Adaptive Online Hyper-Parameters Tuning for Ad Event-Prediction Models

Michal Aharon
Yahoo Research, Haifa, Israel
michala@yahoo-inc.com

Amit Kagian
Yahoo Research, Haifa, Israel
akagian@yahoo-inc.com

Oren Somekh
Yahoo Research, Haifa, Israel
orens@yahoo-inc.com

ABSTRACT

Yahoo's native advertising (also known as Gemini native) is one of its fastest growing businesses, reaching a run-rate of several hundred Millions USD in the past year. Driving the Gemini native models that are used to predict both, click probability (pCTR) and conversion probability (pCONV), is OFFSET - a feature enhanced collaborative-filtering (CF) based event prediction algorithm. OFFSET is a one-pass algorithm that updates its model for every new batch of logged data using a stochastic gradient descent (SGD) based approach. As most learning algorithms, OFFSET includes several hyper-parameters that can be tuned to provide best performance for a given system conditions. Since the marketplace environment is very dynamic and influenced by seasonality and other temporal factors, having a fixed single set of hyper-parameters (or configuration) for the learning algorithm is sub-optimal.

In this work we present an online hyper-parameters tuning algorithm, which takes advantage of the system parallel map-reduce based architecture, and strives to adapt the hyper-parameters set to provide the best performance at a specific time interval. Online evaluation via bucket testing of the tuning algorithm showed a significant 4.3% revenue lift overall traffic, and a staggering 8.3% lift over Yahoo HomePage section traffic. Since then, the tuning algorithm was pushed into production, tuning both click- and conversion-prediction models, and is generating a hefty estimated revenue lift of 5% yearly for Yahoo Gemini native.

The proposed tuning mechanism can be easily generalized to fit any learning algorithm that continuously learns on incoming streaming data, in order to adapt its hyper-parameters to temporal changes.

Keywords

Hyper-parameters tuning, learning, map-reduce, native ads, ad ranking, ad click-prediction

WWW 2017 Companion, April 3–7, 2017, Perth, Australia.
ACM 978-1-4503-4913-0/17/04.
http://dx.doi.org/10.1145/3041021.3054184

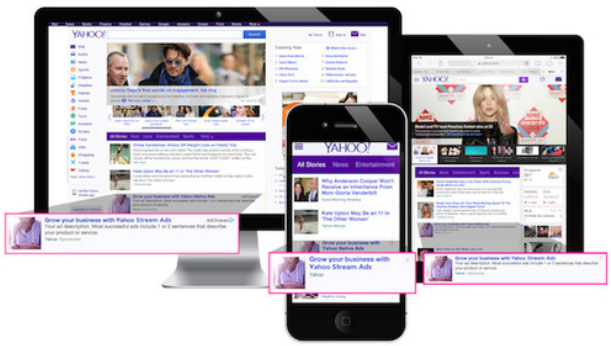

Figure 1: Gemini native ads on different devices.

1. INTRODUCTION

Yahoo's native ad marketplace (also known as *Gemini native*[1]) serves users with ads that are rendered to resemble the surrounding native content (see Figure 1 for examples of Gemini native ads on different devices). In contrast to the search-ads marketplace, users intent during page visit is unknown in general. Launched three years ago and recently reaching a yearly run-rate of several hundred Millions USD, Gemini native is one of Yahoo's fastest growing businesses. With more than a Billion impressions daily, and an inventory of a few hundred thousand active ads, this marketplace performs real-time *generalized second price* (GSP) auctions that take into account ad targeting, budget considerations, and frequency and recency rules, with SLA of less than 80 ms more than 99% of the time. In order to rank the native ads for an incoming user and her specific context, a score (or expected revenue) is calculated by multiplying the advertiser's bid and the predicted click probability (pCTR) for each ad. In addition to the cost-per-click (CPC) price type, Gemini native also supports the oCPx price type. According to this price-type, advertisers declare a target cost-per-action price (tCPA) for a conversion event (such as purchasing, filling a form, or installing an app) that occurs after a click. For this price type the system predicts the probability of a conversion given a click (pCONV) and multiplies it by the tCPA to get the effective oCPx bid, which is used during auction.

The pCTR and pCONV are calculated using models that are periodically updated by OFFSET - a feature enhanced collaborative-filtering (CF) based event-prediction algorithm

[1]https://adspecs.yahoo.com/adformats/native/

[1]. OFFSET is a one-pass algorithm that updates its latent factor model for every new mini-batch of logged data using a stochastic gradient descent (SGD) based learning approach. OFFSET is implemented on the grid using *map-reduce* architecture [4], where every new mini-batch of logged data is preprocessed and parsed by many *mappers* and the ongoing update of a model is conducted as a centralized process on a single *reducer*.

As many other learning algorithms, OFFSET includes several hyper-parameters (or configuration) that can be tuned to provide best performance for given system conditions. The architecture of OFFSET makes it possible to do a *parallel grid-search* to find an "almost" optimal set of hyper-parameters and its resulting model, for optimizing system performance. We note that it usually takes a few days to train the system using a few weeks worth of logged data in order to get a "mature" model which can be pushed to production and start serving ads to users.

The Gemini native marketplace is a dynamic environment that is influenced by seasonality, and other temporal factors such as market trends, churning and appearing of large advertisers, and worldwide events. Therefore, having a single fixed hyper-parameters set (or configuration) is definitely sub-optimal. Even without considering environmental changes, a fixed set of hyper-parameters may not fit a model throughout its whole "life-cycle"; a mature model that has already been trained over months of data may require different set of hyper-parameters than the set found by the initial parallel grid-search described earlier. On the other hand, performing a parallel grid-search from time to time, using "fresh" logged data, is a time consuming task as mentioned earlier.

In this work we take advantage of the system architecture that facilitates training of many models in parallel, and present an online hyper-parameters tuning framework for the OFFSET algorithm[2]. In particular, the propose tuning algorithm trains many models with different configurations in parallel, and identifies the best configuration and its corresponding model according to some performance metric. While this "best" model is used to serve Gemini native users, the tuning algorithm uses the "best" configuration to generate P new configurations in its "vicinity". Then, it continues training P copies of the best model, each with one of the P new configurations, using the new mini-batch of logged data, and so on and so forth. In this manner, the tuning algorithm is continuously experimenting with alternative variations of the currently best performing hyper-parameters configuration. Namely, the tuning algorithm strives to track the best hyper-parameters set and its corresponding model that provide the best performance at each time interval. In addition to the "error-free" procedure briefly described here, the tuning algorithm is able to handle also extreme scenarios where few or even all models were diverged and a recovery mechanism must be applied to ensure correct operation. We note that for simplicity matters we focus here on the click-prediction version of OFFSET. However, in practice the tuning algorithm is optimizing both, click- and conversion-prediction models, in production using slightly different setups.

Our tuning algorithm was tested and evaluated serving real users in online buckets, showing a significant 4.3% revenue lift over all traffic, and a staggering 8.3% lift specifically for the Yahoo Home-Page section traffic. Since then, the tuning algorithm was pushed into production, tuning both click- and conversion-prediction models, and is generating a hefty estimated overall lift of $\sim 5\%$ in revenue yearly for the Gemini native marketplace.

The rest of this paper is organized as follows. In Section 2 we provide background. Section 3 describes common practices and related work. Our approach is detailed in Section 4. Offline and online evaluation setups and their results are presented in Section 5. We conclude and discuss future work in Section 6.

2. BACKGROUND

2.1 Gemini Native

Gemini native is one of Yahoo's fastest growing businesses, reaching a few hundred Millions USD run-rate in revenue in early 2015Q2. Gemini native serves a daily average of more than Billion impressions across North America, Europe and Asia, with SLA of less than 80 ms for more than 99% of the queries, and a native ad inventory of few hundred thousands active ads on average. The online serving system is comprised of a massive Vespa[3] deployment, augmented by ads, budget and model training pipelines. The Vespa index is updated continuously with ad and budget changes, and periodically (e.g., every 15 minutes) with model updates. The Gemini native marketplace serves several ad price-types including CPC (cost-per-click), oCPx (conversion), CPM (cost-per-thousand impression), and also includes RTB (real-time biding) in its auctions.

2.2 OffSet - Ad Click-Prediction Algorithm

The algorithm driving Gemini native models is OFFSET (One-pass Factorization of Feature Sets): a feature enhanced collaborative-filtering (CF) based ad click-prediction algorithm [1]. The predicted click-probability or click-through-rate (pCTR) of a given user u and ad a according to OFFSET is given by

$$pCTR(u,a) = \frac{1}{1+\exp^{-\left(b+\nu_u^T \nu_a\right)}} \in [0,1] \ ,$$

where $\nu_u, \nu_a \in \mathbb{R}^{\mathrm{D}}$ denote the user and ad latent factor vectors, respectively, and $b \in \mathbb{R}$ denotes the model bias. The product $\nu_u^T \nu_a$ denotes the tenancy score of user u towards ad a, where higher score translates into higher predicted click-probability. Note that $\Theta = \{\nu_u, \nu_a, b\}$ are the model parameters which are learned from the logged data as will be explained in the sequel.

Both ad and user vectors are constructed using their features, which enable dealing with data sparsity issues. For ads, we use a simple summation between the vectors of the unique creative id, campaign id, and advertiser id (currently 3 feature vectors, all of dimension D). The combination between the different user feature vectors is more complex, allowing non-linear pair-wise dependencies between features.

The user vectors are constructed using their K features latent vectors $v_k \in \mathbb{R}^{\mathrm{d}}$ (e.g., age, gender, geo, etc.). In particular, o entries are devoted for each pair of user features, and s entries are devoted for each feature alone. The dimension of a single feature vector is therefore $d = (K-1) \cdot o + s$,

[2]We note that the proposed tuning algorithm is generic and can be used for other learning algorithms as well.

[3]Vespa is Yahoo's elastic search engine solution.

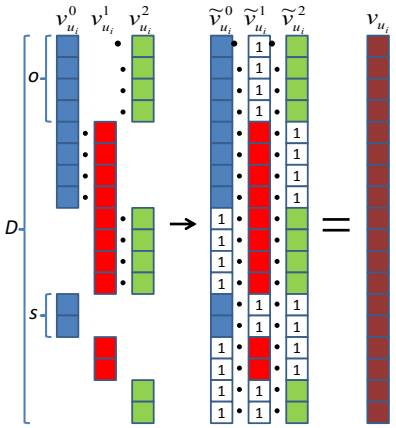

Figure 2: Example of a user latent factor vector construction for $K = 3,\ o = 4,\ s = 2$.

where the dimension of the combined user vector is $D = \binom{K}{2} \cdot o + K \cdot s$ (the ad's side features have the same dimension D to allow an inner product with the user vector). An illustration of this construction is given in Figure 2. In addition to having no user cold-start issues, using this presentation of users the resulting model includes only K user feature latent factor vectors instead of hundreds of millions unique user latent factor vectors of the standard CF presentation.

To learn the model parameters Θ, Offset minimizes the logistic loss (LogLoss) of the training data set $\mathcal{T}$ (i.e., past impressions and clicks) using one-pass *stochastic gradient descent* (SGD) based algorithm. The cost function is as follows

$$\underset{\Theta}{\operatorname{argmin}} \sum_{(u,a,y)\in\mathcal{T}} \mathcal{L}(u,a,y) \ ,$$

where

$$\mathcal{L}(u,a,y) = -(1-y)\log\left(1 - pCTR(u,a)\right) \\ - y \log pCTR(u,a) + \lambda \sum_{\theta\in\Theta} \theta^2 \ ,$$

is the loss function, $y \in \{0,1\}$ is the impression click indicator involving user u and ad a, and λ is the $L2$ regularization parameter. For each training impression (u,a,y) Offset updates its relevant model parameters using SGD step

$$\theta \leftarrow \theta + \eta(\theta) \nabla_\theta \mathcal{L}(u,a,y) \ ,$$

where $\nabla_\theta \mathcal{L}(u,a,y)$ is the divergence of the loss function w.r.t θ. In addition, the parameter dependent step size is given by

$$\eta(\theta) = \eta_0 \frac{1}{\alpha + \left(\sum_{(u,a,y)\in\mathcal{T}'} |\nabla \mathcal{L}(u,a,y)|\right)^\beta} \ ,$$

where η_0 is the SGD primary step-size, $\alpha,\ \beta \in \mathbb{R}^+$ are the parameters of the adaptive gradient (AdaGrad) algorithm, inspired by [5], and $\mathcal{T}'$ is the set of training impressions seen so far. To summarize this part, here is a partial list of hyper-parameters that can be tuned to optimize Offset performance: (a) η_0 - SGD primary step-size; (b) $\alpha,\ \beta$ - AadaGrad parameters; and (c) λ - regularization parameter.

As mentioned earlier, Offset uses an online approach where it continuously updates its model parameters with each mini-batch of new training impressions (e.g., every 15 minutes for click-prediction model). A more elaborate description, including details on AdaGrad use [5], multi-value features, and regularization can be found in [1].

2.3 System Architecture Overview

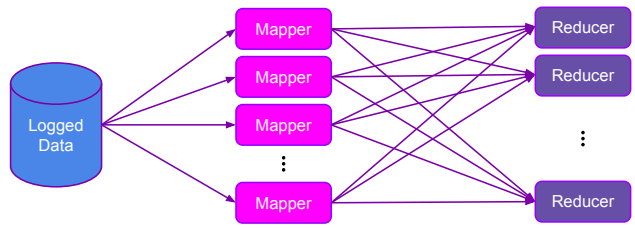

Figure 3: System architecture in a map-reduce paradigm.

The Offset training process is sequential, running on a single reducer. Hence, it is imperative that the learning of a single mini-batch takes less than the time covered by the mini-batch (e.g., 15 minutes). Therefore, the input data should be preprocessed quickly and be organized for Offset to consume.

For this purpose, we design a map-reduce based mechanism to execute Offset training (see Figure. 3). The input data is processed by multiple mappers in parallel. Each mapper employs sub-sampling (includes 1 of n impressions, and all clicks) and extracts for each sampled event only the relevant information required for training. The *map key* is composed of a serialization of the hyper-parameters set (allowing models with different hyper-parameters sets to be trained in parallel). Offset is then executed on a single reducer (per hyper-parameters set) and trains over all preprocessed entries.

Using multiple reducers for different hyper-parameters sets enables a seamless parameter selection process with little additional cost. Such a design allows training hundreds of models in parallel, each with a unique set of training hyper-parameters set. We can take advantage of this architecture and perform an initial training with parallel grid-search of "good" hyper-parameters set (or configuration). After training over a few weeks worth of logged data, which may take a few days to execute, the "best" hyper-parameters set and its resulting model, according to some predefined performance metric, are selected to serve online traffic.

A more sophisticated way of exploiting this parallel architecture for continuous online tuning of Offset hyper-parameters set, is the subject of this work and will be presented in the sequel.

2.4 Serving System Overview

The Gemini native serving platform serves ads across all Yahoo's properties and third-party supply (i.e., syndication). The serving deployment spans on large number of Vespa containers, including several hundred hosts across several colos in the US, Europe and Asia. It abides by very strict latency and SLAs requirements, and is designed to withstand entire data center failures.

The serving charter is to select which ads to show a user in a certain context while maximizing revenue, maintaining a good user experience and a "healthy" marketplace (e.g., not starving out small advertisers). Therefore, for each impression the system conducts a *generalized second price* (GSP) auction [6], and uses the click- and conversion-prediction models, campaign budgets and bids, and various rules[4], to perform ad-ranking over the active ads inventory.

Gemini native serving supports live updates of budget, ad configuration, and prediction model updates. In addition, Gemini serving supports a very flexible bucketing system allowing A/B testing and traffic ramp-up of new models, and configurations without the need to deploy any code. In particular the platform splits traffic between the production (93%) and science buckets (7%) environments, each backed by separate Vespa clusters, with potentially different models, code, and configuration settings.

3. COMMON PRACTICE AND RELATED WORK

Most learning algorithms include hyper-parameters that may be tuned to improve on performance according to a predefined evaluation metric. The problem of hyper-parameter optimization (or tuning) has been studied for decades in various disciplines. For instance, in the field of machine learning, the use of Gaussian processes [10], random forests [8], and reinforcement learning [8], have been proposed. As opposed to these advance methods, simple heuristics such as *grid* and *random search* [2], are widely used by commercial learning systems. In particular, our initial parallel grid search mentioned in Section 2.3 is a form of grid search.

All the methods mentioned above consider a one time initial process of hyper-parameters optimization. A work dealing with a continuous online tuning algorithm is presented in [3]. As we shell elaborate in the sequel, our approach is quite different than the one considered in [3]. Moreover, while [3] presents an experimental algorithm which was tested using public datasets specially engineered to emulate an online setting, our algorithm is tested and evaluated while serving millions of users of Yahoo Gemini native traffic.

4. OUR TUNING APPROACH

In this section we provide definitions and notations needed for presenting our algorithm. After an in-depth presentation of our approach, we consider ways to handle extreme scenarios involving models divergence.

4.1 Definitions and Notations

- $\Theta = \{\nu_{uf_1}, \ldots, \nu_{uf_K}, \nu_{a_1}, \ldots, \nu_{a_\ell}, b\}$ model parameters (K user features latent vectors, ℓ ads latent vectors[5], and model bias)
- $\Phi = \{\phi_1, \ldots, \phi_n\}$ model hyper-parameters set (e.g., regularization coefficient, SGD primary step size, and AdaGrad parameters). For simplicity matters we assume $\phi_i \in \mathbb{R}$
- $\Psi = \{\psi_1, \ldots, \psi_n\}$ model hyper-parameters constraints $\psi_i = [a_i, b_i]\ ;\ a_i,\ b_i \in \mathbb{R}\ ;\ \mathrm{a_i} < \mathrm{b_i}$
- $\mathcal{T} = \{(u, a, y)\}$ logged data which includes triplets of user information, ad information, and event label
- $\mathcal{M} : \Theta, \mathcal{T} \rightarrow \mathbb{R}$ performance metric such as *stratified AUC* and *LogLoss*
- $\mathcal{F} : \Phi \rightarrow \Phi^P$ model hyper-parameter sets generation function which gets a certain model hyper-parameters set Φ, model hyper-parameters constraints Ψ, and a positive integer P, and generates P model hyper-parameter sets $\Phi, \Phi_1, \ldots, \Phi_P$
- $L \geq 1$ hyper-parameters tuning cycle in number of model train periods

4.2 Hyper-Parameters Tuning Algorithm

4.2.1 General description

The basic idea, is to continuously train, in parallel, multiple versions of the learning model with P variations of the hyper-parameters set. In the end of each tuning cycle (e.g., an hour or 4 model training periods), each model version is evaluated and the current best hyper-parameters set and resulting model are identified. The training during the next cycle will continue from the best performing model with new generated variations of its hyper-parameters. By doing so, we continuously experiment with variations of the tuned hyper-parameters set in order to make them adaptive to temporal changes. Figure 4 depicts the process of virtually duplicating the best performing model and resuming its training with multiple hyper-parameter sets (configurations). For simplicity, the diagram shows a fixed sets of configurations while our implementation generates new configuration variations at each tuning cycle. A formal description of the hyper-parameters tuning algorithm is given below and in Algorithm 1.

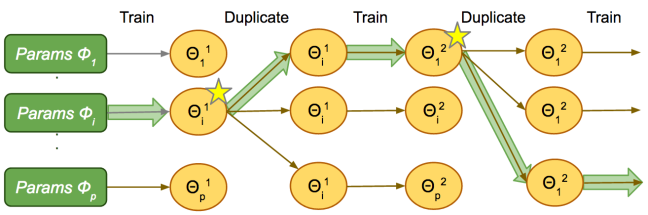

Figure 4: Tuning algorithm diagram. At each tuning cycle, the best performing model is identified (denoted by a star), duplicated into P copies and then continues its training with P variations of hyper-parameters sets.

We assume an initial parameter tuning procedure via parallel grid search (see Section 2.3), conducted offline over several weeks worth of logged data, resulting with an initial mature model Θ^0 and a corresponding hyper-parameters set Φ^0. The tuning cycle starts with the model hyper-parameters sets generation function $\mathcal{F}$ getting Φ^0 and Ψ and generating P hyper-parameters sets (the original set Φ^0, and $P-1$ new sets). Now, the model Θ^0 is virtually duplicated into P copies while each copy is trained for L train periods with its corresponding hyper-parameters set over the logged data

[4]Such as frequency and recency of displaying a certain ad to a specific user.

[5]Ad vectors are actually the summation of the creative, campaign, and advertiser latent vectors (see Section 2.2).

$\mathcal{T}$. After the tuning cycle is due, the model with the best performance metric is selected along with its corresponding hyper-parameters set (Θ_m^1, Φ_m^1). The selected pair is stored and used for the next tuning cycle and so on and so forth. Intuitively, a shorter tuning cycle L enable faster adaption of the hyper-parameters set, and a longer cycle provides more accurate evaluation of each hyper-parameters set.

Algorithm 1 Hyper-parameters tuning algorithm

Input: (Θ^0, Φ^0), Ψ, P, L
Output: $(\Theta^1, \Phi^1), (\Theta^2, \Phi^2), \ldots$ - pairs of best models and corresponding configurations sets for each tuning cycle

$t \leftarrow 0$
for ever **do**
  generate P hyper-parameters sets $\mathcal{F}(\Phi^t, \Psi) = \{\Phi_1^t, \Phi_2^t, \ldots, \Phi_P^t\}$
  duplicate Θ^t into P copies $\{\Theta_1^t, \Theta_2^t, \ldots, \Theta_P^t\}$
  $\Omega \leftarrow \mathbf{0}$, $\ell \leftarrow 1$
  for $\ell \leq L$ **do**
    accumulate data of train period $\mathcal{T}$
    train all P models $\{(\Theta_i^t, \Phi_i^t)\}$ over $\mathcal{T}$
    update performance metric vector $\Omega \leftarrow \Omega + \frac{1}{L}(\mathcal{M}(\Theta_1^t, \mathcal{T}), \mathcal{M}(\Theta_2^t, \mathcal{T}), \ldots, \mathcal{M}(\Theta_P^t, \mathcal{T}))$
    $\ell \leftarrow \ell + 1$
  end for
  $(\Theta^{t+1}, \Phi^{t+1}) \leftarrow (\Theta_i^t, \Phi_i^t)$ where $i = \text{argmin } [\Omega]_j$
  $t \leftarrow t + 1$
end for

It is noted, that here we provide an error-free procedure assuming all P models do not diverge. Section 4.3 describes how the tuning algorithm handles extreme scenarios. Also worth mentioning is that the procedure may be initiated using a randomly initialized model and some arbitrary hyper-parameters set. Such initiation instead of the one based on grid search may cause temporary performance degradation.

Our approach is different from that of [3] in several key points. While we use an incremental training, [3] adds the new data batch to the historical data and trains its models from scratch. As we shell show in the sequel, we also use a more targeted hyper-parameters sets generation function while [3] uses random search.

4.2.2 Performance metrics

Area-under ROC curve (AUC) The AUC specifies the probability that, given two random events (one positive and one negative, e.g., click and skip), their predicted pairwise ranking is correct [7].

Stratified AUC (sAUC) The weighted average (by number of positive event, e.g., number of clicks) of the AUC of each Yahoo section. This metric is used since different Yahoo sections has different prior click bias and therefore even using the section feature alone turns out as sufficient for achieving high AUC values.

Logistic loss (LogLoss)

$$\sum_{(u,a,y)\in\mathcal{T}} -y \log pCTR(u,a) - (1-y) \log (1 - pCTR(u,a)),$$

where $\mathcal{T}$ is a training set and $y \in \{0, 1\}$ is the positive event indicator (e.g., click or skip).

It is noted, that although all three metrics are available in the system, the LogLoss metric, which is the metric used by Offset to train its model, has provided the best results and was used during the online bucket testing and finally was pushed into production.

4.2.3 Hyper-parameter sets generation function

There are many heuristic ways we can generate hyper-parameters sets from a given set. We select a simple scale-up/scale-down approach according to which we set S scale factors (e.g., for $S = 3$: 0.9, 1.0 and 1.1) and use these to generate S new values for each hyper-parameter of the initial set. Then, we limit the new values to the predefined constraints Ψ in case they exceed the given bounds. Assuming we have M hyper-parameters for tuning, and S scale factors, the number of new hyper-parameters sets $\mathcal{F}$ generates equals S^M, which equals to 81 in case $S = 3$ and $M = 4$.

For practical reasons we don't wish the number of new set to exceed a predefined maximum number of sets P_m (say $P_m = 100$). In case $P > P_m$ we select the original set and additional $P_m - 1$ sets at random.

4.3 Handling Extreme Scenarios

To get the best performance of the tuning mechanism the constraints of the hyper-parameters should be rather loose. On the other hand, having a tuning mechanism that strive to get the best performance using loose constraints is risky since our SGD based model learning algorithms might diverge. One can think of this process to resemble a person walking along the edge of a cliff. To get the best view you want to get as close to the edge as possible. However, walking so close to the edge is dangerous since you may slip and fall. Therefore, we must detect when we start to "slip" (detect model divergence), make sure that we have a safety harness (use anchor configurations), and wear a parachute in case we fall (add a recovery mechanism).

Temporal changes and local minima.

As the system keeps learning continuously over time it may face some temporal changes in the environment/market that will lead the online hyper-parameters tuning to be "stuck" in a local minima. For example, in the ad marketplace, during the holiday season, there is an enormous daily addition of new ads. In such a scenario, identifying good new ads may be a critical factor for marketplace revenue. Thus, the hyper-parameters of the model are adapted to allow more rapid changes in the model itself (e.g. a larger step size). This puts more weight on quick learning of new ads rather than more accurate learning of familiar ads. However, as the holiday season abruptly ends, it is difficult for the hyper-parameters to move away from that area in the hyper-parameters search-space. The tuning algorithm is now "stuck" in a local minima. The hyper-parameters were able to adjust to a temporal change in the environment and they cannot find their way back once the marketplace is back to normal. Experimentation with online traffic during the holiday season of December-January 2015-2016 demonstrated this kind of behavior. We introduce the concept of *Anchor hyper-parameters set* in order to deal with scenarios of this sort.

Anchor hyper-parameters set.

The tuning system is all about generating new hyper-parameters sets in the vicinity of the last winning set, and training copies of the winning model with these new sets using the next logged data batch. This is a risky move that can bring more revenues but may cause all models to diverge, or alternatively, lead the hyper-parameters tuning into a local minima. To reduce this risk we use a small number of predefined hyper-parameters sets (e.g., $k = 16$), referred to as *Anchor sets*, that include parameters with moderate values (e.g., "small" SGD step sizes, and "large" regularization constants) and that were tested over long period of time during which their corresponding models showed no sign of divergence. Those Anchor sets $\{\widehat{\Phi}_1, \ldots, \widehat{\Phi}_k\}$ are included in the tuning process along with their corresponding models. So in practice after every tuning cycles we store the best model of that tuning cycle, along with the k models that are trained using the Anchor hyper-parameters sets. This mechanism provides safety anchors, preventing the model from "getting lost" in the hyper-parameters search-space.

Model divergence detection.

There are many heuristics to detect model divergence. The simplest yet effective way to detect a model divergence event is to monitor the magnitude of the model parameters by checking whether the amplitude of one of them surpasses a predefined threshold. Hence, we declare that a specific model Θ diverged if

$$\exists\ \theta \in \Theta,\ \text{such that } |\theta| > T_d\ .$$

Setting T_d is somewhat tricky since it is data and model dependent. It also presents a trade-off between false-alarm and missed-detection and requires a long calibration process via offline and online buckets experimentation.

In case a specific model diverges within a tuning cycle, that model is not updated in the end of the learning period and it will resume training with the next batch of logged data. It is worth mentioning that such a model is less updated than the other models which causes its performance metric to deteriorate and in turn reduces its likelihood to be the best model in the forthcoming cycles. Since in each tuning cycle we train P models in parallel having one diverged model is not critical. We declare that the whole system has failed only if all P models were diverged at once.

Recovery mechanism.

In case all models are diverged at the end of a tuning cycle, the system roles back and start from the latest cycle that ended correctly (i.e., at least one model didn't diverge) using "fresh" logged data. Note, that the system is able to do so since it stores the series of best models and corresponding hyper-parameters sets. If the next tuning cycle still ends with all models diverging, the system uses the previous correct cycle and resume learning from there and so on and so forth. The system is allowed to dive into the past up to a predefined number of cycles. In case all models still diverge after that, the system halts, and human intervention is required. In this extreme and rare case, the system has to be restarted and resume training from scratch or using some other reliable model and hyper-parameters set pair.

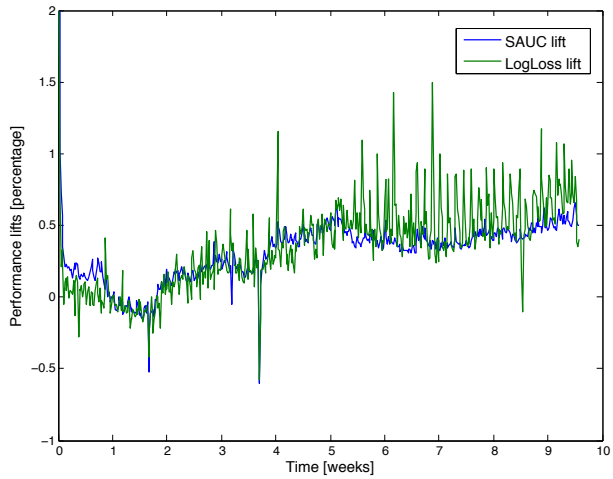

Figure 5: Hyper-parameters tuning mechanism offline performance metrics lifts over time, measured every 3 hours.

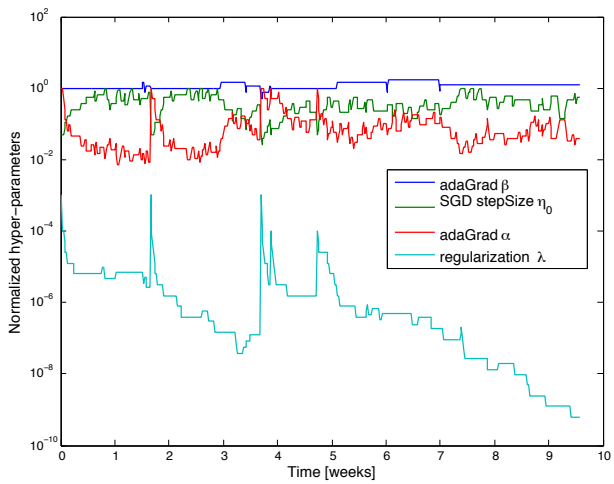

Figure 6: Normalized hyper-parameters values over time, measured every 3 hours.

5. EVALUATION

We have conducted both, offline and online evaluations, to asses and measure the benefits of the proposed tuning mechanism.

5.1 Offline Evaluation

5.1.1 Setup

To assess the potential benefits of the tuning mechanism, we trained the system over almost 10 weeks worth of data, which includes the entire logged traffic of Yahoo's native ads marketplace (many billions of impressions). We use both, sAUC and LogLoss metrics (see Section 4.2.2), to measure offline performance, where each event is used for training the system before applied to the performance metrics. The tested system parameters included:

- Hyper-parameters tuning metric: LogLoss

- Tuning cycle length: 3 hours ($L = 12$, with 15 min. train periods)
- Tuned hyper-parameters: SGD primary step size η_0, AdaGrad α, and AdaGrad β, regularization coefficient λ
- Hyper-parameters sets generating function $\mathcal{F}$: scale parameters $\{0.5,\ 1.0,\ 1.5\}$, $S = 3$, and $P = 3^4 = 81$
- Hyper-parameters constraints Ψ: $\eta_0 \in [0, 1],\ \alpha \in [0, 10^3],\ \beta \in [0, 20],\ ,\ \lambda \in [0, 1]$
- 16 anchor configurations are included

As a baseline approach, we used grid search and chose the "best" hyper-parameters set after training over one week worth of data. Then, we used this set and train a system with no tuning mechanism (referred to as *stale* system) over the same logged data, measuring the same performance metrics.

5.1.2 Results

Figure 5 presents the sAUC and LogLoss lifts of the tuning system when compared to those of the baseline stale system, measured every 3 hours. Examining the figure it is observed that both lifts provided by the tuning mechanism are increasing with time reaching 0.51% and 0.66% average increase in LogLoss and sAUC during the last week of the experiment. The temporal normalized hyper-parameters selected by the tuning mechanism are plotted in a logarithmic scale versus time[6]. It is observed that the most dynamic parameter is the regularization coefficient which is practically vanishes with time.

5.2 Online Evaluation

5.2.1 Setup

To evaluate the tuning mechanism we used the science buckets environment described in Section 2.4, and launched a bucket serving around 1% of all Gemini native traffic for a few days. The performance of the bucket, in terms of CPM (average cost per 1000 impressions), were measured and compared to the science control bucket who runs the same code as production in the science environment and serves also around 1% of the traffic. We used the same system parameters as in the offline evaluation (see Section 5.1.1) except that the tuning cycle is set to one hour ($L = 4$, with 15 min. train periods).

5.2.2 Results

Figure 7 presents the CPM lifts of the tuning bucket over the control bucket during 8 days for 1% of all traffic and of Yahoo Home-Page section traffic. As some sections have their own ad serving implementation (e.g., mail) that adds noise to the evaluation, we focused on Yahoo Home-Page section traffic in order to measure the actual lift provided by our method. The average measured lifts in CPM during the online experiment were 4.3% over all traffic, and 8.3% over the Yahoo Home-Page section traffic. Those improvements were later validated using a larger bucket in the production environment, before ramping-up the tuning system to serve the entire traffic. In retrospective, it is estimated that our hyper-parameters tuning method provided $\sim 5\%$ CPM lift over all traffic since it was pushed into production.

To conclude this section we note that the CPM lifts reported in the online evaluation are much higher than the sAUC and LogLoss lifts reported in the offline evaluation. This gap demonstrates the problematic issue of using offline metrics for predicting online performance.

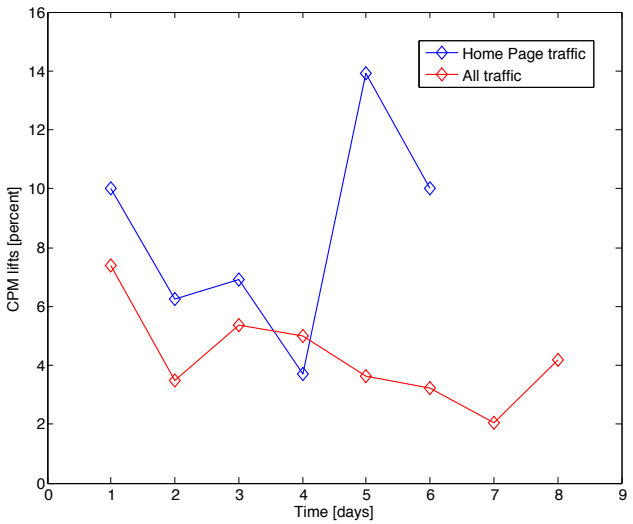

Figure 7: Hyper-parameters tuning mechanism revenue (CPM) lifts over time for all traffic and for Yahoo Home-Page section traffic.

6. CONCLUSIONS AND FUTURE WORK

In this work we present an online hyper-parameters adaptive tuning mechanism for the OFFSET algorithm, which drives the ad click-prediction models in Yahoo's Gemini native. Our approach takes advantage of OFFSET implementation map-reduce architecture, and trains many models in parallel with different hyper-parameters sets for each incoming batch of new logged data. This way the system is tuned to match changing market trends, and other temporal dynamics. The algorithm was pushed into production in 2016Q2 and has been providing a hefty 5% lift in Gemini native revenue since. We note that our approach is generic and may be applied to optimize other learning systems.

Future work may include the use of more sophisticated hyper-parameters sets generation functions. In particular, since the performance metric is a non-convex function of the hyper-parameters, numerical methods such as the *Nelder-Mead* method [9], used to find the exterema of an objective function in a multidimensional space, may be combined into our tuning framework.

7. REFERENCES

[1] Michal Aharon, Natalie Aizenberg, Edward Bortnikov, Ronny Lempel, Roi Adadi, Tomer Benyamini, Liron Levin, Ran Roth, and Ohad Serfaty. Off-set: one-pass factorization of feature sets for online recommendation in persistent cold start settings. In *Proc. RecSys'2013*, pages 375–378, 2013.

[6]Each temporal hyper-parameter is divided by its counterpart within the stale system hyper-parameters set

[2] James Bergstra and Yoshua Bengio. Random search for hyper-parameter optimization. *Journal of Machine Learning Research*, 13(Feb):281–305, 2012.

[3] Simon Chan, Philip Treleaven, and Licia Capra. Continuous hyperparameter optimization for large-scale recommender systems. In *Big Data, 2013 IEEE International Conference on*, pages 350–358. IEEE, 2013.

[4] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, 2008.

[5] John Duchi, Elad Hazan, and Yoram Singer. Adaptive subgradient methods for online learning and stochastic optimization. *The Journal of Machine Learning Research*, pages 2121–2159, 2011.

[6] Benjamin Edelman, Michael Ostrovsky, and Michael Schwarz. Internet advertising and the generalized second-price auction: Selling billions of dollars worth of keywords. *The American economic review*, 97(1):242–259, 2007.

[7] Tom Fawcett. An introduction to ROC analysis. *Pattern recognition letters*, 27(8):861–874, 2006.

[8] Frank Hutter, Holger H Hoos, and Kevin Leyton-Brown. Sequential model-based optimization for general algorithm configuration. In *International Conference on Learning and Intelligent Optimization*, pages 507–523. Springer, 2011.

[9] John A Nelder and Roger Mead. A simplex method for function minimization. *The computer journal*, 7(4):308–313, 1965.

[10] Jasper Snoek, Hugo Larochelle, and Ryan P Adams. Practical bayesian optimization of machine learning algorithms. In *Advances in neural information processing systems*, pages 2951–2959, 2012.