$$\begin{tikzcd} & \text{Clk} \\ \bigcirc \arrow[ur] \arrow[dr] & \\ & \text{no Clk.} \end{tikzcd}$$

→ Scale made 1 feed
→ learning
→

# On the Factory Floor: ML Engineering for Industrial-Scale Ads Recommendation Models

Rohan Anil, Sandra Gadanho, Da Huang, Nijith Jacob, Zhuoshu Li, Dong Lin,
Todd Phillips, Cristina Pop, Kevin Regan, Gil I. Shamir, Rakesh Shivanna, Qiqi Yan
Google Inc.

## ABSTRACT

For industrial-scale advertising systems, prediction of ad click-through rate (CTR) is a central problem. Ad clicks constitute a significant class of user engagements and are often used as the primary signal for the usefulness of ads to users. Additionally, in cost-per-click advertising systems where advertisers are charged per click, click rate expectations feed directly into value estimation. Accordingly, CTR model development is a significant investment for most Internet advertising companies. Engineering for such problems requires many machine learning (ML) techniques suited to online learning that go well beyond traditional accuracy improvements, especially concerning efficiency, reproducibility, calibration, credit attribution. We present a case study of practical techniques deployed in Google's search ads CTR model. This paper provides an industry case study highlighting important areas of current ML research and illustrating how impactful new ML methods are evaluated and made useful in a large-scale industrial setting.

## 1 INTRODUCTION

Ad click-through rate (CTR) prediction is a key component of online advertising systems that has a direct impact on revenue, and continues to be an area of active research [33, 42, 45, 74]. This paper presents a detailed case study to give the reader a "tour of the factory floor" of a production CTR prediction system, describing challenges specific to this category of large industrial ML systems and highlighting techniques that have proven to work well in practice.

The production CTR prediction model consists of billions of weights, trains on more than one hundred billion examples, and is required to perform inference at well over one hundred thousand requests per second. The techniques described here balance accuracy improvements with training and serving costs, without adding undue complexity: the model is the target of sustained and substantial R&D and must allow for effectively building on top of what came before.

100,000 req/s

### 1.1 CTR for Search Ads Recommendations

The recommender problem surfaces a result or set of results from a given corpus, for a given initial context. The initial context may be a user demographic, previously-viewed video, search query, or other. Search advertising specifically looks at matching a **query** $q$ with an **ad** $a$. CTR models for recommendation specifically aim to predict the probability $P(click|x)$, where the input $x$ is an ad-query pair $\langle a, q \rangle$, potentially adorned with additional factors affecting CTR, especially related to **user interface**: how ads will be positioned and rendered on a results page (Section 6).

Beyond surfacing maximally useful results, recommender systems for ads have important additional **calibration** requirements. Actual click labels are stochastic, reflecting noisy responses from users. For any given ad-query $x_i$ and binary label $y_i$, we typically hope to achieve precisely $P(click|x_i) := \mathbb{E}_{\langle x_i, y_i \rangle \sim D}[y_i = click|x_i]$ over some sample of examples $D$ (in test or training). While a typical log-likelihood objective in supervised training will result in zero aggregate calibration bias across a validation set, per-example bias is often non-zero.

Ads pricing and allocation problems create the per-example calibration requirement. Typically, predictions will flow through to an auction mechanism that incorporates bids to determine advertiser pricing. Auction pricing schemes (e.g, VCG [63]) rely on the relative value of various potential outcomes. This requires that predictions for all potential choices of $x$ be well calibrated with respect to each other. Additionally, unlike simple recommenders, ads systems frequently opt to show no ads. This requires estimating the value of individual ads relative to this "null-set" of no ads, rather than simply maximizing for ad relevance.

Consider a query like "yarn for sale"; estimated CTR for an ad from "yarn-site-1.com" might be 15.3%. Estimated CTR for an ad from "yarn-site-2.com" might be 10.4%. Though such estimates can be informed by the semantic relevance of the websites, the requirements for precision are more than what one should expect from general models of language. Additionally, click-through data is highly non-stationary: click prediction is fundamentally an online recommendation problem. An expectation of 15.3% is not static ground truth in the same sense as, for example, translation or image recommendation; it is definitively more subject to evolution over time.

### 1.2 Outline

For ads CTR predictors, minor improvements to model quality will often translate into improved user experience and overall ads system gains. This motivates continuous investments in model research and development. Theoretical and benchmark improvements from ML literature rarely transfer directly to problem-dependent settings of real-world applications. As such, model research must be primarily empirical and experimental. Consequently, a great deal of attention must be paid to the machine costs of model training experiments while evaluating new techniques. In Section 2 we first

*ORSUM @ ACM RecSys, September 23, 2022, Seattle, WA*
© 2022 Association for Computing Machinery.
ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

give a general overview of the model and training setup; Section 3 then discusses **efficiency** concerns and details several successfully deployed techniques. In Section 4, we survey applications of modern ML techniques targeted at improving measures of **accuracy** and geared explicitly toward very-large-scale models. Section 4.4 summarizes empirical results roughly characterizing the relative impact of these techniques.

Deep neural networks (DNNs) provide substantial improvements over previous methods in many applications, including large-scale industry settings. However, non-convex optimization reveals (and exacerbates) a critical problem of prediction: **irreproducibility** [22, 26, 56–59]. Training the same model twice (identical architecture, hyper-parameters, training data) may lead to metrics of the second model being very different from the first. We distinguish between **model irreproducibility**, strictly related to predictions on fixed data, and **system irreproducibility**, where a deployed irreproducible model affects important system metrics. Section 5 characterizes the problem and describes improvements to model irreproducibility.

An effective click prediction model must be able to **generalize across different UI treatments**, including: where an ad is shown on the page and any changes to the formatting of the ad (e.g., bolding specific text or adding an image). Section 6 describes a specific model factorization that improves UI generalization performance. Finally, Section 7 details a general-purpose technique for adding **bias constraints** to the model that has been applied to both improve generalization and system irreproducibility.

This paper makes the following contributions: 1) we discuss practical ML considerations from many perspectives including accuracy, efficiency and reproducibility, 2) we detail the real-world application of techniques that have improved efficiency and accuracy, in some cases describing adaptations specific to online learning, and 3) we describe how models can better generalize across UI treatments through model factorization and bias constraints.

## 2 MODEL AND TRAINING OVERVIEW

A major design choice is how to represent an ad-query pair $x$. The semantic information in the language of the query and the ad headlines is the most critical component. Usage of attention layers on top of raw text tokens may generate the most useful language embeddings in current literature [64], but we find better accuracy and efficiency trade-offs by combining variations of fully-connected DNNs with simple feature generation such as bi-grams and n-grams on sub-word units. The short nature of user queries and ad headlines is a contributing factor. Data is highly sparse for these features, with typically only a tiny fraction of non-zero feature values per example.

All features are treated as categorical and mapped to sparse embedding tables. Given an input $x$, we concatenate the embedding values for all features to form a vector $e$, the **embedding input layer** of our DNN. $E$ denotes a minibatch of embedding values $e$ across several examples.

Next, we formally describe a simplified version of the model's fully-connected neural network architecture. Later sections will introduce variations to this architecture that improve accuracy, efficiency, or reproducibility. We feed $E$ into a fully-connected hidden layer $H_1 = \sigma(EW_1)$ that performs a linear transformation of $E$ using weights $W_1$ followed by non-linear activation $\sigma$. Hidden layers $H_i = \sigma(H_{i-1}W_i)$ are stacked, with the output of the $k$th layer feeding into an output layer $\hat{y} = \text{sigmoid}(H_k W_{k+1})$ that generates the model's prediction corresponding to a click estimate $\hat{y}$. Model weights are optimized following $\min_W \sum_i \mathcal{L}(y_i, \hat{y}_i)$. We found ReLUs to be a good choice for the activation function; Section 5 describes improvements using *smoothed* activation functions. The model is trained through supervised learning with the logistic loss of the observed click label $y$ with respect to $\hat{y}$. Sections 4 and 7 describe additional losses that have improved our model. Training uses synchronous minibatch SGD on Tensor Processing Units (TPUs) [37]: at each training step $t$, compute gradients $G_t$ of the loss on a batch of examples (ranging up to millions of examples), and weights are optimized with an adaptive optimizer. We find that AdaGrad [25, 46] works well for optimizing both embedding weights and dense network weights. Moreover, In Section 4.2 discusses accuracy improvements from deploying a **second-order optimizer**: Distributed Shampoo [5] for training dense network weights, which to our knowledge, is the first known large-scale deployment in a production scale neural network training system.

### 2.1 Online Optimization

Given the non-stationarity of data in ads optimization, we find that online learning methods perform best in practice [45]. Models train using a single sequential pass over logged examples in chronological order. Each model continues to process new query-ad examples as data arrives [62]. For evaluation, we use models' predictions on each example from before the example is trained on (i.e., **progressive validation**) [12]. This setup has a number of practical advantages. Since all metrics are computed before an example is trained on, we have an immediate measure of generalization that reflects our deployment setup. Because we do not need to maintain a holdout validation set, we can effectively use all data for training, leading to higher confidence measurements. This setup allows the entire learning platform to be implemented as a single-pass streaming algorithm, facilitating the use of large datasets.

## 3 ML EFFICIENCY

Our CTR prediction system provides predictions for all ads shown to users, scoring a large set of eligible ads for billions of queries per day and requiring support for inference at rates above 100,000 QPS. Any increase in compute used for inference directly translates into substantial additional deployment costs. Latency of inference is also critical for real-time CTR prediction and related auctions. As we evaluate improvements to our model, we carefully weigh any accuracy improvements against increases in inference cost.

Model training costs are likewise important to consider. For continuous research with a fixed computational budget, the most important axes for measuring costs are bandwidth (number of models that can be trained concurrently), latency (end-to-end evaluation time for a new model), and throughput (models that can be trained per unit time).

Where inference and training costs may differ, several ML techniques are available to make trade-offs. Distillation is particularly useful for controlling inference costs or amortizing training costs (see Section 4.1.2). Techniques related to adaptive network growth [20] can control training costs relative to a larger final model (with larger inference cost).

Efficient management of computational resources for ML training is implemented via maximizing model throughput, subject to constraints on minimum bandwidth and maximum training latency. We find that required bandwidth is most frequently governed by the number of researchers addressing a fixed task. For an impactful ads model, this may represent many dozens of engineers attempting incremental progress on a single modelling task. Allowable training latency is a function of researcher preference, varying from hours to weeks in practice. Varying parallelism (i.e., number of accelerator chips) in training controls development latency. As in many systems, lowered latency often comes at the expense of throughput. For example, using twice the number of chips speeds up training, but most often does so sub-linearly (training is less than twice as fast) because of parallelization overhead.

For any given ML advancement, immediate gains must be weighed against the long-term cost to future R&D. For instance, naively scaling up the size of a large DNN might provide immediate accuracy but add prohibitive cost to future training (Table 1 includes a comparison of techniques and includes one such naive scaling baseline).

We have found that there are many techniques and model architectures from literature that offer significant improvements in model accuracy, but fail the test of whether these improvements are worth the trade-offs (e.g., ensembling many models, or full stochastic variational Bayesian inference [13]). We have also found that many accuracy-improving ML techniques can be recast as efficiency-improving via adjusting model parameters (especially total number of weights) in order to lower training costs. Thus, when we evaluate a technique, we are often interested in two tuning points: 1) what is the improvement in accuracy when training cost is neutral and 2) what is the training cost improvement if model capacity is lowered until accuracy is neutral. In our setting, some techniques are much better at improving training costs (e.g., distillation in Section 4.1.2) while others are better at improving accuracy. Figure 1 illustrates these two tuning axes.

We survey some successfully deployed efficiency techniques in the remainder of this section. Section 3.1 details the use of matrix factorization bottlenecks to approximate large matrix multiplication with reduced cost. Section 3.2 describes AutoML, an efficient RL-based architecture search that is used to identify model configurations that balance cost and accuracy. Section 3.3 discusses a set of effective sampling strategies to reduce data used for training without hurting accuracy.

$\square \to h(W_{x}+b)?$

$n^2$

## 3.1 Bottlenecks

One practical way to achieve accuracy is to scale up the widths of all the layers in the network. The wider they are, the more non-linearities there are in the model, and in practice this improves model accuracy. On the other hand, the size of the matrices involved in the loss and gradient calculations increases, making the underlying **matmul** computations slower. Unfortunately, the cost of matmul operations (naively) scale up quadratically in the size of their inputs. To compute the output of a hidden layer $H_i = \sigma(H_{i-1}W_i)$

$W = \left[\quad\right]^{k}\left[\quad\right]\left[\quad\right]$

where $W_i \in \mathbb{R}^{m \times n}$, we perform $m \times n$ multiply-add operations for each input row in $H_{i-1}$. The 'wider is better' strategy typically isn't cost-effective [23]. We find that carefully inserting **bottleneck layers** of low-rank matrices between layers of non-linearities greatly reduces scaling costs, with only a small loss of relative accuracy.

Applying singular value decomposition to $W_i$'s, we often observe that the top half of singular values contribute to over 90% of the norm of singular values. This suggests that we can approximate $H_{i-1}W_i$ by a bottleneck layer $H_{i-1}U_iV_i$, where $U_i \in \mathbb{R}^{m \times k}, V_i \in \mathbb{R}^{k \times n}$. The amount of compute reduces to $m \times k + k \times n$, which can be significant for small enough $k$. For a fixed $k$, if we scale $m, n$ by constant $c$, compute scales only linearly with $c$. Empirically, we found that accuracy loss from this approximation was indeed small. By carefully balancing the following two factors, we were able to leverage bottlenecks to achieve better accuracy without increasing computation cost: (1) increasing layer sizes toward better accuracy, at the cost of more compute, and (2) inserting bottleneck layers to reduce compute, at a small loss of accuracy. Balancing of these two can be done manually or via AutoML techniques (discussed in the next section). A recent manual application of this technique to the model (without AutoML tuning) reduced time per training step by 7% without impacting accuracy (See Table 2 for a summary of efficiency techniques).

## 3.2 AutoML for Efficiency

To develop an ads CTR prediction model architecture with optimal accuracy/cost trade-off, we typically have to tune the embedding widths of dozens of features and layer widths for each layer in the DNN. Assuming even just a small constant number of options for each such width, the combinatorial search space quickly reaches intractable scales. For industrial-scale models, it is not cost-effective to conduct traditional architecture search with multiple iterations [50, 76]. We have successfully adopted neural architecture search based on weight sharing [9] to efficiently explore network configurations (e.g., varying layer width, embedding dimension) to find versions of our model that provide neutral accuracy with decreased training and serving cost. As illustrated in Figure 2, this is achieved by three components: a weight-sharing network, an RL controller, and constraints.

The weight-sharing network builds a super-network containing all candidate architectures in the search space as sub-networks. In this way, we can train all candidate architectures simultaneously in a single iteration and select a specific architecture by activating part of the super-network with masking. This setup significantly reduces the number of exploration iterations from O(1000) to O(1).

The reinforcement learning controller maintains a sampling distribution, $\theta_{dist}$, over candidate networks. It samples a set of decisions $(d_1, d_2, \ldots)$ to activate a sub-network at each training step. We then do a forward pass for the activated sub-network to compute loss and cost. Based on that, we estimate the reward value $R(d_1, d_2, \ldots)$ and conduct a policy gradient update using the REINFORCE algorithm [68] as follows:

$$\theta_{dist} = \theta_{dist} + \alpha_0 \cdot (R(d_1, d_2, \ldots) - \overline{R}) \cdot \nabla \log P(d_1, d_2, \ldots | \theta_{dist}),$$

where $\overline{R}$ denotes the moving average value of the reward and $\alpha_0$ is the learning rate for the reinforcement learning algorithm.

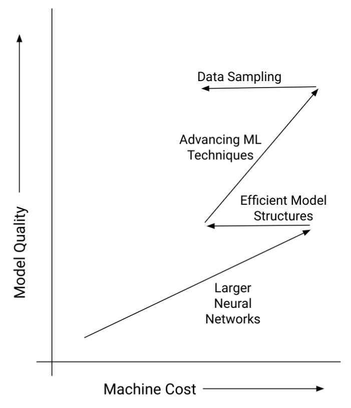


**Figure 1: "Switch-backs" of incremental costly quality-improving techniques and efficiency methods.** ***(Illustration not to any scale.)***

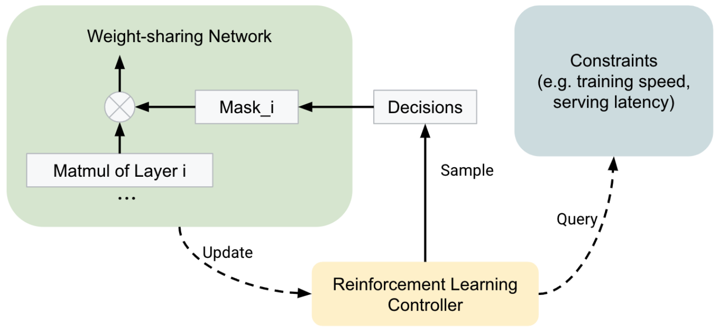


**Figure 2: Weight-sharing based NAS with cost constraints.**

Through the update at each training step, the sampling rate of better architectures will gradually increase and the sampling distribution will eventually converge to a promising architecture. We select the architecture with maximum likelihood at the end of the training. Constraints specify how to compute the cost of the activated sub-network, which can typically be done by estimating the number of floating-point operations or running a pre-built hardware-aware neural cost model. The reinforcement learning controller incorporates the provided cost estimate into the reward (e.g., $R = R_{\text{accuracy}} + \gamma \cdot |\text{cost}/\text{target} - 1|$, where $\gamma < 0$) [9] in order to force the sampling distribution to converge to a cost-constrained point. In order to search for architectures with lower training cost but neutral accuracy, in our system we set up multiple AutoML tasks with different constraint targets (e.g. 85%/90%/95% of the baseline cost) and selected the one with neutral accuracy and smallest training cost. A recent application of this architecture search to the model reduced time per training step by 16% without reducing accuracy.

## 3.3 Data Sampling

Historical examples of clicks on search ads make up a large dataset that increases substantially every day. The diminishing returns of ever larger datasets dictate that it is not beneficial to retain all the data. The marginal value for improving model quality goes toward zero, and eventually does not justify any extra machine costs for training compute and data storage. Alongside using ML optimization techniques to improve ML efficiency, we also use data sampling to control training costs. Given that training is a single-pass over data in time-order, there are two ways to reduce the training dataset: 1) restricting the time range of data consumed; and 2) sampling the data within that range. Limiting training data to more recent periods is intuitive. As we extend our date range further back in time, the data becomes less relevant to future problems. Within any range, clicked examples are more infrequent and more important to our learning task; so we sample the non-clicked examples to achieve rough class balance. Since this is primarily for efficiency, exact class balance is unnecessary. A constant sampling rate (a constant class imbalance prior) can be used with a simple single-pass filter. To keep model predictions unbiased, importance weighting is used to up-weight negative examples by the inverse of the sampling rate. Two additional sampling strategies that have proved effective are as follows:

- Sampling examples associated with a low logistic loss (typically examples with low estimated CTR and no click).
- Sampling examples that are very unlikely to have been seen by the user based on their position on the page.

The thresholds for the conditions above are hand-tuned and chosen to maximize data reduction without hurting model accuracy. These strategies are implemented by applying a small, constant sampling rate to all examples meeting any of the conditions above. Pseudo-Random sampling determines whether examples should be kept and re-weighted or simply discarded. This ensures that all training models train on the same data. This scheme may be viewed as a practical version of [28] for large problem instances with expensive evaluation. Simple random sampling allows us to keep model estimates unbiased with simple constant importance re-weighting. It is important to avoid very small sampling rates in this scheme, the consequent large up-weighting can lead to model instability. Re-weighting is particularly important for maintaining calibration, since these sampling strategies are directly correlated to labels.

For sampling strategies that involve knowing the loss on an example, calculating that loss would require running inference on the training example, removing most of the performance gains. For this reason, we use a proxy value based on a prediction made by a "teacher model". In this two-pass approach. We first train once over all data to compute losses and associated sampling rates, and then once on the sub-sampled data. The first pass uses the same teacher model for distillation (Section 4.1.2) and is only done once. Iterative research can then be performed solely on the subsampled data. While these latter models will have different losses per example, the first pass loss-estimates still provide a good signal for the 'difficulty' of the training example and leads to good results in practice. Overall our combination of class re-balancing and loss-based sampling strategies reduces the data to < 25% of the original dataset for any given period without significant loss in accuracy.

## 4 ACCURACY

Next we detail a set of techniques aimed at improving the accuracy of the system. We discuss: additional losses that better align offline training-time metrics with important business metrics, the application of distillation to our online training setting, the adaptation of the Shampoo second-order optimizer to our model, and the use of Deep and & Cross networks.

### 4.1 Loss Engineering

Loss engineering plays an important role in our system. As the goal of our model is to predict whether an ad will be clicked, our model generally optimizes for logistic loss, often thought of as the cross-entropy between model predictions and the binary task (click/no-click) labels for each example. Using logistic loss allows model predictions to be unbiased so that the prediction can be interpreted directly as a calibrated probability. Binary predictions can be improved by introducing soft prediction through distillation methods [35]. Beyond estimating the CTR per ad, it is important that the set of candidate ads for a particular query is correctly ranked (such that ads with clicks have higher CTR than ads without clicks), thus incorporating proper ranking losses is also important. In this section, we discuss novel auxiliary losses and introduce multi-task and multi-objective methods for joint training with these losses

*4.1.1 Rank Losses.* We found that Area under the ROC curve computed per query (PerQueryAUC) is a metric well correlated with business metrics quantifying the overall performance of a model. In addition to using PerQueryAUC during evaluation, we also use a relaxation of this metric, i.e., rank-loss, as a second training loss in our model. There are many rank losses in the learning-to-rank family [17, 48]. We find one effective approximation is Ranknet loss [16], which is a pairwise logistic loss:

$$-\sum_{i\in\{y_i=1\}}\sum_{j\in\{y_j\neq 1\}}\log(\text{sigmoid}(s_i, s_j)),$$

where $s_i, s_j$ are logit scores of two examples.

Rank losses should be trained jointly with logistic loss; there are several potential optimization setups. In one setup, we create a multi-objective optimization problem [52]:

$$\mathcal{L}(W) = \alpha_1\mathcal{L}_{\text{rank}}(\mathbf{y}_{\text{rank}}, \mathbf{s}) + (1-\alpha_1)\mathcal{L}_{\text{logistic}}(\mathbf{y}, \mathbf{s}),$$

where $\mathbf{s}$ are logit scores for examples, $\mathbf{y}_{\text{rank}}$ are ranking labels, $\mathbf{y}$ are the binary task labels, and $\alpha_1 \in (0, 1)$ is the rank-loss weight. Another solution is to use multi-task learning [18, 51], where the model produces multiple different estimates $s$ for each loss.

$$\mathcal{L}(W_{\text{shared}}, W_{\text{logistic}}, W_{\text{rank}}) =$$
$$\alpha_1\mathcal{L}_{\text{rank}}(\mathbf{y}, \mathbf{s}_{\text{rank}}) + (1-\alpha_1)\mathcal{L}_{\text{logistic}}(\mathbf{y}, \mathbf{s}_{\text{logistic}}),$$

where $W_{\text{shared}}$ are weights shared between the two losses, $W_{\text{logistic}}$ are for the logistic loss output, and $W_{\text{rank}}$ are for the rank-loss output. In this case, the ranking loss affects the "main" prediction $\mathbf{s}_{\text{logistic}}$ as a "regularizer" on $W_{\text{shared}}$.

As rank losses are not naturally calibrated predictors of click probabilities, the model's predictions will be biased. A strong bias correction component is needed to ensure the model's prediction is unbiased per example. More detail can be found in Section 7. Application of ranklosses to the model generated accuracy improvements of $-0.81\%$ with a slight increase in training cost of 1%.

*4.1.2 Distillation.* Distillation adds an additional auxiliary loss requiring matching the predictions of a high-capacity teacher model, treating teacher predictions as soft labels [35]. In our model, we use a **two-pass online distillation** setup. On the first pass, a teacher model records its predictions progressively before training on examples. Student models consume the teacher's predictions while training on the second pass. Thus, the cost of generating the predictions from the single teacher can be amortized across many students (without requiring the teacher to repeat inference to generate predictions). In addition to improving accuracy, distillation can also be used for reducing training data costs. Since the high-capacity teacher is trained once, it can be trained on a larger data set. Students benefit implicitly from the teachers prior knowledge of the larger training set, and so require training only smaller and more recent data. The addition of distillation to the model improved accuracy by 0.41% without increasing training costs (in the student).

*4.1.3 Curriculums of Losses.* In machine learning, curriculum learning [10] typically involves a model learning easy tasks first and gradually switching to harder tasks. We found that training on all classes of losses in the beginning of training increased model instability (manifesting as outlier gradients which cause quality to diverge). Thus, we apply an approach similar to curriculum learning to ramp up losses, starting with the binary logistic loss and gradually ramping up distillation and rank losses over the course of training.

### 4.2 Second-order Optimization

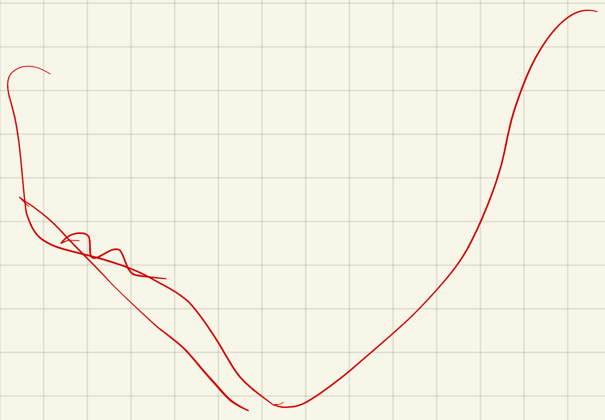
Second-order optimization methods that use second derivatives or second-order statistics are known to have better convergence properties compared to first-order methods [47]. Yet to our knowledge, second-order methods are rarely reported to be used in production ML systems for DNNs. Recent work on Distributed Shampoo [5, 32] has made second-order optimization feasible for our model by leveraging the heterogeneous compute offered by TPUs and host-CPUs, and by employing additional algorithmic efficiency improvements.

For our model, Distributed Shampoo provided much faster convergence with respect to training steps, and yielded better accuracy when compared to standard adaptive optimization techniques including AdaGrad [25], Adam [38], Yogi [72], and LAMB [70]. While second-order methods are known to provide faster convergence compared to first-order methods in the literature - It often fails to provide competitive wall-clock time due to the computational overheads in the optimizer, especially on smaller scale benchmarks. For our model, second-order optimization method was an ideal

$$\theta_i = \theta_{i-1} - \eta \nabla L$$

$$\theta_i = \theta_{i-1} - (\underline{\nabla^2 L})^{-1} \nabla L$$

L G R



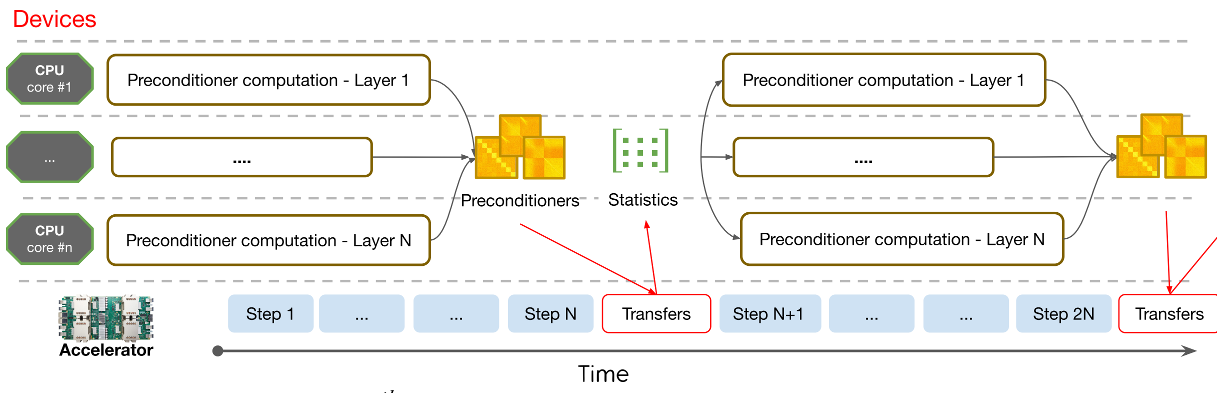

Devices
CPU core #1
Preconditioner computation - Layer 1
...
....
CPU core #n
Preconditioner computation - Layer N
Preconditioners
Statistics
Preconditioner computation - Layer 1
....
Preconditioner computation - Layer N
Accelerator
Step 1
...
...
Step N
Transfers
Step N+1
...
...
Step 2N
Transfers
Time

**Figure 3: Distributed Shampoo [5]: inverse-$p^{th}$ root computations in double precision runs every $N$ steps and asynchronously pipelined on all CPU cores attached to the TPU accelerators.**

candidate due to the large batch sizes used in training which amortizes the cost of costly update rule. Training time only increased by approximately 10%, and the improvements to model accuracy far outweighed the increase in training time. We next discuss salient implementation details specific to our model.

*Learning Rate Grafting.* One of the main challenges in online optimization is defining a learning rate schedule. In contrast to training on static datasets, the number of steps an online model will require is unknown and may be unbounded. Accordingly, popular learning rate schedules from literature depending on fixed time horizons, such as cosine decay or exponential decay, perform worse in contrast to the implicit data-dependent adaptive schedule from AdaGrad [25]. As observed in literature [2], we also find that AdaGrad's implicit schedule works quite well in the online setting; especially after the $\epsilon$ parameter (the initial accumulator value) is tuned. Accordingly, we bootstrap the schedule for Distributed Shampoo via grafting the per-layer step size from AdaGrad. More precisely, we use the direction from Shampoo while using the magnitude of step size from AdaGrad at a per-layer granularity. An essential feature of this bootstrapping is that it allowed us to inherit hyper-parameters from previous AdaGrad tunings to search for a Pareto optimal configuration.

*Momentum.* Another effective implementation choice is the combination of Nesterov-styled momentum with the preconditioned gradient. Our analysis suggests that momentum added modest gains on top of Shampoo without increasing the computational overhead while marginally increasing the memory overhead. Computational overhead was addressed via the approximations described in [61].

*Stability & Efficiency.* Distributed Shampoo has higher computational complexity per step as it involves matrix multiplication of large matrices for preconditioning and statistics/preconditioner computation. We addressed these overheads with several techniques in our deployment. For example, the block-diagonalization suggested in [5] effectively reduced computational complexity while also allowing the implementation of parallel updates for each block in the data-parallel setting via weight-update sharding [69]. This optimization reduced the overall step time. Moreover, optimizer overheads are independent of batch size; thus, our use of large batch sizes helped reduce overall computational overhead. Finally, we found that the condition number of statistics used for preconditioning can vary in range, reaching more than $10^{10}$. As numerical stability and robustness are of utmost importance in production, we use double precision numerics. To compute the preconditioners, we use the CPUs attached to the TPUs to run inverse-$p$th roots and exploit a faster algorithm; the coupled Newton iteration [31] for larger preconditioners as in Figure 3.

When integrated with the ad click prediction model, the optimizer improved our primary measure of accuracy, Area under the ROC curve computed per query (PerQueryAuc), by 0.44%. Accuracy improvements above 0.1% are considered significant. For comparison: a naive scaling of the deep network by 2x yields a PerQueryAUC improvement of 0.13%. See Table 1 for a summary of accuracy technique results.

## 4.3 Deep & Cross Network

Learning effective feature crosses is critical for recommender systems [65, 74]. We adopt an efficient variant of DCNv2 [65] using bottlenecks. This is added between the embedding layer $e$ described in Section 2 and the DNN. We next describe the Deep & Cross Network architecture and its embedding layer input. We use a standard embedding projection layer for sparse categorical features. We project categorical feature $i$ from a higher dimensional sparse space to a lower dimensional dense space using $\tilde{e}_i = W_i x_i$, where $x_i \in \{0,1\}^{v_i}$; $W_i \in \mathbb{R}^{m_i \times v_i}$ is the learned projection matrix; $\tilde{e}_i$ is the dense embedding representation; and $v_i$ and $m_i$ represent the vocabulary and dense embedding sizes respectively. For multivalent features, we use average pooling of embedding vectors. Embedding dimensions $\{m_i\}$ are tuned for efficiency and accuracy trade-offs using AutoML (Section 3.2). Output of the embedding layer is a wide concatenated vector $e_0 = \text{concat}(\tilde{e}_1, \tilde{e}_2 \ldots \tilde{e}_F) \in \mathbb{R}^m$ for $F$ features. For crosses, we adopt an efficient variant of [65], applied directly on top of the embedding layer to explicitly learn feature crosses: $e_i = \alpha_2\big(e_0 \odot U_i V_i e_{i-1}\big) + e_{i-1}$, where $e_i, e_{i-1} \in \mathbb{R}^m$ represent the output and input of the $i^{\text{th}}$ cross layer, respectively; $U_i \in \mathbb{R}^{m \times k}$ and $V_i \in \mathbb{R}^{k \times m}$ are the learned weight matrices leveraging bottlenecks (Section 3.1) for efficiency; $\alpha_2$ is a scalar, ramping up from $0 \rightarrow 1$ during initial training, allowing the model to first learn the

| Technique | Accuracy Improvement | Training Cost Increase | Inference Cost Increase |
|---|---|---|---|
| Deep & Cross Network | 0.18% | 3% | 1% |
| Distributed Shampoo Optimizer | 0.44% | 10% | 0% |
| Distillation | 0.46% | <1%* | 0% |
| Rank Losses | 0.81% | <1% | 0% |
| Baseline: 2x DNN Size | 0.13% | 36% | 10% |

**Table 1: Accuracy improvement and training/inference costs for accuracy improving techniques. * Distillation does not include teacher cost which, due to amortization, is a small fraction of overall training costs.**

| Technique | Training Cost Decrease |
|---|---|
| Bottlenecks | 7% |
| AutoML | 16% |
| Data Sampling | 75% |

**Table 2: Training cost improvements of applied techniques.**

embeddings and then the crosses in a curriculum fashion. Furthermore, this ReZero initialization [7] also improves model stability and reproducibility (Section 5).

In practice adding the Deep & Cross Network to the model yielded an accuracy improvement of 0.18% with a minimal increase in training cost of 3%.

### 4.4 Summary of Efficiency and Accuracy Results

Below we share measurements of the relative impact of the previously discussed efficiency and accuracy techniques as applied to the production model. The goal is to give a very rough sense of the impact of these techniques and their accuracy vs. efficiency tradeoffs. While precise measures of accuracy improvement on one particular model are not necessarily meaningful, we believe the coarse ranking of techniques and rough magnitude of results are interesting (and are consistent with our general experience).

The baseline 2x DNN size model doubles the number of hidden layers. Note, that sparse embedding lookups add to the overall training cost, thus doubling the number layers does not proportionally increase the cost.

## 5 IRREPRODUCIBILITY

Irreproducibility, noted in Section 1, may not be easy to detect because it may appear in post deployment **system metrics** and not in progressive validation quality metrics. A pair of duplicate models may converge to two different optima of the highly non-convex objective, giving equal average accuracy, but different individual predictions, but with different downstream system/auction outcomes. Model deployment leads to further divergence, as ads selected by deployed models become part of subsequent training examples [62]. This can critically affect R&D: experimental models may appear beneficial, but gains may disappear when they are retrained and deployed in production. Theoretical analysis is complex even in the simple convex case, which is considered only in very recent work [3]. Many factors contribute to irreproducibility [29, 30, 54, 59, 60, 75], including random initialization, non-determinism in training due to highly-parallelized and highly-distributed training pipelines, numerical errors, hardware, and more. Slight deviations early in training may lead to very different models [1]. While standard training metrics do not expose system irreproducibility, we can use deviations of predictions on individual examples as a cheap proxy, allowing us to fail fast prior to evaluation at deployment-time. Common statistical metrics (standard deviation, various divergences) can be used [21, 71] but they require training many more models, which is undesirable at our scale. Instead, we use the **Relative Prediction Difference** (PD) [56, 58] metric

$$\Delta_r \triangleq 1/M \cdot \sum_i |\hat{y}_{i,1} - \hat{y}_{i,2}| / [(\hat{y}_{i,1} + \hat{y}_{i,2})/2]$$

, measuring absolute point-wise difference in model predictions for a pair of models (subscripts 1 and 2), normalized by the pair's average prediction. Computing PD requires training a pair of models instead of one, but we have observed that reducing PD is sufficient to improve reproducibility of important system metrics. In this section, we focus on methods to improve PD; Section 7 focuses on directly improving system metrics.

PDs may be as high as 20% for deep models. Perhaps surprisingly, standard methods such as fixed initialization, regularization, dropout, data augmentation, as well as new methods imposing constraints [11, 55] either failed to improve PD or improved PD at the cost of accuracy degradation. Techniques like warm-starting model weights to values of previously trained models may not be preferable because they can anchor the model to a potentially bad solution space and do not help the development cycle for newer more reproducible models for which there is no anchor.

Other techniques have shown varying levels of success. Ensembles [24], specifically *self*-ensembles [4], where we average predictions of multiple model duplicates (each initialized differently), can reduce prediction variance and PD. However, maintaining ensembles in a production system with multiple components builds up substantial technical debt [53]. While some literature [39, 43, 67] describes accuracy advantages for ensembles, in our regime, ensembles degraded accuracy relative to equal-cost single networks. We believe this is because, unlike in the benchmark image models, examples in online CTR systems are visited once, and, more importantly, the learned model parameters are dominated by sparse embeddings. Relatedly, more sophisticated techniques based on ensembling and constraints can also improve PD [6, 56, 57].

Techniques described above trade accuracy and complexity for better reproducibility, requiring either ensembles or constraints. Further study and experimentation revealed that the popular use of Rectified Linear Unit (ReLU) activations contributes to increased PD. ReLU's gradient discontinuity at 0 induces a highly non-convex

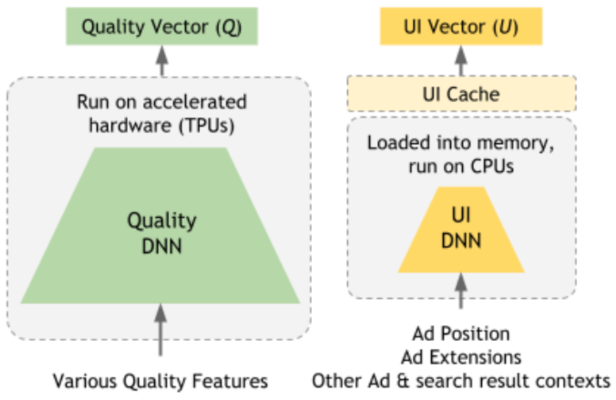


**Figure 4: Model factorization into separable Quality and UI models with estimated CTR** $:= \tau(Q \cdot U)$

loss landscape. Smoother activations, on the other hand, reduce the amount of non-convexity, and can lead to more reproducible models [58]. Empirical evaluations of various smooth activations [8, 34, 49, 73] have shown not only better reproducibility compared to ReLU, but also slightly better accuracy. The best reproducibility-accuracy trade-offs in our system were attained by the simple *Smooth reLU (SmeLU)* activation proposed in [58]. The function form is:

$$f_{\text{SmeLU}}(z) = \begin{cases} 0; & z \le -\beta \\ \frac{(z+\beta)^2}{4\beta}; & |z| \le \beta \\ z; & z \ge \beta. \end{cases} \qquad (1)$$

In our system, 3-component ensembles reduced PD from 17% to 12% and anti-distillation reduced PD further to 10% with no accuracy loss. SmeLU allowed launching a non-ensemble model with PD less than 10% that also improved accuracy by 0.1%. System reproducibility metrics also improved to acceptable levels compared to the unacceptable levels of ReLU single component models.

## 6 GENERALIZING ACROSS UI TREATMENTS

One of the major factors in CTR performance of an ad is its **UI treatment**, including positioning, placement relative to other results on the page, and specific renderings such as bolded text or inlined images. A complex auction must explore not just the set of results to show, but how they should be positioned relative to other content, and how they should be individually rendered [19]. This exploration must take place efficiently over a combinatorially large space of possible treatments.

We solve this through model factorization, replacing estimated CTR with $\tau(Q \cdot U)$, composed of a transfer function $\tau$ where $Q$, $U$ are separable models that output *vectorized* representations of the *Quality* and the *UI*, respectively, and are combined using an inner-product. While $Q$, consisting of a large DNN and various feature embeddings, is a costly model, it needs to be evaluated *only once* per ad, irrespective of the number of UI treatments. In contrast, $U$, being a much lighter model, can be evaluated hundreds of times per ad. Moreover, due to the relatively small feature space of the UI model, outputs can be cached to absorb a significant portion of lookup costs (as seen in Figure 4).

Separately from model performance requirements, accounting for the influence of UI treatments on CTR is also a crucial factor for model quality. Auction dynamics deliberately create strong correlations between individual ads and specific UI treatments. Results that are lower on the page may have low CTR regardless of their relevance to the query. Failure to properly disentangle these correlations creates inaccuracy when generalizing over UI treatments (e.g., estimating CTR if the same ad was shown higher on the page). Pricing and eligibility decisions depend crucially on CTR estimates of sub-optimal UIs that are rarely occurring in the wild. For instance, our system shouldn't show irrelevant ads, and so such scenarios will not be in the training corpus, and so estimates of their irrelevance (low CTR) will be out of distribution. But these estimates are needed to ensure the ads do not show. Even for relevant ads, there is a similar problem. Performance of ads that rarely show in first position may still be used to set the price of those ads that often do show in first position. This creates a specific generalization problem related to UI, addressed in Section 7.

Calibration is an important characteristic for large-scale ads recommendation. We define calibration bias as label minus prediction, and want this to be near zero *per ad*. A calibrated model allows us to use estimated CTR to determine the trade-off between showing and not showing an ad, and between showing one ad versus another; both calculations can be used in downstream tasks such as UI treatment selection, auction pricing, or understanding of ad viewability.

The related concept of **credit attribution** is similar to counterfactual reasoning [15] or bias in implicit feedback [36]. It is a specific non-identifiability in model weights that can contribute to irreproducibility (Section 5). Consider an example to illustrate the UI effect (Section 6): assume that model A has seen many training examples with high-CTR ads in high positions, and (incorrectly) learned that ad position most influences CTR. Model B, defined similar to A, trains first on the few examples where high-CTR ads appear in low positions, and (correctly) learns that something else (e.g., ad relevancy to query) is causing high CTR. Both models produce the same estimated CTR for these ads but for different reasons, and when they are deployed, model A will likely show fewer ads because it will not consider otherwise useful ads in lower positions; these models will show system irreproducibility.

In our system, we use a novel, general-purpose technique called **bias constraints** to address both calibration and credit attribution. We add calibration bias constraints to our objective function, enforced on relevant slices of either the training set or a separate, labelled dataset. This allows us reduce non-identifiability by anchoring model loss to a desired part of the solution space (e.g., one that satisfies calibration) (Figure 5a). By extension, we reduce irreproducibility by anchoring a retrained model to the same solution.

Our technique is more lightweight than other methods used for large-scale, online training (counterfactual reasoning [15], variations of inverse propensity scoring [36, 41]): in practice, there are fewer parameters to tune, and we simply add an additional term to our objective rather than changing the model structure. To address calibration, [14] adjusts model predictions in a separate calibration step using isotonic regression, a non-parametric method. Our technique does calibration jointly with estimation, and is more similar to methods which consider efficient optimization of complex and

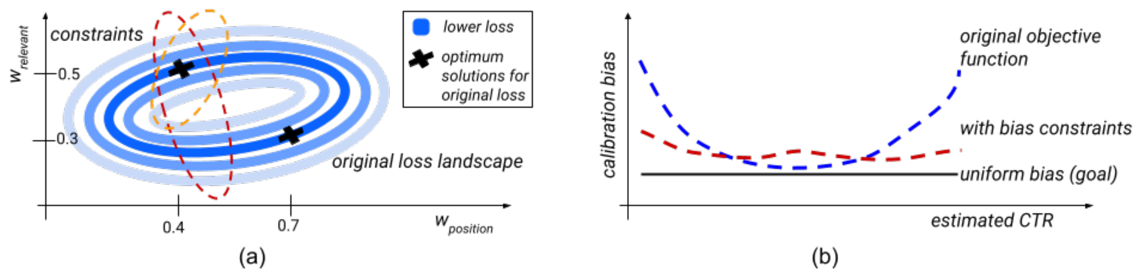


**Figure 5: (a) Loss landscape for a model with non-identifiability across two weights and how bias constraints help find the right solution: we add additional criteria (red and orange curves) such that we choose the correct solution at optimum loss (dark blue curve). (b) Calibration bias across buckets of estimated CTR. For calibrated predictions, we expect uniform bias (black curve). Whereas a model with the original objective function is biased for certain buckets of estimated CTR (blue curve), we can get much closer to uniform with bias constraints (red curve).**

augmented objectives (e.g., [27, 44]). Using additional constraints on the objective allows us to address a wide range of calibration and credit attribution issues.

## 7 BIAS CONSTRAINTS

### 7.1 Online Optimization of Bias Constraints

We now optimize our original objective function with the constraint that $\forall k \forall i \in S_k, (y_i - \hat{y}_i) = 0$. Here, $S_k$ are subsets of the training set which we'd like to be calibrated (e.g., under-represented classes of data) or new training data that we may or may not optimize the original model weights over (e.g., out-of-distribution or off-policy data gathered from either randomized interventions or exploration scavenging [36, 40, 66]). To aid optimization, we first transform this into an unconstrained optimization problem by introducing a dual variable $\lambda_{k,i}$ for each constraint and maximizing the Lagrangian relative to the dual variables. Next, instead of enforcing zero bias per example, we ask that the squared average bias across $S_k$ is zero. This reduces the number of dual variables to $\{\lambda_k\}$, and is equivalent to adding an L2 regularization on $\lambda_k$ with a constraint of zero average bias. For a constant $\alpha_3$ controlling regularization, and tuned via typical hyperparameter tuning techniques (e.g. grid search), our new optimization is:

$$\min_{W} \max_{\lambda_k} \sum_{i} \mathcal{L}(y_i, \hat{y}_i) + \sum_{k=1}^{K} \sum_{i \in S_k} (\lambda_k (y_i - \hat{y}_i) - \frac{\alpha_3}{2} \lambda_k^2)$$

Any degraded accuracy or stability is mitigated by combinations of the following tunings, ordered by impact: ramping up the bias constraint term, reducing the learning rate on $\{\lambda_k\}$, increasing $\alpha_3$, or adding more or finer-grained constraints (breaking up $S_k$). We believe the first two can help normalize any differences between the magnitude of the dual variables and other weights, and the latter two help lessen the strength of the bias term if $S_k$ aren't optimally selected.

### 7.2 Bias Constraints for General Calibration

If we plot calibration bias across buckets of interesting variables, such as estimated CTR or other system metrics, we expect a calibrated model to have uniform bias. However, for several axes of interest, our system shows higher bias at the ends of the range (Figure 5b). We apply bias constraints to this problem by defining $S_k$ to be examples in each bucket of, e.g., estimated CTR. Since we don't use the dual variables during inference, we can include estimated CTR in our training objective. With bias constraints, bias across buckets of interest becomes much more uniform: variance is reduced by more than half. This can in turn improve accuracy of downstream consumers of estimated CTR.

### 7.3 Exploratory Data and Bias Constraints

We can also use bias constraints to solve credit attribution for UI treatments. We pick $S_k$ by focusing on classes of examples that represent uncommon UI presentations for competitive queries where the ads shown may be quite different. For example, $S_1$ might be examples where a high-CTR ad showed at the bottom of the page, $S_2$ examples where a high-CTR ad showed in the second-to-last position on the page, etc. Depending on how model training is implemented, it may be easier to define $S_k$ in terms of existing model features (e.g., for a binary feature $f$, we split one sum over $S_k$ into two sums). We choose $\{f\}$ to include features that generate partitions large enough to not impact convergence but small enough that we expect the bias per individual example will be driven to zero (e.g., if we think that query language impacts ad placement, we will include it in $\{f\}$). For the model in Table 3, we saw substantial bias improvements on several data subsets $S_k$ related to out-of-distribution ad placement and more reproducibility with minimal accuracy impact when adding bias constraints.

Viewing the bias constraints as anchoring loss rather than changing the loss landscape (Figure 5a), we find that the technique does not fix model irreproducibility but rather mitigates system irreproducibility: we were able to cut the number of components in the ensemble by half and achieve the same level of reproducibility.

| $S_1$ Bias | $S_2$ Bias | $S_3$ Bias | Loss | Ads/Query Churn |
|---|---|---|---|---|
| -15% | -75% | -43% | +0.03% | -85% |

**Table 3: Progressive validation and deployed system metrics reported as a percent change for a bias constraint over the original model (negative is better). Ads/Query Churn records how much the percent difference in the number of ads shown above search results per query between two model retrains changes when deployed in similar conditions; we want this to be close to zero.**

## 8 CONCLUSION

We detailed a set of techniques for large-scale CTR prediction that have proven to be truly effective "in production": balancing improvements to accuracy, training and deployment cost, system reproducibility and model complexity—along with describing approaches for generalizing across UI treatments. We hope that this brief visit to the factory floor will be of interest to ML practitioners of CTR prediction systems, recommender systems, online training systems, and more generally to those interested in large industrial settings.

## REFERENCES

[1] Alessandro Achille, Matteo Rovere, and Stefano Soatto. 2017. Critical learning periods in deep neural networks. *arXiv preprint arXiv:1711.08856* (2017).

[2] Naman Agarwal, Rohan Anil, Elad Hazan, Tomer Koren, and Cyril Zhang. 2020. Disentangling adaptive gradient methods from learning rates. *arXiv preprint arXiv:2002.11803* (2020).

[3] Kwangjun Ahn, Prateek Jain, Ziwei Ji, Satyen Kale, Praneeth Netrapalli, and Gil I. Shamir. 2022. Reproducibility in optimization: Theoretical framework and Limits. *arXiv preprint arXiv:2202.04598* (2022).

[4] Zeyuan Allen-Zhu and Yuanzhi Li. 2020. Towards understanding ensemble, knowledge distillation and self-distillation in deep learning. *arXiv preprint arXiv:2012.09816* (2020).

[5] Rohan Anil, Vineet Gupta, Tomer Koren, Kevin Regan, and Yoram Singer. 2020. Second Order Optimization Made Practical. *https://arxiv.org/abs/2002.09018* (2020).

[6] Rohan Anil, Gabriel Pereyra, Alexandre Passos, Robert Ormandi, George E Dahl, and Geoffrey E Hinton. 2018. Large scale distributed neural network training through online distillation. *arXiv preprint arXiv:1804.03235* (2018).

[7] Thomas Bachlechner, Bodhisattwa Prasad Majumder, Henry Mao, Gary Cottrell, and Julian McAuley. 2021. Rezero is all you need: Fast convergence at large depth. In *UAI*.

[8] Jonathan T Barron. 2017. Continuously differentiable exponential linear units. *arXiv preprint arXiv:1704.07483* (2017).

[9] Gabriel Bender, Hanxiao Liu, Bo Chen, Grace Chu, Shuyang Cheng, Pieter-Jan Kindermans, and Quoc V Le. 2020. Can weight sharing outperform random architecture search? an investigation with tunas. In *CVPR*.

[10] Yoshua Bengio, Jérôme Louradour, Ronan Collobert, and Jason Weston. 2009. Curriculum Learning. In *ICML*.

[11] Srinadh Bhojanapalli, Kimberly Jenney Wilber, Andreas Veit, Ankit Singh Rawat, Seungyeon Kim, Aditya Krishna Menon, and Sanjiv Kumar. 2021. On the Reproducibility of Neural Network Predictions.

[12] Avrim Blum, Adam Kalai, and John Langford. 1999. Beating the hold-out: Bounds for k-fold and progressive cross-validation. In *COLT*.

[13] Charles Blundell, Julien Cornebise, Koray Kavukcuoglu, and Daan Wierstra. 2015. Weight uncertainty in neural network. In *International conference on machine learning*. PMLR, 1613–1622.

[14] Alexey Borisov, Julia Kiseleva, Ilya Markov, and M. de Rijke. 2018. Calibration: A Simple Way to Improve Click Models. *CIKM* (2018).

[15] Léon Bottou, Jonas Peters, Joaquin Quiñonero-Candela, Denis X. Charles, D. Max Chickering, Elon Portugaly, Dipankar Ray, Patrice Simard, and Ed Snelson. 2013. Counterfactual Reasoning and Learning Systems: The Example of Computational Advertising. *JMLR* (2013).

[16] Chris Burges, Tal Shaked, Erin Renshaw, Ari Lazier, Matt Deeds, Nicole Hamilton, and Greg Hullender. 2005. Learning to Rank Using Gradient Descent. In *ICML*.

[17] Christopher JC Burges. 2010. From Ranknet to Lambdarank to LambdaMart: An overview. *Learning* (2010).

[18] Rich Caruana. 1997. Multitask Learning. *Machine Learning* (1997).

[19] Ruggiero Cavallo, Prabhakar Krishnamurthy, Maxim Sviridenko, and Christopher A. Wilkens. 2017. Sponsored Search Auctions with Rich Ads. *CoRR* abs/1701.05948 (2017). arXiv:1701.05948 http://arxiv.org/abs/1701.05948

[20] Tianqi Chen, Ian Goodfellow, and Jonathon Shlens. 2015. Net2net: Accelerating learning via knowledge transfer. *arXiv preprint arXiv:1511.05641* (2015).

[21] Zhe Chen, Yuyan Wang, Dong Lin, Derek Cheng, Lichan Hong, Ed Chi, and Claire Cui. 2020. Beyond point estimate: Inferring ensemble prediction variation from neuron activation strength in recommender systems. *arXiv preprint arXiv:2008.07032* (2020).

[22] A. D'Amour, K. Heller, D. Moldovan, B. Adlam, B. Alipanahi, A. Beutel, C. Chen, J. Deaton, J. Eisenstein, M. D. Hoffman, F. Hormozdiari, N. Houlsby, S. Hou, G. Jerfel, A. Karthikesalingam, M. Lucic, Y. Ma, C. McLean, D. Mincu, A. Mitani, A. Montanari, Z. Nado, V. Natarajan, C. Nielson, T. F. Osborne, R. Raman, K. Ramasamy, R. Sayres, J. Schrouff, M. Seneviratne, S. Sequeira, H. Suresh, V. Veitch, M. Vladymyrov, X. Wang, K. Webster, S. Yadlowsky, T. Yun, X. Zhai, and D. Sculley. 2020. Underspecification Presents Challenges for Credibility in Modern Machine Learning. *arXiv preprint arXiv:2011.03395* (2020).

[23] Misha Denil, Babak Shakibi, Laurent Dinh, Marc'Aurelio Ranzato, and Nando de Freitas. [n. d.]. Predicting Parameters in Deep Learning. *CoRR* ([n. d.]).

[24] T. G. Dietterich. 2000. Ensemble methods in machine learning. *Lecture Notes in Computer Science* (2000).

[25] John Duchi, Elad Hazan, and Yoram Singer. 2011. Adaptive subgradient methods for online learning and stochastic optimization. *JMLR* (2011).

[26] Michael W Dusenberry, Dustin Tran, Edward Choi, Jonas Kemp, Jeremy Nixon, Ghassen Jerfel, Katherine Heller, and Andrew M Dai. 2020. Analyzing the role of model uncertainty for electronic health records. In *CHIL*.

[27] Elad Eban, Mariano Schain, Alan Mackey, Ariel Gordon, Rif A Saurous, and Gal Elidan. 2017. Scalable Learning of Non-Decomposable Objectives. In *AIStats*.

[28] William Fithian and Trevor Hastie. 2014. Local case-control sampling: Efficient subsampling in imbalanced data sets. *The Annals of Statistics* (2014).

[29] Stanislav Fort, Huiyi Hu, and Balaji Lakshminarayanan. 2020. Deep Ensembles: A Loss Landscape Perspective. arXiv:1912.02757

[30] Jonathan Frankle, Gintare Karolina Dziugaite, Daniel Roy, and Michael Carbin. 2020. Linear mode connectivity and the lottery ticket hypothesis. In *International Conference on Machine Learning*.

[31] Chun-Hua Guo and Nicholas J Higham. 2006. A Schur–Newton Method for the Matrix\boldmath p th Root and its Inverse. *SIAM J. Matrix Anal. Appl.* (2006).

[32] Vineet Gupta, Tomer Koren, and Yoram Singer. 2018. Shampoo: Preconditioned stochastic tensor optimization. In *ICML*.

[33] Xinran He, Junfeng Pan, Ou Jin, Tianbing Xu, Bo Liu, Tao Xu, Yanxin Shi, Antoine Atallah, Ralf Herbrich, Stuart Bowers, and Joaquin Quiñonero Candela. 2014. Practical Lessons from Predicting Clicks on Ads at Facebook.

[34] Dan Hendrycks and Kevin Gimpel. 2016. Gaussian error linear units (gelus). *arXiv preprint arXiv:1606.08415* (2016).

[35] Geoffrey Hinton, Oriol Vinyals, and Jeffrey Dean. 2015. Distilling the Knowledge in a Neural Network. In *NIPS Deep Learning and Representation Learning Workshop*.

[36] Thorsten Joachims, Adith Swaminathan, and Tobias Schnabel. 2017. Unbiased Learning-to-Rank with Biased Feedback. In *WSDM*.

[37] Norman P Jouppi, Doe Hyun Yoon, George Kurian, Sheng Li, Nishant Patil, James Laudon, Cliff Young, and David Patterson. 2020. A domain-specific supercomputer for training deep neural networks. *Commun. ACM* (2020).

[38] Diederik P Kingma and Jimmy Ba. 2014. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980* (2014).

[39] Dan Kondratyuk, Mingxing Tan, Matthew Brown, and Boqing Gong. 2020. When ensembling smaller models is more efficient than single large models. *arXiv preprint arXiv:2005.00570* (2020).

[40] John Langford, Alexander Strehl, and Jennifer Wortman. 2008. Exploration scavenging. In *ICML*.

[41] Damien Lefortier, Adith Swaminathan, Xiaotao Gu, Thorsten Joachims, and M. de Rijke. 2016. Large-scale Validation of Counterfactual Learning Methods: A Test-Bed. *arXiv preprint arXiv:1612.00367* (2016).

[42] Xiaoliang Ling, Weiwei Deng, Chen Gu, Hucheng Zhou, Cui Li, and Feng Sun. 2017. Model ensemble for click prediction in bing search ads. In *WWW*.

[43] Ekaterina Lobacheva, Nadezhda Chirkova, Maxim Kodryan, and Dmitry Vetrov. 2020. On power laws in deep ensembles. *arXiv preprint arXiv:2007.08483* (2020).

[44] Gideon S Mann and Andrew McCallum. 2007. Simple, Robust, Scalable Semi-supervised Learning via Expectation Regularization. In *ICML*.

[45] H Brendan McMahan, Gary Holt, D. Sculley, Michael Young, Dietmar Ebner, Julian Grady, Lan Nie, Todd Phillips, Eugene Davydov, Daniel Golovin, et al. 2013. Ad click prediction: a view from the trenches. In *SIGKDD*.

[46] H Brendan McMahan and Matthew Streeter. 2010. Adaptive bound optimization for online convex optimization. *arXiv preprint arXiv:1002.4908* (2010).

[47] Jorge Nocedal and Stephen J. Wright. 2006. *Numerical Optimization*. Springer.

[48] R. K. Pasumarthi, S. Bruch, X. Wang, C. Li, M. Bendersky, M. Najork, J. Pfeifer, N. Golbandi, R. Anil, and S. Wolf. 2019. TF-Ranking: Scalable TensorFlow Library for Learning-to-Rank. In *SIGKDD*.

[49] Prajit Ramachandran, Barret Zoph, and Quoc V Le. 2017. Searching for activation functions. *arXiv preprint arXiv:1710.05941* (2017).

[50] Esteban Real, Alok Aggarwal, Yanping Huang, and Quoc V Le. 2019. Regularized evolution for image classifier architecture search. In *AAAI*.

[51] Sebastian Ruder. 2017. An Overview of Multi-Task Learning in Deep Neural Networks. arXiv:1706.05098

[52] D. Sculley. 2010. Combined regression and ranking. In *In KDD'10*.

[53] David Sculley, Gary Holt, Daniel Golovin, Eugene Davydov, Todd Phillips, Dietmar Ebner, Vinay Chaudhary, and Michael Young. 2014. Machine learning: The high interest credit card of technical debt. (2014).

[54] Christopher J Shallue, Jaehoon Lee, Joseph Antognini, Jascha Sohl-Dickstein, Roy Frostig, and George E Dahl. 2018. Measuring the effects of data parallelism on neural network training. *arXiv preprint arXiv:1811.03600* (2018).

[55] Gil I. Shamir. 2018. Systems and Methods for Improved Generalization, Reproducibility, and Stabilization of Neural Networks via Error Control Code Constraints.

[56] Gil I Shamir and Lorenzo Coviello. 2020. Anti-Distillation: Improving reproducibility of deep networks. *arXiv preprint arXiv:2010.09923* (2020).
[57] Gil I. Shamir and Lorenzo Coviello. 2020. Distilling from Ensembles to Improve Reproducibility of Neural Networks.
[58] Gil I Shamir, Dong Lin, and Lorenzo Coviello. 2020. Smooth activations and reproducibility in deep networks. *arXiv preprint arXiv:2010.09931* (2020).
[59] Robert R Snapp and Gil I Shamir. 2021. Synthesizing Irreproducibility in Deep Networks. *arXiv preprint arXiv:2102.10696* (2021).
[60] Cecilia Summers and Michael J. Dinneen. 2021. On Nondeterminism and Instability in Neural Network Optimization.
[61] Ilya Sutskever, James Martens, George Dahl, and Geoffrey Hinton. 2013. On the importance of initialization and momentum in deep learning. In *ICML*.
[62] Adith Swaminathan and Thorsten Joachims. 2015. Batch Learning from Logged Bandit Feedback through Counterfactual Risk Minimization. *JMLR* (2015).
[63] Hal R Varian and Christopher Harris. 2014. The VCG auction in theory and practice. *American Economic Review* (2014).
[64] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. 2017. Attention is all you need. In *NeurIPS*.
[65] Ruoxi Wang, Rakesh Shivanna, Derek Cheng, Sagar Jain, Dong Lin, Lichan Hong, and Ed Chi. 2021. DCN V2: Improved Deep & Cross Network and Practical Lessons for Web-scale Learning to Rank Systems. In *WWW*.
[66] Xuanhui Wang, Michael Bendersky, Donald Metzler, and Marc Najork. 2016. Learning to Rank with Selection Bias in Personal Search. In *ACM SIGIR*.
[67] Xiaofang Wang, Dan Kondratyuk, Eric Christiansen, Kris M. Kitani, Yair Alon, and Elad Eban. 2021. Wisdom of Committees: An Overlooked Approach To Faster and More Accurate Models. *arXiv preprint arXiv:2012.01988* (2021).
[68] Ronald J Williams. 1992. Simple statistical gradient-following algorithms for connectionist reinforcement learning. *Machine learning* (1992).
[69] Yuanzhong Xu, HyoukJoong Lee, Dehao Chen, Hongjun Choi, Blake Hechtman, and Shibo Wang. 2020. Automatic cross-replica sharding of weight update in data-parallel training. *arXiv preprint arXiv:2004.13336* (2020).
[70] Y. You, J. Li, S. Reddi, J. Hseu, S. Kumar, S. Bhojanapalli, X. Song, J. Demmel, K. Keutzer, and C. Hsieh. 2019. Large batch optimization for deep learning: Training bert in 76 minutes. *arXiv preprint arXiv:1904.00962* (2019).
[71] Haichao Yu, Zhe Chen, Dong Lin, Gil Shamir, and Jie Han. 2021. Dropout Prediction Variation Estimation Using Neuron Activation Strength. *arXiv preprint arXiv:2110.06435* (2021).
[72] Manzil Zaheer, Sashank Reddi, Devendra Sachan, Satyen Kale, and Sanjiv Kumar. 2018. Adaptive methods for nonconvex optimization. *NeurIPS* (2018).
[73] Hao Zheng, Zhanlei Yang, Wenju Liu, Jizhong Liang, and Yanpeng Li. 2015. Improving deep neural networks using softplus units. In *IJCNN*.
[74] Guorui Zhou, Xiaoqiang Zhu, Chenru Song, Ying Fan, Han Zhu, Xiao Ma, Yanghui Yan, Junqi Jin, Han Li, and Kun Gai. 2018. Deep interest network for click-through rate prediction. In *SIGKDD*.
[75] Donglin Zhuang, Xingyao Zhang, Shuaiwen Leon Song, and Sara Hooker. 2021. Randomness in neural network training: Characterizing the impact of tooling. *arXiv preprint arXiv:2106.11872* (2021).
[76] Barret Zoph, Vijay Vasudevan, Jonathon Shlens, and Quoc V Le. 2018. Learning transferable architectures for scalable image recognition. In *CVPR*.

# Towards Practical Second Order Optimization for Deep Learning

**Anonymous authors**
Paper under double-blind review

## Abstract

Optimization in machine learning, both theoretical and applied, is presently dominated by first-order gradient methods such as stochastic gradient descent. Second-order optimization methods, that involve second derivatives and/or second order statistics of the data, are far less prevalent despite strong theoretical properties, due to their prohibitive computation, memory and communication costs. In an attempt to bridge this gap between theoretical and practical optimization, we present a scalable implementation of a second-order preconditioned method (concretely, a variant of full-matrix Adagrad), that along with several critical algorithmic and numerical improvements, provides significant convergence and wall-clock time improvements compared to conventional first-order methods on state-of-the-art deep models. Our novel design effectively utilizes the prevalent heterogeneous hardware architecture for training deep models, consisting of a multicore CPU coupled with multiple accelerator units. We demonstrate superior performance compared to state-of-the-art on very large learning tasks such as machine translation with Transformers, language modeling with BERT, click-through rate prediction on Criteo, and image classification on ImageNet with ResNet-50.

## 1 Introduction

Second order methods are among the most powerful algorithms in mathematical optimization. Algorithms in this family often use a preconditioning matrix to transform the gradient before applying each step. Classically, the preconditioner is the matrix of second-order derivatives (i.e., the Hessian) in the context of exact deterministic optimization (e.g., Fletcher, 2013; Lewis & Overton, 2013; Nocedal, 1980). While second-order methods often have significantly better convergence properties than first-order methods, the size of typical problems prohibits their use in practice, as they require quadratic storage and cubic computation time for each gradient update. Approximate algorithms such as quasi-Newton methods are aimed at significantly reducing these requirements; nonetheless, they still impose non-trivial memory costs equivalent to storing several copies of the model (and often quadratic computation, as in the popular two-loop recursion (Nocedal, 1980)), which severely limits their use at the immense scale of present-day deep learning.

Arguably, one of the greatest challenges of modern optimization is to bridge this gap between theoretical and practical optimization towards making second-order methods feasible to implement and deploy at immense scale. Besides the compelling scientific and mathematical developments it may stimulate, this challenge has also a clear real-world significance: recent practice of training deep learning models suggests that the utility of common first-order methods is quickly reaching a plateau, in large part because their time-per-step is already negligible (compared to other parts of the computation) and cannot be optimized further; thus, the only way to obtain faster training performance is by drastically reducing the number of update steps. To this end, utilizing second-order methods seem a very natural and promising approach.

In this paper we attempt to narrow the gap between theory and practice of second-order methods, focusing on second-order *adaptive* methods for stochastic optimization. These methods can be thought of as full-matrix analogues of common adaptive algorithms such as AdaGrad (Duchi et al., 2011; McMahan & Streeter, 2010) and Adam (Kingma & Ba, 2014): they precondition each gradient with a second moment matrix, akin to a covariance matrix, that accumulates the outer products of the stochastic gradients. Full-matrix versions are potentially more powerful than first-order methods as they can exploit statistical correlations between (gradients of) different parameters; geometrically, they can scale and rotate gradients whereas first order methods only scale gradients. However they suffer from similar prohibitive runtime and memory costs as Hessian-based methods.

Recent developments in the space of second-order methods, on which we focus on in this paper, include the K-FAC (Heskes, 2000; Martens & Grosse, 2015) and Shampoo (Gupta et al., 2018) algorithms that exploit the structure of deep networks (and more generally, models described by a collection of tensors) for mitigating the space and runtime costs of full-matrix second-order algorithms. These methods approximate each preconditioning matrix using a factored representation that stems from the network structure. However, in very large applications, such algorithms are still impractical due to a number of numerical and infrastructural pitfalls and are difficult to parallelize.

*Contributions.* We provide solutions to practical concerns and challenges that arise in implementing and using second-order methods at large scale. Our focus will be on the Shampoo algorithm, but most of the challenges we address are relevant to the implementation of many other second-order methods. These include:

- We design and implement an pipelined version of the optimization algorithm, critically exploiting the heterogeneity and computing power of CPU-Accelerator coupled architectures;
- We extend Shampoo in a number of ways so as to make it applicable to a larger range of deep architectures; in particular, the extensions allow Shampoo to be used for training very large layers such as embedding layers ubiquitous in language and translation models;
- We replace expensive spectral decompositions (e.g., SVD) used for manipulating preconditioners with an efficient and numerically-stable iterative method for computing roots of PSD matrices;
- We describe practical challenges and limitations we faced in our design, which we argue could be useful for the design considerations of next-generation accelerator hardware architectures.

Our distributed implementation demonstrates significant improvements in performance, both in terms of number of steps, and often in actual wall-clock time, on some extremely large deep learning tasks:

- *Machine translation*: we train Transformer models (Vaswani et al., 2017) on the WMT'14 English to French translation task (Bojar et al., 2014) in *half as many steps* compared to state-of-the-art (well tuned Adam), resulting with *up to 45% reduction in wall-time*.
- *Language modeling*: we trained BERT (Devlin et al., 2018) in *16% fewer steps* and achieve *higher masked-LM accuracy* compared to state-of-the-art optimizer (You et al., 2019) at 32K batch size; *overall wall-time decreased by 4% from 3.8 to 3.65 hours*. (For this task, our system has not yet been tuned for performance; we discuss several possible optimizations below.)
- *Click-Through Rate (CTR) prediction*: we trained the DLRM model (Naumov et al., 2019) on the terabyte Criteo dataset (Criteo Labs, 2015) at 64K batch size in *half as many steps* as the current state-of-the-art optimizer, with a *wall-time reduction of 37.5%*. We achieve a new state-of-the-art performance of 80.56% AUC ($\approx$ 0.3% improvement) on this task. (An improvement of 0.1% is considered significant; see Rong et al., 2020; Wang et al., 2017.)
- *Image classification*: we achieve MLPerf target accuracy of 75.9% (Mattson et al., 2019) at 32K batch size on the standard ResNet-50 ImageNet benchmark in *10% fewer steps* than previous state-of-the-art. Here we do not see wall-time gains, mainly because the problem is too small (only few thousand steps for convergence which does not allow for amortization of costs). However, we expect that one would be able to better exploit parallelism via improved software and hardware support.

We note that one of our main points in this work was to demonstrate wall-time speedups with second-order methods implemented on a *real-world distributed setup* being used to train state-of-the-art deep models. In our view, this is important for influencing future hardware accelerator design and runtime software. Indeed, first-order methods have received huge investments in tuning, implementation, platform support and tailored accelerator hardware over the last decade; we believe there are numerous opportunities to improve the per-step time performance of preconditioned methods as well. For example, our results provide a concrete justification for incorporating 64bit accumulation units in hardware for distributed training, adding larger on-chip memory, better model parallelism and tighter coupling between accelerators and CPUs, which would make second order methods feasible across more domains and models.

*Related work.* Classic techniques for addressing the high storage and computation costs of second-order methods mostly belong to the quasi-Newton or the trust-region families of algorithms (Conn et al., 2000; Nocedal & Wright, 2006). Traditionally, these methods need nearly-accurate gradients in order to construct useful quadratic approximations and implement reliable line searches, rendering them as suitable for training with very large batch sizes, and resulting in expensive iterations that make the overall algorithm slow compared with stochastic first-order methods (see, e.g., Bollapragada et al., 2018 for a recent account). Hence, our focus in this paper is on adaptive second-order methods which are directly applicable in a stochastic setting. That said, our effort could be relevant to quasi-Newton and trust-region methods as well: e.g., each iteration of typical trust-region methods amounts to solving a certain generalized eigenvalue problem, which presents numerical difficulties of similar nature to those encountered in matrix root/inverse computations, being addressed here.

Various approximations to the preconditioning matrix have been proposed in the recent literature (e.g., Gonen & Shalev-Shwartz, 2015; Erdogdu & Montanari, 2015; Agarwal et al., 2016; Xu et al., 2016; Pilanci & Wainwright, 2017). However, so far the only prevalent and pragmatic approximation is the diagonal approximation. Some recent approaches for approximating a full-matrix preconditioner are K-FAC (Martens & Grosse, 2015), Shampoo (Gupta et al., 2018) and GGT (Agarwal et al., 2018). K-FAC uses a factored approximation of the Fisher-information matrix as a preconditioner. While our focus in this paper is on Shampoo, we believe that many of the techniques presented here could also be applied to make K-FAC practical in large scale (see Appendix C). GGT uses a clever trick to compute a low-rank approximation to the AdaGrad preconditioner. However, GGT maintains several hundred copies of the gradient in memory, which is too expensive even for mid-sized models.

Ba et al. (2017) took a first important step at experimenting with distributed K-FAC for training deep models, using a single machine with 8 GPUs to simulate a distributed environment for training. In contrast, a main thrust of our work is to demonstrate wall-time speedups with second-order methods on a real-world distributed setup used for training state-of-the-art deep models, that call for design considerations crucially different than in (Ba et al., 2017). More recently, Osawa et al. (2019) scaled up K-FAC for training convolutional networks, but fell short of reaching the accuracy of first order methods, despite making changes to data augmentation and model architecture.

## 2 Preliminaries

*Adaptive preconditioning methods.* First order methods iteratively update the parameters solely based on gradient information: $w_{t+1} = w_t - \eta_t \bar{g}_t$ where $w_t$ and $\bar{g}_t$ are (column) vectors in $\mathbb{R}^d$. Here $\bar{g}_t$ denotes a linear combination of the current and past gradients $g_1, \ldots, g_t$, where different algorithms use different combinations. Preconditioned methods take the form $w_{t+1} = w_t - P_t \bar{g}_t$ where $P_t$ is an $d \times d$ matrix. Whereas in Newton-type methods this matrix is related to the Hessian matrix of second-order derivatives, adaptive preconditioning is based on gradient-gradient correlations.

The parameters of a deep network are structured as a set of tensors of order two (i.e., a matrix), three, or four. For simplicity of presentation we focus on the matrix case—however our design, analysis, and implementation hold for tensors of arbitrary order. We denote the space of parameters by the matrix $W \in \mathbb{R}^{m \times n}$ and an estimate of its gradient by $G$. Full matrix Adagrad flattens $W, G$ to vectors of dimension $mn$, it thus requires $m^2n^2$ space to store the preconditioner and $m^3n^3$ time to perform the update. $m$ and $n$ are in the 1000's in state-of-the-art models, thus rendering full-matrix preconditioning impractical. For this reason, both AdaGrad and Adam constrain the preconditioning matrices to be diagonal. Shampoo bridges the gap between full matrix preconditioning and the diagonal version by approximating the matrices.

*The Shampoo algorithm.* We describe Shampoo in the context of the Online Convex Optimization (OCO) framework, which generalizes stochastic optimization (see, e.g., Shalev-Shwartz, 2012; Hazan, 2016). In OCO, learning progresses in rounds where on round $t$ the learner receives an input $X_t$ and then uses the parameters $W_t$ to form a prediction denoted $\hat{y}_t$. After making the prediction, the true outcome $y_t$ is revealed. The discrepancy between the true and predicted outcomes is assessed by a loss function $\ell$ which takes values in $\mathbb{R}_+$. The learner then uses the discrepancy to update the matrix to $W_{t+1}$ and prepare for the next round. For instance, the input on round $t$ can be an example $x_t \in \mathbb{R}^n$ for which the learner predicts $\hat{y} = f(W_t, x_t)$ where $f : \mathbb{R}^m \to \mathbb{R}$ and the loss is a function $\ell : \mathbb{R} \times \mathbb{R} \to \mathbb{R}_+$ such as $\ell(\hat{y}, y) = (y - \hat{y})^2$ or $\ell(\hat{y}, y) = \log(1 + \exp(-y\hat{y}))$.

Stochastic gradient methods use the gradient $G_t = \nabla_W \ell(f(W, x_t), y_t)$, thus $G_t \in \mathbb{R}^{m \times n}$ if the parameters are shaped as a matrix $W \in \mathbb{R}^{m \times n}$. For matrix-shaped parameters, Shampoo tracks two statistics over the course of its run, $L_t$ and $R_t$, which are defined as follows:

$$\boxed{L_t = \epsilon I_m + \textstyle\sum_{s=1}^t \underset{\downarrow}{G_s G_s^\top}}\ ; \qquad R_t = \epsilon I_n + \textstyle\sum_{s=1}^t \underset{\downarrow}{G_s^\top} G_s\ .$$

$G^{t}_{2} \cdot G^{1}_{2}$

PSE

$L = \epsilon I + G G^T$

$U(\Lambda + G) V$

Note that $L_t \in \mathbb{R}^{m\times m}$ and $R_t \in \mathbb{R}^{n\times n}$. These are used to precondition the gradient and update $W$:

$$W_{t+1} = W_t - \eta\, L_t^{-1/4} G_t R_t^{-1/4} \,.$$

The primary complexity of Shampoo arises from the computation of $L_t^{-1/4}$ and $R_t^{-1/4}$, which was naively implemented using spectral decompositions (i.e., SVD).

## 3 Full-Matrix Preconditioning: Challenges

We discuss the main challenges and design choices in the development of the distributed implementation of Shampoo. These largely arose from the fact that modern accelerators are highly optimized for training using first-order optimizers, which have low computational and memory requirements. The Shampoo algorithm is computationally expensive. The extra computation in Shampoo compared to standard first-order methods is in the following steps:

- Preconditioner statistics computation: $L_t = L_{t-1} + G_t G_t^\mathsf{T}$ and $R_t = R_{t-1} + G_t^\mathsf{T} G_t$ ;
- Inverse $p$'th root computation: $L_t^{-1/4}$ and $R_t^{-1/4}$ ;
- Preconditioned gradient computation: $L_t^{-1/4} G_t R_t^{-1/4}$ .

Preconditioner statistics and gradient computations are expensive for large fully connected as well as embedding layers, we address these below. For other layers we show in Section 5 that they do not add significantly to the runtime of each step. Computing the inverse $p$'th roots is very slow—as much as 100 times the step time in some cases—and performing these without slowing down training was a key challenge in our system.

### 3.1 Algorithmic challenges

Modern ML architectures often use very large embedding layers, where the longer dimension can be in the millions. For example, DLRM (Naumov et al., 2019) on Criteo-1Tb uses a vocabulary with ~186 million hash buckets, while in Transformer models (Shazeer et al., 2018) the largest layer can have up to 65536 units *per* dimension. This makes preconditioning impossible due to $O(d^2)$ memory and $O(d^3)$ computational complexity. We show how to extend Shampoo to overcome these problems; we provide proofs and convergence results in Appendix B.

*Large layers.* For embedding layers specifically, we extend the Shampoo algorithm to allow us use only one of the preconditioners, in case both preconditioners are too expensive to compute. Our choice is empirically supported by the experiments shown in Figs. 2b, 3a and 5a which suggest that there is a benefit from preconditioning one dimension of the large softmax and embedding layers with minimal increase in time. The following result allows us to choose a subset of preconditioners:

Lemma 1. Let $G_1, \ldots, G_t \in \mathbb{R}^{m\times n}$ be matrices of rank at most $r$. Let $g_s = \mathrm{vec}(G_s)$ and define $\widehat{H}_t = \epsilon I_{mn} + \sum_{s=1}^t g_s g_s^\mathsf{T}$ . Let $L_t, R_t$ be defined as above: $L_t = \epsilon I_m + \sum_{s=1}^t G_s G_s^\mathsf{T}$, $R_t = \epsilon I_n + \sum_{s=1}^t G_s^\mathsf{T} G_s$ . Then for any $p, q > 0$ such that $1/p + 1/q = 1$, we have $\widehat{H}_t \preceq r L_t^{1/p} \otimes R_t^{1/q}$.

A consequence is that for any $p, q > 0$ such that $1/p + 1/q = 1$, the full AdaGrad preconditioned gradient $\widehat{H}_t^{-1/2} g_t$ is approximated by $(L_t^{1/p} \otimes R_t^{1/q})^{-1/2} g_t$, giving us $\widetilde{G}_t = L_t^{-1/2p} G_t R_t^{-1/2q}$. Now, by choosing $(p,q) = (1,\infty)$ and $(p,q) = (\infty, 1)$ we obtain the simple preconditioned gradients: $G_t R_t^{-1/2}$ and $L_t^{-1/2} G_t$. Theorem 3 shows that Lemma 1 can be used to prove a regret bound for this extended Shampoo in the online convex optimization setting – this provides intuitive justification for the usefulness of this approximation. We further optimize the computation of these preconditioned gradients for embedding layers by taking advantage of the sparse inputs, see details in Appendix D.

*Preconditioning blocks from large tensors.* In addition to embedding layers, large models occasionally have large fully connected layers. To reduce the computational cost of computing statistics and preconditioned gradient: we divide the tensor into blocks and treating individual block as a separate tensor. Concretely this would entail dividing tensor $W \in \mathbb{R}^{km\times kn}$, into $W_{1,1} \ldots W_{m,n}$ such that $W_{i,j} \in \mathbb{R}^{k\times k}$ $\forall i,j$. Shampoo still converges in this case in the convex setting (Theorem 4), showing that the extension is justified.

Lemma 2. Assume that $g_1, \ldots, g_t \in \mathbb{R}^{mk}$ are vectors, and let $g_i = [g_{i,1}, \ldots, g_{i,k}]$ where $g_{i,j} \in \mathbb{R}^m$. Define $\widehat{H}_t = \epsilon I_{mn} + \sum_{s=1}^t g_s g_s^\mathsf{T}$, and let $B_t \in \mathbb{R}^{mk\times mk}$ be the block diagonal matrix with $k$ $m\times m$ blocks, where the $j$-th block is $B_t^{(j)} = \epsilon I_m + \sum_{s=1}^t g_{s,j} g_{s,j}^\mathsf{T}$ . Then $\widehat{H}_t \preceq k B_t$.

We performed experiments to study the effect of partitioning intermediate layers into blocks, in which we observed that the latter had minimal impact on quality of the solution while providing faster step time as well as reduced memory overheads; see Fig. 3b.

*Delayed preconditioners.* As remarked above, computing the preconditioners is the most expensive computation in every Shampoo step. In Fig. 3c we show that we can compute the preconditioners once every few hundred steps without a significant effect on the accuracy which indicates that the loss function landscape does not change significantly with each step. We observe that there is a performance/quality tradeoff here — in our experiments we set the frequency of computing preconditioners to the smallest value that does not degrade performance, i.e. the number of training steps that can be completed in the amount of time needed to compute the largest preconditioner. The only way to increase the frequency of computing preconditioners is with better hardware/software support.

### 3.2 Numerical challenges

Inverse $p$'th roots (where typically $p = 2, 4, 8$) can be computed using SVD, but there are efficient iterative algorithms such as the coupled Newton iteration algorithm (Guo & Higham, 2006; Iannazzo, 2006) that can compute the inverse $p$'th root via a sequence of matrix-vector and matrix-matrix products, which are highly optimized on modern accelerators. However, our experiments suggest that on real workloads the condition numbers of the $L_t, R_t$ matrices are very large (see Fig. 6 in Appendix E) so both SVD and the coupled iteration must be run in double-precision, but this is very expensive on accelerators. We applied several further optimizations to speedup the coupled Newton iteration in our implementation; these are described in Appendix E.

### 3.3 Infrastructural challenges

*Heterogeneous training hardware.* Neural network accelerators are custom designed to run machine learning workloads faster and at lower cost. Accelerator design is trending towards preferring lower-precision (8-bit/16-bit) arithmetic that satisfy both of these goals on existing benchmarks. Our method demands double-precision arithmetic as described above, which makes running computation on accelerators a non-starter, and therefore we had to design the system to leverage the existing underutilized CPUs attached to the accelerators (Section 4).

*API inflexibility.* Deep learning libraries such as TensorFlow (Abadi et al., 2016) offer APIs for optimizer implementation that are well suited for first-order optimizers and for mini-batch training. Our design requires that we interact with the training loop in non-standard ways, which requires framework level changes. Our Transformer experiments were carried out in the Lingvo (Shen et al., 2019) TensorFlow framework, while BERT-Large, DRLM, as well as ResNet-50 used the MLPerf v0.7 Tensorflow baselines (Mattson et al., 2019). Experimentation required changes to the training loop such as gathering statistics at regular intervals, distributing computation across all the CPUs available in the cluster without blocking the TPU training, as well as updating the preconditioners. We anticipate that this proof-of-concept for full-matrix preconditioning will encourage the development of more flexible API's to fully utilize heterogeneous hardware.

## 4 Distributed System Design

We present our distributed system design of the modified Shampoo algorithm. Our method is designed to run effectively on modern neural network accelerators such as TPUs (Jouppi et al., 2017) or GPUs. We first describe the standard paradigm of data parallelism used in training models on these accelerators (Dean et al., 2012). Parameters are replicated on each core of the accelerator, and each core computes forward propagation and back propagation on a sub-batch (a subset of a mini-batch, which itself is a small randomly selected subset of the training set) of input examples. These gradients are averaged across all cores via all-reduction to get the average gradient for the mini-batch. Each core uses the average mini-batch gradient to update its copy of the parameters.

All-reduction adds a barrier as all the cores need to synchronize to compute the mini-batch gradient. In Fig. 2b we measure the overhead of each of the steps on a Transformer model (Vaswani et al., 2017) described in the experiment section. We observe that the overheads from all-reduction and weight updates are a minor part ($< 5\%$) of the overall step time.

The overall design of our implementation is illustrated by the timeline in Fig. 1. As discussed in the previous section the preconditioner computation (inverse $p$th root) is expensive and requires double

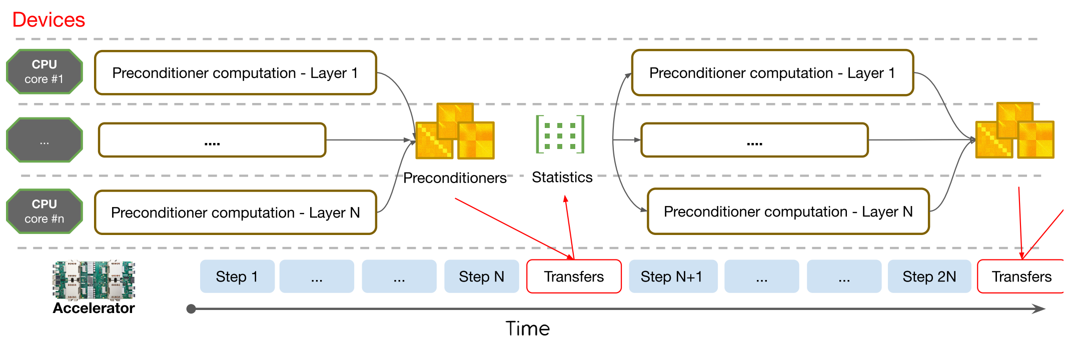


Figure 1: Timeline illustrating the design of the optimization algorithm. Preconditioner statistics ($L_t$ and $R_t$) are computed at each step by the accelerators. Preconditioners ($L_t^{1/4}$ and $R_t^{1/4}$) are only computed every $N$ steps and this computation is distributed to all available CPU cores.

precision, also we need to do this computation once every few hundred steps. These observations naturally suggested using the often underutilized CPUs on the machines to which the accelerators such as GPUs or Cloud TPUs are attached. CPUs offer double precision arithmetic but are slower than GPUs or Cloud TPUs, which makes them a perfect choice to run the preconditioner computation without adding any extra cost to the training run, as the computation is pipelined and runs asynchronously without blocking the training loop.

Preconditioners need to be computed for every layer of the network so we distribute the computation across all the CPUs that are part of the training system. As a result, the most expensive step in Shampoo adds almost nothing to the overall training time. Moreover, the computational overhead of preconditioned gradient is independent of the batch size. Thus, increasing the batch size allows us to linearly decrease the overhead making Shampoo practical for very large scale training setups. On smaller problems such as CIFAR-10, we find that our design still results in training time improvements (Appendix G.3) as preconditioner computations take very little time.

## 5 Experiments

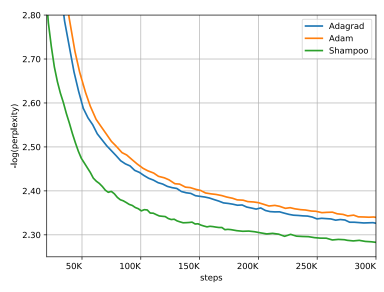
We compare our method against various widespread optimization algorithms for training large state-of-the-art deep models for machine translation, language modeling, recommendation systems as well as image classification. Details of the experiments are given in Appendix G and we will opensource our code before publication.

### 5.1 Machine Translation with a Transformer

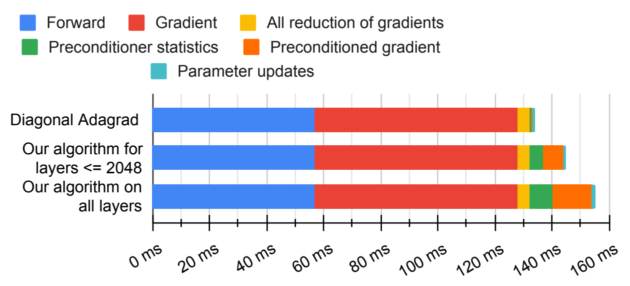
We demonstrate the effectiveness of our implementation on the standard machine translation dataset from WMT'14 English to French (en→fr) with 36.3M sentence pairs. We used the state-of-the-art Transformer architecture (Vaswani et al., 2017). This architecture contains 93.3M parameters and consists of 6 layers for its encoder and decoder. Each layer is composed of 512 model dimensions, 2048 hidden dimensions, and 8 attention heads. The model makes use of a sub-word vocabulary that contains 32K word pieces (Schuster & Nakajima, 2012). The experiment was run on 32 cores of a Cloud TPU v3 Pod, and the implementation of the optimizer was carried out in the Lingvo (Shen et al., 2019) sequence to sequence modeling based on TensorFlow. Our results are shown in Fig. 2a: our algorithm achieves the same accuracy as AdaGrad or Adam in about half as many steps.

*Preconditioning of embedding and softmax layers.* Following the first extension in Section 3.1 the algorithm preconditions the large layers with only one of the preconditioners ($G_t R_t^{-1/2}$ or $L_t^{-1/2} G_t$) to make it tractable. Fig. 2b shows the increase in step time is only 6% while Fig. 3a shows that we can reduce the number of steps to convergence by $\approx$20%.

*Reducing overhead in fully-connected layers.* Following the second extension in Section 3.1 we ran two experiments where we partitioned fully connected layer of size [512, 2048] into two blocks of size [512, 1024] and four blocks of size [512, 512]. Our experiments show no drop in quality under this approximation with a small reduction in runtime ($<$3%).

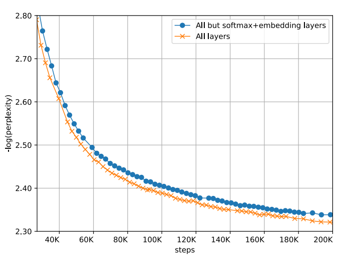


(a)

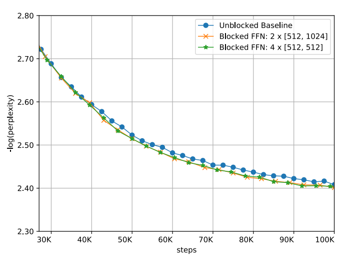


(b)

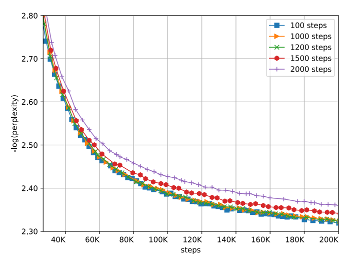
Figure 2: Results for a Transformer model on WMT'14 en→fr, trained with batch size of 1536. (a) Test log-perplexity vs. number of steps; the algorithm converges 1.95x faster in steps, while being only ≈ 16% slower per step. This allows the method to attain a particular log-perplexity in *40% less wall-time*. (b) Detailed breakdown of latency of a single step (Appendix G.6). Diagonal AdaGrad optimizer: 134ms, Shampoo: 145ms (all layers except embedding and softmax layers) and 155ms (all layers). Preconditioner computation is pipelined and distributed over CPUs, thus not adding any overhead, and transfer latency (≈100ms) is amortized over hundreds of steps.



(a)



(b)



(c)

Figure 3: Impact of Shampoo extensions on WMT'14 en→fr training: (a) preconditioning applied to all layers except embedding and softmax layers, vs. applied to all layers; (b) preconditioning with fully-connected layers partitioned into sub-blocks; (c) varying interval between preconditioner updates.

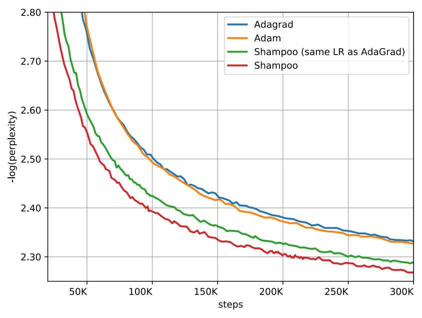
### 5.2 Transformer-Big model

We also ran experiments with a larger Transformer model with 375.4M parameters, consisting of 6 layers for its encoder and decoder. Each layer is composed of 1024 model dimensions, 8192 hidden dimensions, and 16 attention heads. Results are presented in Fig. 4a where again we see an improvement in the end-to-end wall-clock time. For the softmax, embedding and the projection fully-connected layer (with 8192 hidden dimensions) we only make use of the left preconditioner. We note that step time is dominated by the preconditioned gradient computation which can be reduced by sub-blocking the layers.

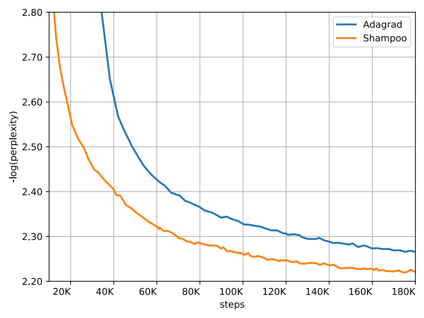
*On the overhead of the optimizer.* We show the computational and memory complexity of the Shampoo extensions described in Section 3.1 in Table 2 in the appendix. We note that the overhead from computing the statistics, as well as from computing the preconditioned update for single step of training, can be further reduced by increasing the batch sizes (indeed, these overheads are independent of the batch size) as shown in Fig. 4b where the overhead dramatically reduces from 40% to 19%.

### 5.3 Ads Click-Through Rate (CTR) prediction

We trained the Deep Learning Recommendations Model (DLRM) of Naumov et al. (2019) on the terabyte Criteo click logs dataset for online advertisement click-through-rate prediction task (Criteo Labs, 2015). We compared Shampoo against the highly tuned SOTA baseline from MLPerf v0.7 training benchmarks (Wu et al., 2020). We trained the model with a batch size of 65536 for 64000 steps (1 epoch). We trained a version of the model where Shampoo is applied only to the hidden layers as well as one where we apply it for all layers. We only tune the learning rate, and keep the exact same setup as the baseline. We found that Shampoo achieves the target accuracy of 80.25% in only 30.97K steps compared to 64K steps for the baseline. Moreover, Shampoo achieves new



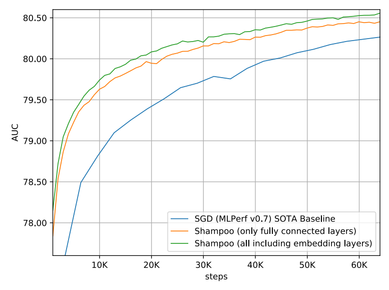
(a) Batch size: 384



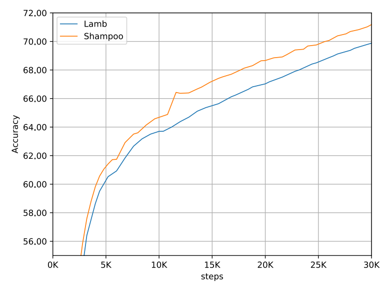
(b) Batch size: 1536

Figure 4: Test log-perplexity of a Transformer-Big model on WMT'14 en→fr. (a) Shampoo converges faster than AdaGrad (≈2x faster in steps), and allows larger learning rates; due to the large overhead in step time, this results in only 30% improvement in wall-time. (b) Larger batch sizes reduce the optimizer overhead from 40% to 19%, resulting in an *end-to-end improvement of 41%* in wall-time for convergence.

state-of-the-art performance of 80.56% AUC (an ≈0.3% improvement) on this dataset, note that an improvement of 0.1% is considered significant in this task; see Rong et al., 2020; Wang et al., 2017. Here preconditioning embedding layers further reduced the number of steps needed to reach the target accuracy from 39.96K to 30.97K.



(a) Test AUC on the Criteo-1Tb dataset.



(b) Masked Language accuracy on BERT-Large.

Figure 5: (a) Shampoo reaches a target AUC of 80.25% in half as many steps with preconditioning embedding layers improving the results, and achieves a new state-of-the-art AUC of 80.56%; (b) Shampoo converges in ≈16% fewer steps, and achieves ≈1% higher MLM accuracy than the baseline on BERT-Large.

## 5.4 Language modeling

We trained BERT-Large (the Bidirectional Encoder Representation architecture of Devlin et al., 2018) for the language modeling task on the concatenation of Wikipedia and BooksCorpus, with 2.5B and 800M words respectively. BERT-Large is a large bidirectional transformer model containing 24 transformer blocks with 1024 hidden dimensions and 16 self attention heads. It has 340M parameters and is set up to jointly optimize two objectives: (a) masked language model (Masked-LM) loss where the task is to predict masked tokens based on surrounding context, and (b) next sentence prediction (NSP) loss where the task is to predict whether two given sentences are consecutive in the text. In Fig. 5b we compare our results against the current state of the art in training BERT (You et al., 2019). Models were trained with batch size 16K; in these experiments we replaced the Adam update rule in Lamb that produces the preconditioned gradient with Shampoo. Both experiments used existing well-tuned hyperparameters of the baseline.

## 5.5 Image classification

We trained a ResNet-50 model (He et al., 2016) on the ImageNet-2012 (Russakovsky et al., 2015) dataset and compared it against the state-of-the-art baseline using SGD+Momentum. We base our experiments off the Tensorflow baseline available from Mattson et al. (2019) where the target criteria is reaching 75.9% accuracy. See results in Table 1; in particular, we find that Shampoo reaches the target accuracy in fewer steps than the current state of the art. Tuning details are in Appendix G.4.

| Optimizer | Batch Size | Epochs | Steps |
|---|---|---|---|
| SGD+Momentum | 4096 | 85 | 26586 |
| LARS | 4096 | 45 | 14040 |
| LARS | 32768 | 64 | 2512 |
| Shampoo | 4096 | 45 | 14040 |
| Shampoo | 16384 | 48 | 3744 |
| **Shampoo** | **32768** | **58** | **2262** |

Table 1: Epochs and steps to MLPerf target accuracy of 75.9% with a ResNet-50.

## 6 Concluding Remarks

We have presented an implementation of a second order optimizer, and demonstrated step time as well as wall time improvements on multiple large tasks in different domains — in each case our implementation performed as well or better than state-of-the-art optimizers specialized for each domain. The main point of our work is to demonstrate that second order methods implemented on a *real-world distributed setup* can be used to train state-of-the-art deep models. We hope that this work will influence future hardware accelerator design and runtime software — first order methods have received large investments in tuning, implementation, platform support and hardware tailored for them, and we believe there are several opportunities to improve the per-step time performance of second order methods as well:

- Most second order methods use symmetric matrices, but we haven't found support for typing operands as symmetric, which can reduce compute flops and storage by upto 50%.
- Several optimizations that are currently tuned towards first order methods could be extended to second order methods. For example, weight update sharding pattern matches first order methods (Xu et al., 2020) and dramatically reduces the time spent in the update step as well as memory used. This change can also be applied to Shampoo with blocked preconditioners – but we do not have support for it yet as it requires compiler level support, and is not expressible at the program layer. Currently every core must update all layers which is quite inefficient.
- Mixed precision algorithms may work for inverse pth roots and can help increase the frequency of preconditioner computation.
- Increased memory per chip can allow larger preconditioners.
- Hardware support for high-precision arithmetic in accelerators can allow more frequent preconditioner computation. The benefits of high precision arithmetic for optimization run counter to the prevailing wisdom in ML[1] which has led to the focus on low-precision formats such as bfloat16 (Wang & Kanwar, 2019).
- Hardware support for storing/packing and using upper/lower triangular matrices efficiently, as available in LAPACK.

Our hope is that these suggestions could result in innovations that would make second-order methods practical across more domains and models, especially in data limited regimes where we may not able to amortize the latency added in the data transfer between the accelerator and the CPU.

[1]For example, (Gupta et al., 2015) say "It is well appreciated that in the presence of statistical approximation and estimation errors, high-precision computation in the context of learning is rather unnecessary (Bottou & Bousquet, 2007)" and (Higham & Pranesh, 2019) say "... machine learning provides much of the impetus for the development of half precision arithmetic in hardware ..."

## References

Martin Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, Manjunath Kudlur, Josh Levenberg, Rajat Monga, Sherry Moore, Derek G. Murray, Benoit Steiner, Paul Tucker, Vijay Vasudevan, Pete Warden, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. Tensorflow: A system for large-scale machine learning. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, pp. 265–283, 2016.

Naman Agarwal, Brian Bullins, and Elad Hazan. Second order stochastic optimization in linear time. *arXiv preprint arXiv:1602.03943*, 2016.

Naman Agarwal, Brian Bullins, Xinyi Chen, Elad Hazan, Karan Singh, Cyril Zhang, and Yi Zhang. The case for full-matrix adaptive regularization. *CoRR*, abs/1806.02958, 2018.

Naman Agarwal, Rohan Anil, Elad Hazan, Tomer Koren, and Cyril Zhang. Disentangling adaptive gradient methods from learning rates. *arXiv preprint arXiv:2002.11803*, 2020.

Tsuyoshi Ando, Chi-Kwong Li, and Roy Mathias. Geometric means. *Linear algebra and its applications*, 385:305–334, 2004.

Jimmy Ba, James Martens, and Roger Grosse. Distributed second-order optimization using kronecker-factored approximations. In *International conference on machine learning*, pp. 2408–2417, 2017.

Ondrej Bojar, Christian Buck, Christian Federmann, Barry Haddow, Philipp Koehn, Johannes Leveling, Christof Monz, Pavel Pecina, Matt Post, Herve Saint-Amand, Radu Soricut, Lucia Specia, and Ale s Tamchyna. Findings of the 2014 workshop on statistical machine translation. In *Proceedings of the Ninth Workshop on Statistical Machine Translation*, pp. 12–58, Baltimore, Maryland, USA, June 2014. Association for Computational Linguistics. URL http://www.aclweb.org/anthology/W/W14/W14-3302.

Raghu Bollapragada, Jorge Nocedal, Dheevatsa Mudigere, Hao-Jun Shi, and Ping Tak Peter Tang. A progressive batching l-bfgs method for machine learning. In *International Conference on Machine Learning*, pp. 620–629, 2018.

Andrew R Conn, Nicholas IM Gould, and Philippe L Toint. *Trust region methods*. SIAM, 2000.

Criteo Labs. Criteo releases industry's largest-ever dataset for machine learning to academic community, July 2015. URL https://www.criteo.com/news/press-releases/2015/07/criteo-releases-industrys-largest-ever-dataset/.

Jeffrey Dean, Greg Corrado, Rajat Monga, Kai Chen, Matthieu Devin, Mark Mao, Marc' Aurelio Ranzato, Andrew Senior, Paul Tucker, Ke Yang, Quoc V. Le, and Andrew Y. Ng. Large scale distributed deep networks. *Advances in Neural Information Processing Systems 25*, 2012.

Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. Bert: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805*, 2018.

John Duchi, Elad Hazan, and Yoram Singer. Adaptive subgradient methods for online learning and stochastic optimization. *Journal of Machine Learning Research*, 12(Jul):2121–2159, 2011.

Murat A Erdogdu and Andrea Montanari. Convergence rates of sub-sampled newton methods. In *Proceedings of the 28th International Conference on Neural Information Processing Systems-Volume 2*, pp. 3052–3060. MIT Press, 2015.

Roger Fletcher. *Practical methods of optimization*. John Wiley & Sons, 2013.

Thomas George, César Laurent, Xavier Bouthillier, Nicolas Ballas, and Pascal Vincent. Fast approximate natural gradient descent in a Kronecker factored eigenbasis. In *Advances in Neural Information Processing Systems*, pp. 9550–9560, 2018.

Alon Gonen and Shai Shalev-Shwartz. Faster sgd using sketched conditioning. *arXiv preprint arXiv:1506.02649*, 2015.

Chun-Hua Guo and Nicholas J Higham. A Schur-Newton method for the matrix p'th root and its inverse. *SIAM Journal On Matrix Analysis and Applications*, 28(3):788–804, 2006.

Suyog Gupta, Ankur Agrawal, Kailash Gopalakrishnan, and Pritish Narayanan. Deep learning with limited numerical precision. In *International Conference on Machine Learning*, pp. 1737–1746, 2015.

Vineet Gupta, Tomer Koren, and Yoram Singer. Shampoo: Preconditioned stochastic tensor optimization. In *Proceedings of the 35th International Conference on Machine Learning*, volume 80, pp. 1842–1850, 2018.

Elad Hazan. Introduction to online convex optimization. *Foundations and Trends in Optimization*, 2 (3-4):157–325, 2016.

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

Tom Heskes. On "natural" learning and pruning in multilayered perceptrons. *Neural Computation*, 12(4):881–901, 2000.

Nicholas J Higham and Srikara Pranesh. Simulating low precision floating-point arithmetic. *SIAM Journal on Scientific Computing*, 41(5):C585–C602, 2019.

Bruno Iannazzo. On the Newton method for the matrix p-th root. *SIAM journal on matrix analysis and applications*, 28(2):503–523, 2006.

Norman P Jouppi, Cliff Young, Nishant Patil, David Patterson, Gaurav Agrawal, Raminder Bajwa, Sarah Bates, Suresh Bhatia, Nan Boden, Al Borchers, et al. In-datacenter performance analysis of a tensor processing unit. In *Computer Architecture (ISCA), 2017 ACM/IEEE 44th Annual International Symposium on*, pp. 1–12. IEEE, 2017.

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

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

Frederik Kunstner, Philipp Hennig, and Lukas Balles. Limitations of the empirical fisher approximation for natural gradient descent. In *Advances in Neural Information Processing Systems*, pp. 4156–4167, 2019.

Adrian S Lewis and Michael L Overton. Nonsmooth optimization via quasi-newton methods. *Mathematical Programming*, 141(1-2):135–163, 2013.

James Martens and Roger Grosse. Optimizing neural networks with Kronecker-factored approximate curvature. In *International conference on machine learning*, pp. 2408–2417, 2015.

Peter Mattson, Christine Cheng, Cody Coleman, Greg Diamos, Paulius Micikevicius, David Patterson, Hanlin Tang, Gu-Yeon Wei, Peter Bailis, Victor Bittorf, et al. Mlperf training benchmark. *arXiv preprint arXiv:1910.01500*, 2019.

H Brendan McMahan and Matthew Streeter. Adaptive bound optimization for online convex optimization. *COLT 2010*, pp. 244, 2010.

Maxim Naumov, Dheevatsa Mudigere, Hao-Jun Michael Shi, Jianyu Huang, Narayanan Sundaraman, Jongsoo Park, Xiaodong Wang, Udit Gupta, Carole-Jean Wu, Alisson G Azzolini, et al. Deep learning recommendation model for personalization and recommendation systems. *arXiv preprint arXiv:1906.00091*, 2019.

Jorge Nocedal. Updating quasi-newton matrices with limited storage. *Mathematics of computation*, 35(151):773–782, 1980.

Jorge Nocedal and Stephen Wright. *Numerical optimization*. Springer Science & Business Media, 2006.

Kazuki Osawa, Yohei Tsuji, Yuichiro Ueno, Akira Naruse, Rio Yokota, and Satoshi Matsuoka. Large-scale distributed second-order optimization using kronecker-factored approximate curvature for deep convolutional neural networks. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pp. 12359–12367, 2019.

Michael L Overton. *Numerical computing with IEEE floating point arithmetic*. SIAM, 2001.

Mert Pilanci and Martin J. Wainwright. Newton sketch: A near linear-time optimization algorithm with linear-quadratic convergence. *SIAM Journal on Optimization*, 27(1):205–245, 2017.

Haidong Rong, Yangzihao Wang, Feihu Zhou, Junjie Zhai, Haiyang Wu, Rui Lan, Fan Li, Han Zhang, Yuekui Yang, Zhenyu Guo, et al. Distributed equivalent substitution training for large-scale recommender systems. In *Proceedings of the 43rd International ACM SIGIR Conference on Research and Development in Information Retrieval*, pp. 911–920, 2020.

Olga Russakovsky, Jia Deng, Hao Su, Jonathan Krause, Sanjeev Satheesh, Sean Ma, Zhiheng Huang, Andrej Karpathy, Aditya Khosla, Michael Bernstein, et al. Imagenet large scale visual recognition challenge. *International Journal of Computer Vision*, 115(3):211–252, 2015.

Mike Schuster and Kaisuke Nakajima. Japanese and Korean voice search. In *ICASSP*, pp. 5149–5152. IEEE, 2012.

Shai Shalev-Shwartz. Online learning and online convex optimization. *Foundations and Trends in Machine Learning*, 4(2):107–194, 2012.

Noam Shazeer, Youlong Cheng, Niki Parmar, Dustin Tran, Ashish Vaswani, Penporn Koanantakool, Peter Hawkins, HyoukJoong Lee, Mingsheng Hong, Cliff Young, et al. Mesh-tensorflow: Deep learning for supercomputers. In *Advances in Neural Information Processing Systems*, pp. 10414–10423, 2018.

Jonathan Shen, Patrick Nguyen, Yonghui Wu, Zhifeng Chen, et al. Lingvo: a modular and scalable framework for sequence-to-sequence modeling, 2019.

Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. In *Advances in Neural Information Processing Systems*, pp. 5998–6008, 2017.

Ruoxi Wang, Bin Fu, Gang Fu, and Mingliang Wang. Deep & cross network for ad click predictions. In *Proceedings of the ADKDD'17*, pp. 1–7. 2017.

Shibo Wang and Pankaj Kanwar. Bfloat16: The secret to high performance on cloud tpus. https://cloud.google.com/blog/products/ai-machine-learning/bfloat16-the-secret-to-high-performance-on-cloud-tpus, 2019.

Carole-Jean Wu, Robin Burke, Ed Chi, Joseph Konstan, Julian McAuley, Yves Raimond, and Hao Zhang. Developing a recommendation benchmark for mlperf training and inference. *arXiv preprint arXiv:2003.07336*, 2020.

Peng Xu, Jiyan Yang, Farbod Roosta-Khorasani, Christopher Ré, and Michael W Mahoney. Sub-sampled newton methods with non-uniform sampling. In *Advances in Neural Information Processing Systems*, pp. 3000–3008, 2016.

Yuanzhong Xu, HyoukJoong Lee, Dehao Chen, Hongjun Choi, Blake Hechtman, and Shibo Wang. Automatic cross-replica sharding of weight update in data-parallel training. *arXiv preprint arXiv:2004.13336*, 2020.

Yang You, Jing Li, Sashank Reddi, Jonathan Hseu, Sanjiv Kumar, Srinadh Bhojanapalli, Xiaodan Song, James Demmel, Kurt Keutzer, and Cho-Jui Hsieh. Large batch optimization for deep learning: Training bert in 76 minutes. *arXiv preprint arXiv:1904.00962*, 2019.

## A Notation

We use lowercase letters to denote scalars and vectors, and uppercase letters to denote matrices. $\|A\|_F$ denotes the Frobenius norm of $A$, i.e., $\|A\|_F^2 = \sum_{i,j} A_{ij}^2$. $A \bullet B$ denotes the Hadamard or element-wise product of $A$ and $B$ which have the same shape, so $C = A \bullet B \iff C_{ij} = A_{ij}B_{ij}$. $D^{\odot\alpha}$ is the element-wise power, $(D^{\odot\alpha})_{ij} = D_{ij}^{\alpha}$.

We use $\preceq$ to denote the Loewner order: given square symmetric matrices $A, B$, we write $A \preceq B$ iff $B - A$ is positive semidefinite (PSD).

Given a symmetric PSD matrix $A$, and $\alpha \in \mathbb{R}$, $A^\alpha$ is defined as follows: let $A = UDU^\top$ be the singular value decomposition of $A$, where $U$ is a unitary matrix and $D$ is a diagonal matrix (with $D_{ii} \geq 0$ as $A$ is PSD), then $A^\alpha = UD^\alpha U^\top$, where $(D^\alpha)_{ii} = D_{ii}^\alpha$. If $\alpha < 0$, this is defined for positive definite matrices only, where $D_{ii} > 0$.

We use $\text{vec}(A)$ to denote the flattening of the $m \times n$ matrix $A$: if $A$ has rows $a_1, \ldots, a_m$, then $\text{vec}(A)$ is the $mn \times 1$ column vector $\text{vec}(A) = (a_1, \ldots, a_m)^\top$. $A \otimes B$ denotes the Kronecker product of two matrices $A$ and $B$, and we will use the identities $(A \otimes B)^\alpha = A^\alpha \otimes B^\alpha$ for $\alpha \in \mathbb{R}$, and $(A \otimes B)\,\text{vec}(C) = \text{vec}(ACB^\top)$.

## B Deferred Proofs

Proof (of Lemma 1). Lemma 8 in Gupta et al. (2018) shows that $\widehat{H}_t \preceq rL_t \otimes I_n$ and $\widehat{H}_t \preceq rI_m \otimes R_t$. By using Ando's inequality (Ando et al., 2004), we get

$$\begin{aligned}\widehat{H}_t &\preceq r(L_t \otimes I_n)^{1/p}(I_m \otimes R_t)^{1/q} \\ &= r(L_t^{1/p} \otimes I_n)(I_m \otimes R_t^{1/q}) \\ &= rL_t^{1/p} \otimes R_t^{1/q}\,,\end{aligned}$$

which concludes the proof. □

This lemma immediately allows us to prove a regret bound for Shampoo with extended exponents:

Theorem 3. Assume that the gradients $G_1, \ldots, G_T$ are matrices of rank at most $r$. Then the regret of Shampoo with extended exponents compared to any $W^\star \in \mathbb{R}^{m\times n}$ is bounded as follows,

$$\sum_{t=1}^{T} f_t(W_t) - \sum_{t=1}^{T} f_t(W^\star) \leq \sqrt{2r}D\,\text{Tr}(L_T^{\frac{1}{2p}})\,\text{Tr}(R_T^{\frac{1}{2q}})\,,$$

where

$$L_T = \epsilon I_m + \sum_{t=1}^{T} G_t G_t^\top\,,\quad R_T = \epsilon I_n + \sum_{t=0}^{T} G_t^\top G_t\,,\quad D = \max_{t\in[T]} \|W_t - W^\star\|_2\,.$$

and $1/p + 1/q = 1, p, q \geq 1$.

Proof. The proof follows the proof of Theorem 7 in Gupta et al. (2018). Let $H_t = L_t^{\frac{1}{2p}} \otimes R_t^{\frac{1}{2q}}$. Then the update rule of the extended Shampoo algorithm is equivalent to $w_{t+1} = w_t - \eta H_t^{-1} g_t$. Since $0 \preceq L_1 \preceq \ldots \preceq L_T$ and $0 \preceq R_1 \preceq \ldots \preceq R_T$, standard properties of the Kronecker product and the operator monotonicity of the function $x \mapsto x^\alpha$ for $\alpha \leq 1$ (an immediate consequence of Ando's inequality) ensure that $0 \preceq H_1 \preceq \ldots \preceq H_T$.

Following the aforementioned proof, we have the regret bound

$$\sum_{t=1}^{T} f_t(W_t) - \sum_{t=1}^{T} f_t(W^\star) \leq \frac{D^2}{2\eta}\,\text{Tr}(H_T) + \frac{\eta}{2}\sum_{t=1}^{T} \|g_t\|_{H_t^*}^2,$$

where $D = \max_t \|W_t - W^\star\|_2$. Define $g_t = \text{vec}(G_t)$ and $\widehat{H}_t = (\epsilon I_m + \sum_{s=1}^{t} g_s g_s^\top)^{1/2}$, then Lemma 1 shows that $\widehat{H}_t \preceq \sqrt{r}H_t$, using operator monotonicity. Using this equation twice, along with Equation (6) from the proof of Theorem 7, we have

$$\sum_{t=1}^{T} \|g_t\|_{H_t^*}^2 \leq \sqrt{r}\sum_{t=1}^{T} \|g_t\|_{\widehat{H}_t^*}^2 \leq 2\sqrt{r}\,\text{Tr}(\hat{H}_T) \leq 2r\,\text{Tr}(H_T).$$

This gives us

$$\sum_{t=1}^{T} f_t(W_t) - \sum_{t=1}^{T} f_t(W^\star) \le \frac{D^2}{2\eta}\operatorname{Tr}(H_T) + \eta r \operatorname{Tr}(H_T).$$

Setting $\eta = D/\sqrt{2r}$ and observing that $\operatorname{Tr}(H_t) = \operatorname{Tr}(L_t^{1/2p})\operatorname{Tr}(R_t^{1/2q})$ gives us the required bound. □

Proof (of Lemma 2). Let $x \in \mathbb{R}^{mk}$, and $x = [x_1, x_2, \ldots, x_k]$, where $x_j \in \mathbb{R}^m$. Then

$$\begin{aligned}
x^\top \widehat{H}_t x &= \epsilon\|x\|_2^2 + \sum_{s=1}^{t} x^\top g_s g_s^\top x = \epsilon\|x\|_2^2 + \sum_{s=1}^{t} (g_s^\top x)^2 = \epsilon\|x\|_2^2 + \sum_{s=1}^{t}\left(\sum_{j=1}^{k} g_{s,j}^\top x_j\right)^2 \\
&\le k\epsilon\|x\|_2^2 + k\sum_{s=1}^{t}\sum_{j=1}^{k}(g_{s,j}^\top x_j)^2 = k\sum_{j=1}^{k}\left(\epsilon\|x_j\|_2^2 + \sum_{s=1}^{t} x_j^\top g_{s,j} g_{s,j}^\top x_j\right) \\
&= k\sum_{j=1}^{k} x_j^\top\big(\epsilon I_m + \sum_{s=1}^{t} g_{s,j} g_{s,j}^\top\big)x_j = k\sum_{j=1}^{k} x_j^\top B_t^{(j)} x_j = k x^\top B_t x.
\end{aligned}$$

Here we used the inequality $\left(\sum_{j=1}^{k}\alpha_j\right)^2 \le k\sum_{j=1}^{k}\alpha_j^2$, which follows from the convexity of $x \mapsto x^2$ (or from the fact that variance of a random variable is non-negative). □

This lemma once again allows us to prove a regret bound, exactly following the proof of the regret bound above:

Theorem 4. Assume that the gradients are $g_1, \ldots, g_T \in \mathbb{R}^{mk}$, and let $g_i = [g_{i,1}, \ldots, g_{i,k}]$ where $g_{i,j} \in \mathbb{R}^m$. Then the regret of Shampoo with blocking compared to any $w^\star \in \mathbb{R}^{mk}$ is bounded as follows:

$$\sum_{t=1}^{T} f_t(w_t) - \sum_{t=1}^{T} f_t(w^\star) \le \sqrt{2k}D\sum_{j=1}^{k}\operatorname{Tr}\left(\left(\epsilon I_m + \sum_{t=1}^{T} g_{t,j} g_{t,j}^\top\right)^{\frac{1}{2}}\right).$$

The two regret bounds can be combined to show that Shampoo with both extensions also converges.

## C Comparison with K-FAC

K-FAC is a natural gradient algorithm, and approximates the curvature of the loss using the Fisher Information Matrix:

$$\mathbf{F} = \mathop{\mathbb{E}}_{p(x|\theta)}\left[\nabla \log p(x|\theta)\,\nabla \log p(x|\theta)^\top\right] = \mathop{\mathbb{E}}_{p(x|\theta)}\left[g_{p(x|\theta)}\, g_{p(x|\theta)}^\top\right].$$

For a fully connected layer with $W \in \mathbb{R}^{m\times n}$, where $Wx = s$, the gradient for the layer $G_t \in \mathbb{R}^{m\times n}$ can be written via the chain rule as $G_t = \nabla_s \ell(s_t, y_t)x^\top$ and in vectorized form as: $\nabla_s \ell(s_t, y_t) \otimes x$. We can then write the Fisher information matrix as:

$$\begin{aligned}
\mathbf{F} &= \mathop{\mathbb{E}}_{p(x|\theta)}\left[(\nabla_s \ell(s_t, y_t) \otimes x)\,(\nabla_s \ell(s_t, y_t) \otimes x)^\top\right] \\
&= \mathop{\mathbb{E}}_{p(x|\theta)}\left[(\nabla_s \ell(s_t, y_t)\nabla_s \ell(s_t, y_t)^\top) \otimes (x_t x_t^\top)\right].
\end{aligned}$$

Assuming independence between $\nabla_s \ell(s_t, y_t)$ and $x$, K-FAC rewrites the Fisher in tractable form as:

$$\mathbf{F} \approx \mathbb{E}\left[(\nabla_s \ell(s_t, y_t)\nabla_s \ell(s_t, y_t)^\top)\right] \otimes \mathbb{E}\left[x_t x_t^\top\right].$$

If we let $D = \mathbb{E}\left[(\nabla_s \ell(s_t, y_t)\nabla_s \ell(s_t, y_t)^\top)\right]$ and $X = \mathbb{E}\left[x_t x_t^\top\right]$, the update rule then becomes:

$$W_{t+1} \approx W_t - \eta\, D^{-1} G_t X^{-1}.$$

We note some of the differences and similarities between the two updates here. KFAC preconditioners use exponent of $-1$ (as original Fisher is inverted) whereas Shampoo uses $-1/2p$ where $p$ is the rank of the tensor. KFAC computes statistics based on gradients with labels sampled from the model's predictive distribution (hence requiring strictly more computation) where as Shampoo relies on the gradient of the mini-batch.

Now we can compute each term in the Shampoo preconditioners as:

$$G_tG_t^\intercal = \nabla_s\ell(s_t,y_t)x_t^\intercal x_t\nabla_s\ell(s_t,y_t)^\intercal = \|x_t\|_2^2\nabla_s\ell(s_t,y_t)\nabla_s\ell(s_t,y_t)^\intercal;$$
$$G_t^\intercal G_t = x_t\nabla_s\ell(s_t,y_t)^\intercal\nabla_s\ell(s_t,y_t)x_t^\intercal = \|\nabla_s\ell(s_t,y_t)\|_2^2 x_t x_t^\intercal.$$

Dividing by the scale, and taking expectations on both sides:

$$\mathbb{E}\left[\frac{G_tG_t^\intercal}{\|x_t\|_2^2}\right] = \mathbb{E}\left[\nabla_s\ell(s_t,y_t)\nabla_s\ell(s_t,y_t)^\intercal\right] = D;$$
$$\mathbb{E}\left[\frac{G_t^\intercal G_t}{\|\nabla_s\ell(s_t,y_t)\|_2^2}\right] = \mathbb{E}\left[x_tx_t^\intercal\right] = X.$$

This shows that K-FAC preconditioners are closely related to Shampoo preconditioners, especially when one uses the empirical Fisher (Kunstner et al., 2019).

The main difficulty in implementing K-FAC on a model is that current optimizer APIs make it difficult to send additional information such as $\|x_t\|_2^2, \|\nabla_s\ell(s_t,y_t)\|_2^2$ to the optimizer, so K-FAC implementations have to register the structure of each layer. Moreover, due to the dependence of K-FAC on the structure of the network, it is difficult to implement standard operators like batch norm, weight norm, layer norm, etc., which are prevalent in the tasks and models we considered. For example, if we write a fully connected layer with weight norm as $s = Wx/\|W\|$, then the gradient

$$G_t = \frac{1}{\|W\|}\nabla_s\ell(s_t,y_t)x^\intercal - \frac{\nabla_s\ell(s_t,y_t)^\intercal Wx}{\|W\|^3}W,$$

so rewriting $\mathbb{E}[\mathrm{vec}(G_t)\,\mathrm{vec}(G_t)^\intercal]$ as a Kronecker product is not an easy task.

The similarity between K-FAC and Shampoo preconditioners also allows us to use techniques explored by the K-FAC community for Shampoo. One of the extensions for KFAC is the E-KFAC algorithm (George et al., 2018) which constructs a better approximation of the Fisher matrix by using the eigenbasis computed from the Kronecker approximation, but rescaling the eigenvalues to match the diagonal of the Fisher matrix in this eigenbasis. This method produces a provably better approximation, and can immediately be applied to Shampoo too with a simple modification:

Let $\hat{H}_t \approx L_t^{1/2}\otimes R_t^{1/2}$. Let the singular value decompositions of the factors be $L_t^{1/2} = UDU^\intercal$ and $R_t^{1/2} = VD'V^\intercal$. The $L_t^{1/2}\otimes R_t^{1/2} = (U\otimes V)(D\otimes D')(U\otimes V)^\intercal$. Now the EKFAC correction replaces $D\otimes D'$ by the optimal diagonal

$$\begin{aligned}\Lambda &= \mathrm{diag}((U\otimes V)^\intercal\hat{H}_t(U\otimes V))\\ &= \epsilon I + \sum_{s=1}^t \mathrm{diag}((U\otimes V)^\intercal\,\mathrm{vec}(G_s)\,\mathrm{vec}(G_s)^\intercal(U\otimes V))\\ &= \epsilon I + \sum_{s=1}^t \mathrm{diag}(\mathrm{vec}(U^\intercal G_sV)\,\mathrm{vec}(U^\intercal G_sV)^\intercal)\\ &= \epsilon I + \sum_{s=1}^t \mathrm{vec}(U^\intercal G_sV)^{\odot 2},\end{aligned}$$

Thus we can approximately compute $\Lambda_{t+1} \approx \Lambda_t + (U^\intercal G_tV)^{\odot 2}$, and the new update becomes: $W_{t+1} = W_t - \eta_tU(\Lambda_t^{-1/2}\bullet(U^\intercal G_tV))V^\intercal$. This technique does have the disadvantage that it requires computing the singular value decompositions (which we already observed are much slower than coupled Newton iterations), and doubles the number of matrix multiplications in the preconditioned gradient computation. At this time our experiments did not show significant improvements over the standard Shampoo implementation, but we plan to explore this further.

## D Shampoo for embedding layers

In modern networks, embedding layers are usually very large, and even computing the left preconditioner as described in Section 3.1 can be prohibitively expensive. However we can take advantage of the fact that the inputs to the network are very sparse, and use this to reduce the computation significantly.

Let our input example to such a network consist of a set of categorical features: each feature such as user language, user country etc consists of one out of a set of options. Then the output of the embedding layer is the concatenation of the embeddings for each such feature. If the embeddings are of width $d$ and there are $N$ such embeddings, then the embedding layer is $W \in \mathbb{R}^{d\times N}$. The input can be represented as $x \in \mathbb{R}^{N\times m}$, where $m$ is the number of categorical features, and each column is one-hot: if the $k$-th feature is $x(k)$, then $x_{jk} = \delta_{j,x(k)}$. The output of the layer is $y = Wx$.

Now $G = \nabla_W \ell = \nabla_y \ell \, x^\mathsf{T}$, so $GG^\mathsf{T} = \nabla_y \ell \, x^\mathsf{T} x \, \nabla_y \ell^\mathsf{T}$. But $x^\mathsf{T} x = \mathbf{I}_m$, so $GG^\mathsf{T} = \nabla_y \ell \, \nabla_y \ell^\mathsf{T}$. Thus we can compute the preconditioner for $W$ by computing it on the output of the embedding layer, and this is a much smaller computation since $y$ is of dimension $b \times m$, this computation is $O(d^2 m)$ rather than $O(d^2 N)$. Note that sparse multiplication would also be $O(d^2 m)$, but accelerators usually implement sparse operations by densifying the tensors.

If each column of $x$ is multi-hot, as is the case when the features are words and their embeddings are averaged, $x^\mathsf{T} x$ is a diagonal matrix, where each diagonal entry is a function of the number of ones in each column of $x$. Computing $GG^\mathsf{T} = \nabla_y \ell (x^\mathsf{T} x) \nabla_y \ell^\mathsf{T}$ is still $O(d^2 m) \ll O(d^2 N)$.

## E A coupled Newton iteration for computation of inverse $p$-th roots

The Newton method for solving the matrix equation $X^{-p} - A = 0$ produces the iteration $X_{k+1} = \frac{1}{p}[(p+1)X_k - X_k^{p+1} A]$, where we take $X_0 = \frac{1}{c} I$. This iteration satisfies $X_k \to A^{-1/p}$ as $k \to \infty$, but it is not numerically stable. Introducing the matrix $M_k = X_k^p A$, we get

$$X_{k+1} = X_k \left( \frac{(p+1)I - M_k}{p} \right), \qquad X_0 = \frac{1}{c} I,$$

and

$$M_{k+1} = X_{k+1}^p A = \left( \frac{(p+1)I - M_k}{p} \right)^p X_k^p A = \left( \frac{(p+1)I - M_k}{p} \right)^p M_k, \qquad M_0 = \frac{1}{c^p} A,$$

since $X_k, M_k$ and $A$ commute with each other. This is the coupled Newton iteration for computing inverse $p$-th roots, and was shown to be numerically stable in (Guo & Higham, 2006; Iannazzo, 2006).

We implemented the following optimizations to the coupled Newton iteration method:

- *Warm Start*: The coupled Newton iteration to compute $G^{-1/p}$ starts with $X = I, M = G$ and maintains the invariant $M = X^p G$ while driving $M \to I$, resulting in $X \to G^{-1/p}$. We need to find the $p$-th root of a sequence $G_t$, so we instead set $X = G_t^{-1/p}, M = X^p G_{t+1}$; since the difference between $G_t$ and $G_{t+1}$ is small, this ensures that $M$ is already close to $I$. In our experiments warmstart improves convergence (by upto 4x fewer steps).
- *Projecting top singular values*: In practice our $G_t$ matrices have large condition numbers, which sometimes leads to inaccurate results. As a rule of thumb, computing $G^{-1/p}$ leads to a loss of $\log_2(\frac{1}{p}\kappa(G))$ bits of precision (Overton, 2001), where $\kappa(G)$ is the condition number of the $G$. However we also find that usually there is a sharp falloff within the first few singular values, so in order to reduce the condition number, we project away the largest singular values, since these are the least important after taking inverses. We find the top-$k$ singular values $\lambda_1, \ldots, \lambda_k$ and their associated singular vectors using a standard iterative method, and replace each with $\lambda_{k+1}$. This produces a better approximation than adding $\epsilon I$ to each $G_t$: the latter can wash out the smallest (and most crucial) singular values, see Fig. 7 where the smallest singular value for a layer can be as small as $10^{-10}$ to $10^{-6}$ during the course of optimization.
- *Dynamic tuning of projection*: We dynamically tune the number of singular values we need to project in the previous step, by computing the condition number $\kappa(G_t)$ and using it to estimate the smallest singular value of $G_{t+1}$ as $\lambda_{\max}(G_{t+1})/\kappa(G_t)$. We then keep projecting out singular values of $G_{t+1}$ until we get an acceptable condition number.
- *Correcting for projection*: If $G_t = \sum_i \lambda_i \mathbf{v}_i \mathbf{v}_i^\mathsf{T}$, then $G_t^{-1/p} = \sum_i \lambda_i^{-1/p} \mathbf{v}_i \mathbf{v}_i^\mathsf{T}$. Projection above means replacing $\lambda_1, \ldots \lambda_k$ by $\lambda_{k+1}$, but since we have already computed the corresponding $\mathbf{v}_1, \ldots \mathbf{v}_k$, we correct the approximate $p$-th root by adding $\sum_{i=1}^{k} (\lambda_i^{-1/p} - \lambda_{k+1}^{-1/p}) \mathbf{v}_i \mathbf{v}_i^\mathsf{T}$. This is a small effect, but adding it is a straightforward modification (details deferred to Appendix E).

**Algorithm I** A coupled Newton iteration procedure for computing inverse $p$-th roots of a PSD matrix, with warm start and singular value projection

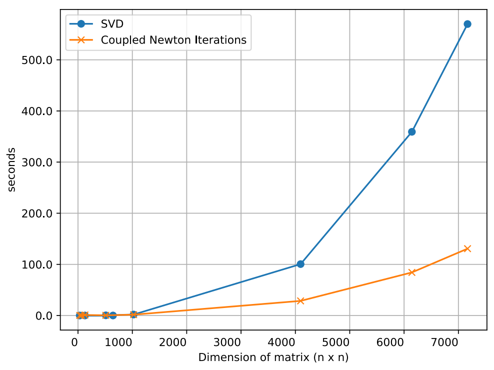
```
procedure MaxSV($\mathbf{G}$)
    Parameters: $\epsilon > 0$, $n_{\text{step}}$
    $\mathbf{v} \in \mathbb{R}^n$, where $\mathbf{G} \in \mathbb{R}^{n \times n}$
    $i = 0$, error $= \infty$, $\lambda = 0$
    while $i < n_{\text{step}}$ and error $> \epsilon$ do
        $\hat{\mathbf{v}} = \mathbf{v}/\|\mathbf{v}\|$
        $\mathbf{v} = \mathbf{G}\hat{\mathbf{v}}$
        $\lambda_{\text{old}} = \lambda; \lambda = \hat{\mathbf{v}}^{\mathsf{T}}\mathbf{v}$
        error $= |\lambda - \lambda_{\text{old}}|; i = i + 1$
    return $\lambda, \mathbf{v}/\|\mathbf{v}\|$

procedure Project($\mathbf{G}$, $\kappa$ (optional), $\kappa_d$ (optional), $n_{\text{proj}}$ (optional))
    $i = 0$
    $\Delta = \mathbf{0}$
    $\lambda, \mathbf{v} =$ MaxSV($\mathbf{G}$)
    $\lambda_{\max} = \lambda$
    while $\lambda > \frac{\kappa_d}{\kappa}\lambda_{\max}$ or $i < n_{\text{proj}}$ do
        $\mathbf{G} = \mathbf{G} - \lambda\mathbf{v}\mathbf{v}^{\mathsf{T}}$
        $\Delta = \Delta + \mathbf{v}\mathbf{v}^{\mathsf{T}}$
        $\lambda, \mathbf{v} =$ MaxSV($\mathbf{G}$)
        $i = i + 1$
    return $\mathbf{G} + \lambda\Delta$

procedure CoupledIteration($\mathbf{G}$, $p \in \mathbb{N}$, $\mathbf{X}$ (optional), $\kappa$ (optional))
    Parameters: $\epsilon > 0$, $\kappa_d$, $n_{\text{proj}}$
    Outputs: $\mathbf{G}^{-1/p}$
    $\mathbf{G} =$ Project($\mathbf{G}, \kappa, \kappa_d, n_{\text{proj}}$)
    $\alpha = -\frac{1}{p}$
    if $\mathbf{X}$ is provided then
        $\mathbf{M} = \mathbf{X}^p\mathbf{G}$
    else
        $z = \frac{1+p}{2\|\mathbf{G}\|_F}$
        $\mathbf{X} = \frac{1}{z^\alpha}\mathbf{I}$
        $\mathbf{M} = z\mathbf{G}$
    while $\|\mathbf{M} - \mathbf{I}\|_\infty > \epsilon$ do
        $\mathbf{M}_1 = (1 - \alpha)\mathbf{I} + \alpha\mathbf{M}$
        $\mathbf{X} = \mathbf{X}\mathbf{M}_1$
        $\mathbf{M} = \mathbf{M}_1^p\mathbf{M}$
    return $\mathbf{X}$
```

## F Implementation Details of Shampoo

Our implementation of the Shampoo algorithm for fully-connected layers is described in Algorithm II. The algorithm can use heavy-ball momentum for its updates, as well an exponential moving average over the preconditioners, like Adam. The configuration parameter $\tau_1$ denotes the number of steps between subsequent fetches of the latest available preconditioner by the accelerator. $\tau_1$ must be set sufficiently high so that there is enough time for the CPU to complete the computation of the preconditioner asynchronously and pipeline it efficiently, but otherwise its setting does not have a significant effect on convergence. The configuration parameter $\tau_2$ (default value = 1) determines the frequency of gathering gradient statistics - we update $L_t, R_t$ every $\tau_2$ steps only for efficiency.

### F.1 Computation cost of Shampoo

We capture the computational and memory complexity under various schemes described in Section 3.1 of handling large layers in Table 2.



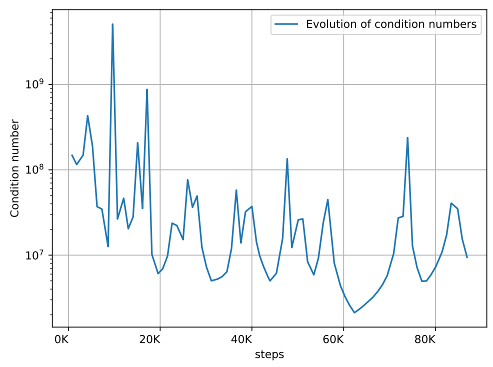


Figure 6: Benchmarks on computing inverse-pth root for statistics of varying dimensions (left), and the condition numbers for $L_t$ of a layer in the transformer model over time (right). We find that the coupled Newton iteration method can effectively utilize the CPUs and give large walltime improvements compared to SVD (that relies on bidiagonal divide-and-conquer). These were measured without warmstart which provides additional speedup of upto 4x by reducing the number of iterations to the solution.These were measured on Intel Skylake CPUs. Note that since $\sim \log_2(\frac{1}{p}\kappa(L_t))$ bits of precision are lost in computing $p$-th roots, 64-bit arithmetic becomes necessary.

**Algorithm II** Sketch of the Shampoo algorithm

```
1:  parameters: learning rate $\eta_t$, momentum: $\beta_1, \beta_2$
2:  for $t = 1, \ldots, T$ do
3:      Receive stochastic gradients $G_t$ for each layer
4:      if $t \% \tau_2 = 0$ then
5:          if $\beta_2 < 1$ then
6:              $L_t \leftarrow \beta_2\ L_{t-\tau_2} + (1-\beta_2)\ G_t G_t^\top$
7:              $R_t \leftarrow \beta_2\ R_{t-\tau_2} + (1-\beta_2)\ G_t^\top G_t$
8:          else
9:              $L_t \leftarrow L_{t-\tau_2} + G_t G_t^\top$
10:             $R_t \leftarrow R_{t-\tau_2} + G_t^\top G_t$
11:     $D_t \leftarrow D_{t-1} + G_t \bullet G_t$
12:     $M_t \leftarrow \beta_1\ M_{t-1} + (1-\beta_1)\ D_t^{\odot -1/2} \bullet G_t$
13:     if $t \% \tau_1 = 0$ then
14:         Gather preconditioners $L_{(t-\tau_1)}^{-1/4}, R_{(t-\tau_1)}^{-1/4}$ from CPUs
15:         Send $L_t, R_t$ to CPU host to compute $L_t^{-1/4}, R_t^{-1/4}$
16:     if $t > \tau_1$ then
17:         $P_t \leftarrow \beta_1 P_{t-1} + (1-\beta_1)\ L_t^{-1/4} G_t R_t^{-1/4}$
18:         $\eta_t \leftarrow \eta_0 \|M_t\|_F / \|P_t\|_F$
19:         $W_t = W_{t-1} - \eta_t P_t$
20:     else
21:         $\eta_t \leftarrow \eta_0$
22:         $W_t = W_{t-1} - \eta_t M_t$
```

## G Further Details on Experiments

*Layer wise learning rates.* As seen in Fig. 7 the step size scale for each layer is dependent on the operator norm of the preconditioners (inverse-pth root of the smallest singular value of the statistics

| Type | Computation | Memory |
|---|---|---|
| All preconditioner $W_t$: $[n, m]$ | $O(n^2 m + m^2 n)$ | $O(n^2 + m^2)$ |
| Left only preconditioner for $W_t$: $[n, m]$ | $O(n^2 m)$ | $O(n^2)$ |
| Preconditioner: block size $b$ | $O(mnb)$ | $O(mn)$ |

Table 2: Computational and memory complexity of variants of Shampoo.

matrix) has large spread in its range which results in optimization instabilities in practice. Moreover, as statistics as well as preconditioner computation are amortized across many steps the norm does not grow at every step. Hence, we rely on a learning rate schedule based on the update directions of a well tuned first order optimizer (in our experiments we use diagonal AdaGrad for Transformers in machine translation, as well as Criteo, layer-wise scaling heuristic proposed in LARS/LAMB optimizer, where each layer's learning rate is set to be $\|W_t\|_F / \|G_t\|_F$ for BERT and ResNet training. For example, when used with diagonal AdaGrad: Shampoo is used to determine the direction of the update, and AdaGrad to determine its magnitude.

This procedure termed Grafting in (Agarwal et al., 2020) allows us to bootstrap a reasonable learning rate schedule for a specific problem that is well tuned, and study the effect of preconditioned gradient directions in isolation. The weight matrix $W_t$ is updated as $W_t = W_{t-1} - A_t \hat{\mathbf{S}}_\mathbf{t}$, where:

$$D_t = \sum_{s=1}^{t} G_s \bullet G_s; \quad A_t = \eta_0 \left\| D_t^{\odot -1/2} \bullet G_t \right\|_F \qquad \text{(Adagrad magnitude)}$$

$$\hat{\mathbf{S}}_\mathbf{t} = \frac{L_t^{-1/4} G_t R_t^{-1/4}}{\left\| L_t^{-1/4} G_t R_t^{-1/4} \right\|_F} \qquad \text{(Shampoo direction).}$$

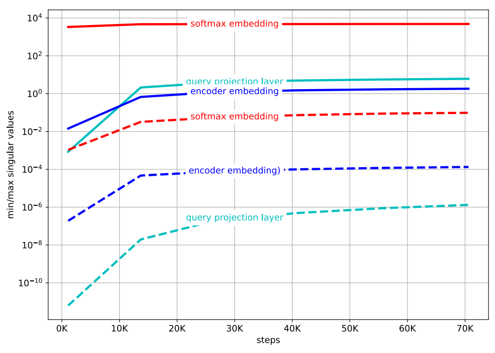


Figure 7: Minimum (dashed) and maximum (solid) singular values for statistics matrices of the embedding, softmax and intermediate attention query projection layers.

## G.1 Transformer model on WMT'14 en→fr

For all optimizers, we make use of a warmup schedule where the learning rate is increased from 0.0 to $\eta$ over 40k steps. For the smaller transformer experiments, we use a quadratic warmup, and for the larger transformer experiments we use a linear warmup. We found that quadratic warmup improves all optimizers equally and provides a better log-perplexity. For the Adam optimizer experiments, we use a learning rate decay schedule of the form $\eta_t = \eta \sqrt{d/t}$, following the suggestion of Vaswani et al. (2017). For the smaller Transformer experiments, we tuned the hyperparameters for each algorithm over 100 trials. We took the best settings for the momentum and second-moment parameters, and tuned the learning rates until either the model became unstable, or did not increase performance. For Shampoo, we used a per layer learning rate derived from AdaGrad (see Appendix G for details), and found that for the exact same hyperparameter settings as AdaGrad, Shampoo provides a modest improvement in performance. Moreover, Shampoo allows for larger learning rates than AdaGrad does, as shown in Fig. 4a.

## G.2 Step time for BERT-Large

Our current implementation showed a 14% increase in step time for BERT-Large, nearly wiping out all the gains from reduced number of steps (16%). We note that due amount of resources it would require to tune BERT, we used Shampoo with exact same hyper-parameters as LAMB with grafting to understand the effect of preconditioner. Moreover, step time can be optimized considerably as the current implementation is not heavily optimized. For example, larger batch sizes help amortize the preconditioning overhead, and reduce overall wall time to reach the same accuracy. Furthermore,

| Experiment (TPU cores) | Optimizer | Batch | Optimizer Parameters | Warmup |
|---|---|---|---|---|
| Transformer (32) | Adam | 1536 | $\eta = 0.000225$, $\beta_1 = 0.9$, $\beta_2 = 0.98$ | 40k steps |
| | Adagrad | 1536 | $\eta = 0.125$, $\beta_1 = 0.95$ | 40k steps |
| | Shampoo | 1536 | $\eta = 0.225$, $\beta_1 = 0.95$, $\kappa = 500$ $\tau_1 = 1000$, $\tau_2 = 1$ | 40k steps |
| Transformer-Big (32) | Adam | 384 | $\eta = 0.000154$, $\beta_1 = 0.9$, $\beta_2 = 0.999$ | 40k steps |
| | Adagrad | 384 | $\eta = 0.03$, $\beta_1 = 0.9$ | 40k steps |
| | Shampoo | 384 | $\eta = 0.06$, $\beta_1 = 0.9$, $\kappa = 500$ $\tau_1 = 1000$, $\tau_2 = 1$ | 40k steps |
| Transformer-Big (32) | Adagrad | 1536 | $\eta = 0.06$, $\beta_1 = 0.9$ | 40k steps |
| | Shampoo | 1536 | $\eta = 0.08$, $\beta_1 = 0.9$, $\kappa = 500$ $\tau_1 = 1000$, $\tau_2 = 1$ | 40k steps |
| Bert-Large (256) | LAMB | 16384 | $\eta = 0.0060$ $\beta_1 = 0.9$, $\beta_2 = 0.999$ | 6.4k steps |
| | Shampoo | 16384 | $\eta = 0.0060$ $\beta_1 = 0.9$, $\beta_2 = 0.999$, $\lambda_2 = 10^{-2}$, $\tau_1 = 400$, $\tau_2 = 10$ Block size: 1024 | 6.4k steps |
| DLRM (32) | SGD | 65536 | $\eta = 0.1$, poly decay(p=2) at 38k steps | 2k steps |
| | Shampoo | 65536 | $\eta = 0.1$ poly decay(p=2) at 38k steps $\beta_1 = 0.9$, $\tau_1 = 999$, $\tau_2 = 10$ | 2k steps |
| | (w/ embd) | 65536 | $\eta_{\text{embd}} = 0.31$ | |

Table 3: Hyperparameter setup used in our experiments.

in our current implementation, all TPU cores compute all the preconditioning statistics and the preconditioned gradients, which involves over a hundred $1024 \times 1024$ matrix multiplications. This repeated work can be avoided by cross-replica sharding of weight update (Xu et al., 2020), which distributes this computation across cores, and should save at least half the step time overhead.

### G.3 CIFAR-10

We train a ResNet-50 model on CIFAR-10 (Krizhevsky et al., 2009) with 2 cores of CloudTPU-v2 at batch size 2048. Our baseline achieves 93.45% accuracy at 300 epochs, where as Shampoo reaches the same accuracy in 143 epochs. We see an overall training time reduction of 42% (1428 seconds to 827 seconds). As it is a smaller problem, the time taken for preconditioner inverse computation for the largest preconditioning matrix is less than 1ms on the CPU. We use a total of 8 CPU cores to run these inverses.

### G.4 ImageNet

For SGD with Momentum, the learning rate is warmed up over the first 5 epochs from 0 to 1.6, followed by a 10x drops of the learning rate at 30, 60 and 80 epochs. For LARS, we use warmup learning rate over 20 epochs for 4K and 16K batch sizes, 25 epochs for 32K batch size with a polynomial decay (p=2) until end of training. For Shampoo we use the same layer-wise heuristics and hyperparameters as LARS with Grafting such that the direction is changed to the one computed by Shampoo. We make use weight decay with value: $\lambda_2 = 2\text{x}10^{-4}$ and label smoothing of $10^{-1}$.

### G.5 Detailed results for experiments

Approximate wall clock times for the various tasks are as follows:

| Task | Model | Baseline | Shampoo |
|---|---|---|---|
| Recommendations: Criteo-1Tb | DLRM | 13 min | 8.2 min |
| Translation: WMT-14 En-Fr | Transformer | $\approx$ 12 hrs | 6.5 hrs |
| Translation: WMT-14 En-Fr | Transfomer-Big | $\approx$ 47 hrs | 29.5 hrs |
| Language Modeling: Wikipedia+Books | BERT-Large | 228 mins | 219 mins |

### G.6 Breakdown of step-time in Fig. 2b

Each step of training consists of the following phases, whose times are shown in Fig. 2b.

- Forward Pass: Each core independently computes the predictions for each training example in its sub-batch.
- Gradient: The gradient is for the sub-batch is computed using the back-propagation algorithm.
- All reduction: The gradients for the sub-batches from all cores are averaged to compute the gradient for the minibatch. This is then sent back to each core.
- Preconditioner statistics: The preconditioner statistics for adaptive algorithms are updated, e.g. for AdaGrad, we set $H_i := H_i + g_i^2$ for all parameters, while for Shampoo, we set $L_i := L_i + GG^\mathsf{T}$ etc.
- Preconditioned gradient: The preconditioned gradient is computed - e.g. for AdaGrad, we compute $g_i/\sqrt{H_i}$, while for Shampoo, we compute $L^{-1/4}GR^{-1/4}$.
- Parameter updates: The parameters are updated using the preconditioned gradients. This step is the same for all algorithms: $W := W - \eta\tilde{G}$, where $\tilde{G}$ is the preconditioned gradient.

Note that the Shampoo computation of the preconditioners $L^{-1/4}, R^{-1/4}$ is pipelined on the host CPU, so does not show up in the step times.