scenarios with different neural network structures.

Scenario	Description
reg-lif	one-layer life expectancy prediction[a]
reg-bos	two-layer boston housing price projection with keras [b]
vae-mnist	variational autoencoder from Keras [c]
bnn-mnist	Bayesian neural network with tensorflow-probability[d] [20]
resnet	ResNet architecture from Keras [e]

[a]https://valueml.com/predicting-the-life-expectancy-using-tensorflow/.
[b]https://www.kaggle.com/code/prasadperera/the-boston-housing-dataset.
[c]https://keras.io/examples/generative/vae/.
[d]https://www.tensorflow.org/probability/.
[e]https://www.tensorflow.org/api_docs/python/tf/keras/applications/resnet50/ResNet50.

mini-batches in parallel, and the resulting update vectors are accumulated using an Allreduce. Note that this is different to e.g. a i-times as big batch or i-times as many steps since this would use an updated weight when computing gradient information via backpropagations. In a smoothly defined function, this could converge to a similar minimum, however due to stochasticity this may not.

Horovod is a data-parallel distributed training framework (open source) for TensorFlow, Keras, PyTorch, and Apache MXNet, that scales a training script up to many GPUs using MPI [21]. We apply Horovod for the data parallelisation of the second-order Newton-CG approach. In a second step the whole algorithms could be parallelized, this would then be model parallelism. The following matrix summarizes the data and model parallelism in the context of neural network optimization.

Data parallelism	Model parallelism
Operations performed on different batches of data	Parallel operations performed on same data (in identical batch)

5.3 Software and Hardware Setup

Training with Keras and Horovod was used to show applicability and scalability of the proposed second order optimization. The ResNets for computer vision were pretrained for 200 epochs to improve second-order convergence, see Fig. 1 (f), with training data from the Imagenet Large Scale Visual Recognition Challenge 2012 (ILSVRC2012)[4]. Test runs were performed on the *Leibniz Rechenzentrum (LRZ)* AI System DGX-1 A100 Architecture with 8 NVIDIA Tesla A100 and 80 GB per GPU.

---

[4] Following parameters were utilized in the pretraining: training/val-batch-size: 64, learning-rate: 0.001, momentum: 0.9, weight-decay: 0.00005. After each step, ten validation steps were used to calculate the top_5 accuracy, resulting in a final loss of 4.5332 and a final top_5 accuracy of 0.6800 after 2e5 steps.

6 Results

6.1 Accuracy Results for Different Scenarios

In this study, we applied the Newton-CG method as well as the two state-of-the-art methods SGD and Adam for the five different network architectures and specific scenarios described in Sect. 4 and Table 1 to evaluate the performance for each case and obtain insight into potential patterns. We show the detailed optimization behavior in Fig. 1 while the final validation loss optimum is summarized in Fig. 2. A similar comparison figure was used in [9] highlighting a similar insight that it is hard to predict the performance of different optimizers for considerably different scenarios.

One can observe significant benefits of the 2nd-order Newton-CG in regression models, be it the life expectancy prediction or the boston housing data regression. We believe this is mostly due to the continuity in loss/optimization, whereas in the other scenarios this could jump, due to mini batches and classification.

The variatonal autoencoder seems to work better with the conventional optimizers. Our hope was that due to the continuous behaviour we may see some benefits. However, this is also very hyper-parameter dependent, and the conventional methods have to be considerably tuned for that. In the Bayesian Neural Network we see benefits of Newton-CG especially against Adam.

We observe hardly any benefits of 2nd-order optimization for the ResNet50 model. While at first we follow the near-optimal training curve, Newton-CG moves away from the minimum. One problem could be that we work with a fixed learning rate. This could be tuned with a *learning-rate-scheduler*, which we currently work on.

6.2 Parallel Runs

Exploiting parallelism allows for distributing work in case of failures (e.g. resilience), usage of modern compute architectures with accelerators, and ultimately, lower *time-to-solution*. All network architectures shown before can be run in parallel, in the data parallel approach explained in Sect. 5.2.

For the following measurements, we ran the ResNet50 model on the DGX-1 partition of the *LRZ*, since it is our biggest network model and therefore, allows for the biggest parallelism gains (see Table 2).[5] Note that the batch size is reduced with *GPUs*, in order to account for a similar problem to be solved when increasing the amount of *workers*. However, it cannot be fully related to *strong scaling*, since the algorithm changes as explained in Sect. 5. In a parallel setup, the loss is calculated for a smaller *mini-batch* and then the *update* is accumulated. This is different to looking at a bigger *batch*, since the loss function is a different one (computed for *mini-batch* per *GPU* only).
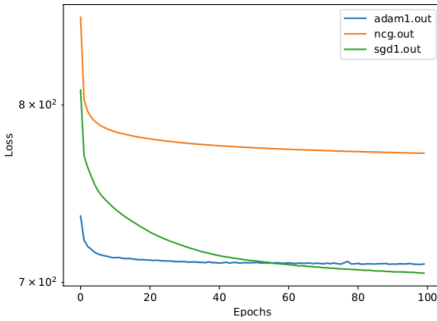
---

[5] On the LRZ cluster, we had to reduce to 20% training images for lower memory disk usage and 60% of the optimization layers, 30 layers for ResNet-50.
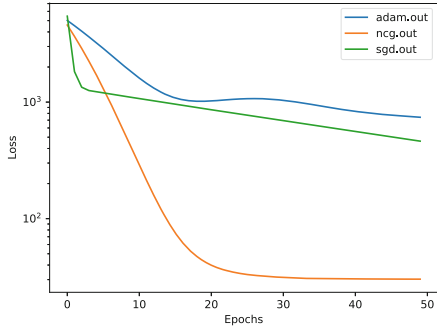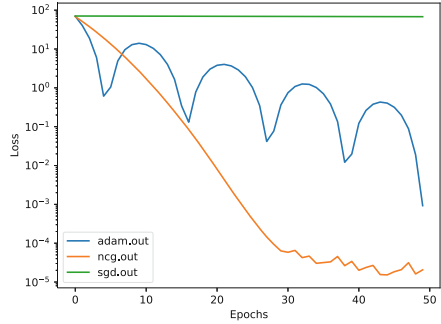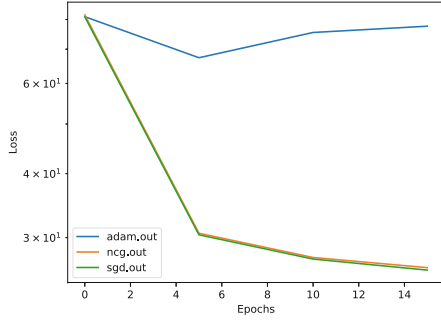
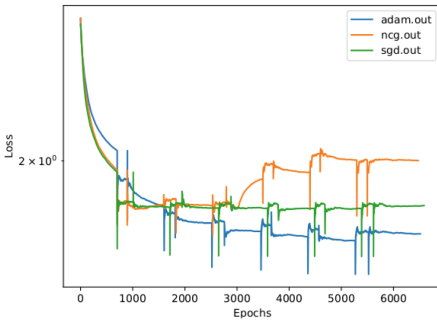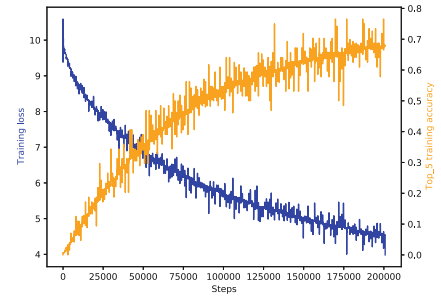**Fig. 1.** Results for the training loss ((a), (b) and (f)) and the validation loss ((c)–(e)) for the three compared methods: SGD in green, Adam in blue and Newton-CG in orange. The methods have been applied to the five different ML scenarios with corresponding different neural network structure: (a) regression case for life expectancy prediction, (b) regression for boston housing dataset, (c) Variational Auto Encoder with MNIST, (d) Bayesian Neural Network with MNIST, (e) ResNet50 with ImageNet, and (f) the corresponding sgd pretraining run ("steps" corresponds to epochs). (Color figure online)
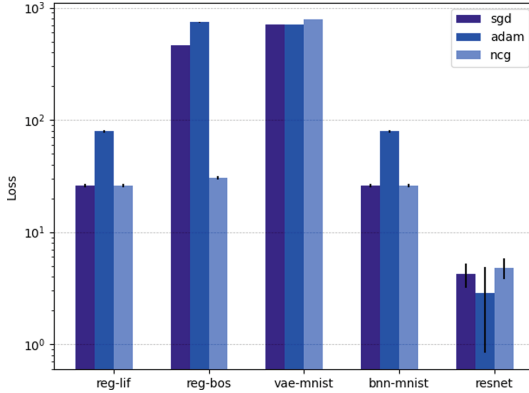
Fig. 2. Final loss value of each optimizer for the five different neural network architectures and scenarios.

Table 2. Newton-CG runtimes per epoch with batch-size 512, ResNet-50 on ImageNet

	1 GPU	2 GPUs	4 GPUs	8 GPUs
A100 runtime	238 s	121 s	65 s	37 s
A100 parallel efficiency	100%	98.3%	91.5%	80.4%

Similarly, we conducted the performance study on a single GPU for the two other optimizers. *SGD* and *Adam* take 238 s and 242 s per epoch, resp., showing similar runtimes as *Newton-CG* with 238 s. We believe that for this big scenario the runtime is dominated by memory transfer and the one vs. two backpropagations hardly makes a difference.

7 Conclusion and Future Work

In conclusion, we found benefits of second-order curvature information plugged into the optimization of the neural network weights especially for regression cases, but not much benefits in classification scenarios. In order to improve for classification, we experimented with a cyclical learning rate scheduler for ResNets for computer vision and Natural Language Processing, but more studies need to be investigated. The data-parallel approach seems to work well in performance numbers, since we reach about 80% parallel efficiency for 8 A100 GPUs.

For showcasing purposes, you may also try the frontend android application TUM-lens[6], where some models have been trained with Newton-CG.

---

[6] https://play.google.com/store/apps/details?id=com.maxjokel.lens.

References

1. Goodfellow, I., Bengio, Y., Courville, A.: Deep Learning. MIT Press, Cambridge (2016). http://www.deeplearningbook.org
2. Pearlmutter, B.A.: Fast exact multiplication by the Hessian. Neural Comput. **6**(1), 147–160 (1994)
3. Nocedal, J., Wright, S.J.: Numerical Optimization. Springer, Heidelberg (1999)
4. Martens, J., et al.: Deep learning via Hessian-free optimization. In: ICML, vol. 27, pp. 735–742 (2010)
5. Martens, J.: Second-order optimization for neural networks. University of Toronto (Canada) (2016)
6. Osawa, K., Tsuji, Y., Ueno, Y., Naruse, A., Yokota, R., Matsuoka, S.: Large-scale distributed second-order optimization using kronecker-factored approximate curvature for deep convolutional neural networks. In: Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition, pp. 12359–12367 (2019)
7. Yao, Z., Gholami, A., Shen, S., Mustafa, M., Keutzer, K., Mahoney, M.W.: Adahessian: an adaptive second order optimizer for machine learning. arXiv preprint arXiv:2006.00719 (2020)
8. O'Leary-Roseberry, T., Alger, N., Ghattas, O.: Inexact newton methods for stochastic nonconvex optimization with applications to neural network training. arXiv preprint arXiv:1905.06738 (2019)
9. Schmidt, R.M., Schneider, F., Hennig, P.: Descending through a crowded valley-benchmarking deep learning optimizers. In: International Conference on Machine Learning, pp. 9367–9376. PMLR (2021)
10. Chenhan, D.Y., Reiz, S., Biros, G.: Distributed-memory hierarchical compression of dense SPD matrices. In: SC 2018: International Conference for High Performance Computing, Networking, Storage and Analysis, pp. 183–197. IEEE (2018)
11. Chen, C., Reiz, S., Yu, C.D., Bungartz, H.-J., Biros, G.: Fast approximation of the Gauss-Newton Hessian matrix for the multilayer perceptron. SIAM J. Matrix Anal. Appl. **42**(1), 165–184 (2021)
12. Lecun, Y., Bengio, Y., Hinton, G.: Deep learning. Nature **521**(7553), 436 (2015)
13. Shewchuk, J.R., et al.: An introduction to the conjugate gradient method without the agonizing pain (1994)
14. Suk, J.: Application of second-order optimisation for large-scale deep learning. Masterarbeit, TUM (2020)
15. Kingma, D.P., Welling, M.: An introduction to variational autoencoders. Found. Trends® Mach. Learn. **12**(4), 307–392 (2019)
16. Bishop, C.M., et al.: Neural Networks for Pattern Recognition. Oxford University Press, Oxford (1995)
17. Blundell, C., Cornebise, J., Kavukcuoglu, K., Wierstra, D.: Weight uncertainty in neural network. In: International Conference on Machine Learning, pp. 1613–1622. PMLR (2015)
18. Yosinski, J., Clune, J., Bengio, Y., Lipson, H.: How transferable are features in deep neural networks? In: Advances in Neural Information Processing Systems, vol. 27 (2014)
19. Krizhevsky, A., Hinton, G., et al.: Learning multiple layers of features from tiny images. Master's thesis, University of Tront (2009)
20. Weigold, H.: Second-order optimization methods for Bayesian neural networks. Masterarbeit, Technical University of Munich (2021)
21. Sergeev, A., Del Balso, M.: Horovod: fast and easy distributed deep learning in TensorFlow. arXiv preprint arXiv:1802.05799 (2018)