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.

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

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 σ. Hidden layers $H_i = \sigma(H_{i-1}W_i)$ are stacked, with the output of the kth 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.

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 $H_i = \sigma(H_{i-1}W_i)$ 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, θ_{dist}, over candidate networks. It samples a set of decisions $(d_1, d_2, ...)$ 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 $R(d_1, d_2, ...)$ and conduct a policy gradient update using the REINFORCE algorithm [68] as follows:

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

where $\overline{R}$ denotes the moving average value of the reward and α_0 is the learning rate for the reinforcement learning algorithm.

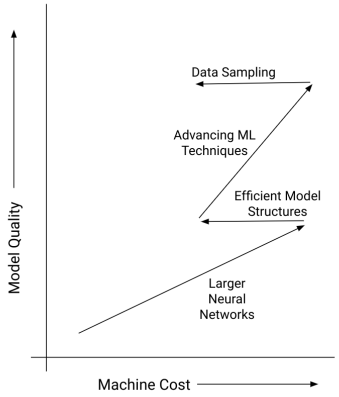

Figure 1: "Switch-backs" of incremental costly quality-improving techniques and efficiency methods. *(Illustration not to any scale.)*

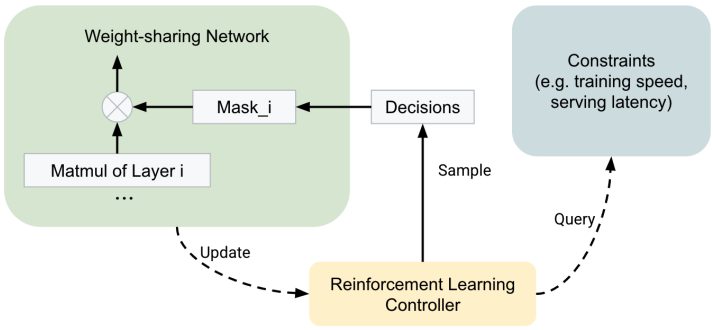

Figure 2: Weight-sharing based NAS with cost constraints.

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., $R = R_{\text{accuracy}} + \gamma \cdot |\text{cost/target} - 1|$, 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 sub-sampled 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_{shared} are weights shared between the two losses, W_{logistic} are for the logistic loss output, and W_{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_{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

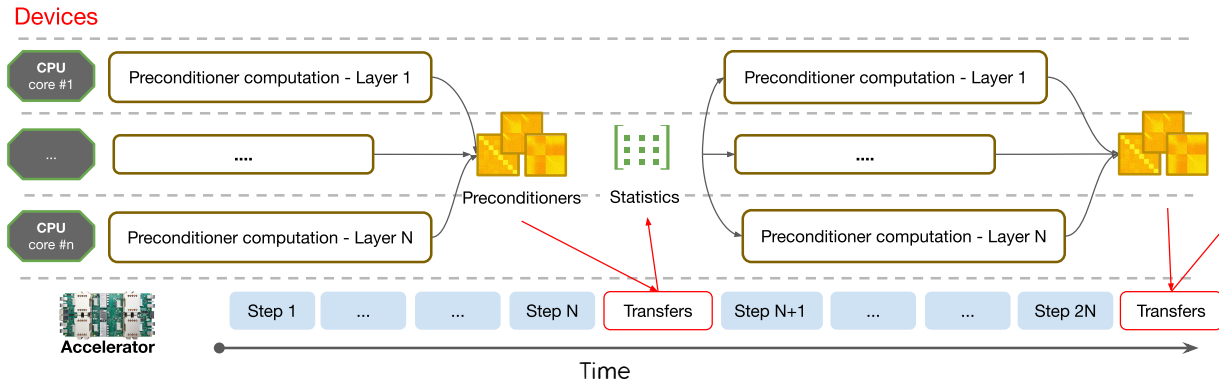

Figure 3: Distributed Shampoo [5]: inverse-p^{th} root computations in double precision runs every N steps and asynchronously pipelined on all CPU cores attached to the TPU accelerators.

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 ϵ 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-pth 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: $e_i = \alpha_2\left(e_0 \odot U_i V_i e_{i-1}\right) + e_{i-1}$, where $e_i, e_{i-1} \in \mathbb{R}^m$ represent the output and input of the i^{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; α_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

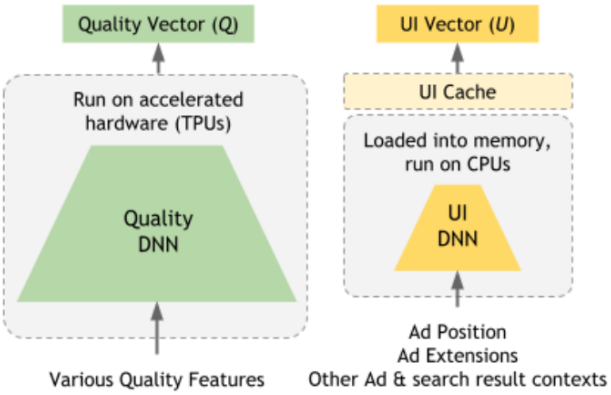

Figure 4: Model factorization into separable Quality and UI models with estimated CTR $:= \tau(Q \cdot U)$

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:

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

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 τ 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

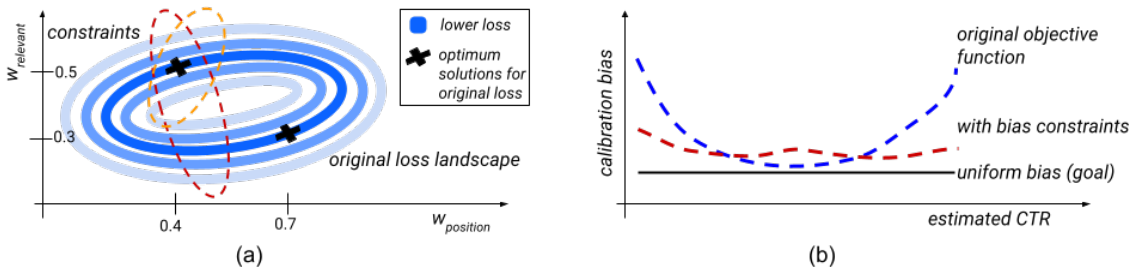

Figure 5: (a) Loss landscape for a model with non-identifiability across two weights and how bias constraints help find the right solution: we add additional criteria (red and orange curves) such that we choose the correct solution at optimum loss (dark blue curve). (b) Calibration bias across buckets 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 λ_k with a constraint of zero average bias. For a constant α_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 α_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*.