Continuous Deployment of Machine Learning Pipelines

ABSTRACT

Machine learning is increasingly pervasive in many business and scientific applications. The life cycle of machine learning applications does not end with the training of the machine learning pipelines. Once trained, these pipelines must be deployed in an environment where they answer prediction queries. In order to guarantee accurate predictions, one has to continuously monitor and update the machine learning pipelines. Current deployment platforms update the pipeline using online learning methods. When online learning alone is not adequate to guarantee the prediction accuracy, some deployment systems provide a mechanism for automatic or manual retraining of a pipeline and deploy it back into the system. While the online training is fast, the retraining of the pipeline is time-consuming and adds extra overhead and complexity to the process of deployment.

We propose a novel approach for continuously updating the deployed pipelines using a combination of the incoming real-time data and the existing batch data. We offer fast sampling techniques for including the historical data in the training process, thus eliminating the need for complete retraining of deployed pipeline. Moreover, we offer several optimizations such as *live statistics analysis* and *materialization of preprocessed data* to minimize the total training and data preprocessing time. In our experiments, we show that our continuous training approach updates the pipeline more frequently while using fewer resources which results in an improvement of 1.6% in error rate and up to 2 orders of magnitude faster than state-of-the-art deployment approaches.

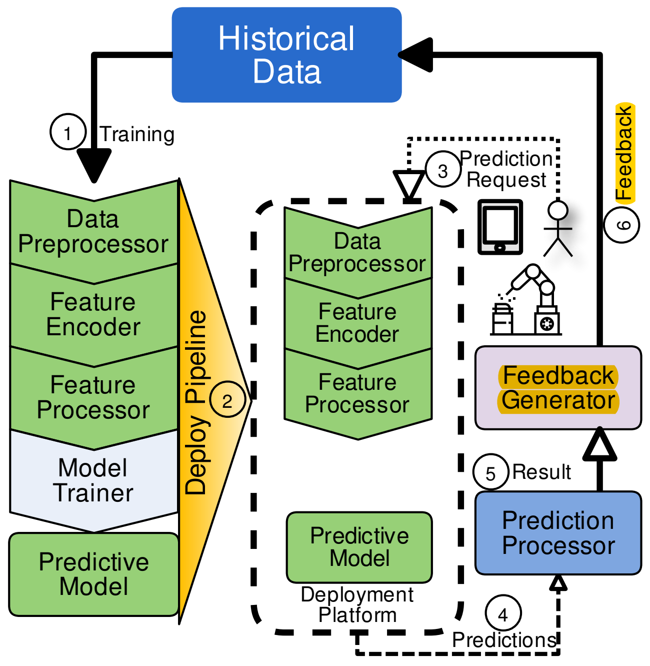


Figure 1: Deployment process of machine learning pipelines

1 INTRODUCTION

Machine learning techniques are increasingly being used in industrial and scientific applications to gain insight from the data. Typically, a machine learning pipeline is a series of complex data processing steps to process a labeled training dataset and results in a machine learning model. The machine learning pipeline is then used to make predictions on new unlabeled data. To fully utilize the pipeline, it has to be deployed into an environment where it can answer prediction queries in real-time.

After the pipeline is deployed, new training data may become available. In order to adapt to the new training data, one has to train and redeploy the pipeline. In many real-world use cases, training datasets are very large which may require hours of training to result in a pipeline. Therefore, it is not feasible to train new pipelines frequently. This means that the deployed pipeline is not always up-to-date. Online learning methods are utilized to provide fresh and up-to-date pipelines. However, unless the online learning method is highly tuned to the specific use case, they do not guarantee a high prediction accuracy [18, 19]. This results in a trade-off between the training cost and accuracy.

Figure 1 shows the typical lifecycle of a machine learning application in real-world use cases. From an existing dataset, a pipeline consists of several data and feature processing steps, and training algorithm is trained ①. Then, a deployment platform deploys this pipeline ② where the pipeline will answer prediction queries arriving at the platform in real-time ③. The pipeline first processes each prediction request (using the same set of steps used in the training phase) and makes a prediction ④. The resulting predictions are further processed and converted to meaningful actions before being presented back to the users or the entities issuing the prediction queries ⑤. Based on the result of the prediction query, feedback, containing the original prediction query and the correct label, may be generated that is appended to the existing training data ⑥. Throughout the deployment process, it is possible that the accuracy of the predictions being made drop below an acceptable threshold due to changes in the distribution of the incoming prediction requests. As a result, pipelines are regularly retrained and redeployed back into the deployment platform.

Several real-world use cases follow the workflow described in Figure 1. One example is the problem of Ad click prediction [19]. In Ad click prediction, the pipeline typically consists of extracting features from the users and ads. Logistic regression models have shown to perform well in this setting [19]. Prediction queries consist of the user's info and a pool of available ads for displaying to the user. Once a prediction score for each of these ads are made, an Ads selector unit (similar to the Prediction Processor in the Figure) picks a set of ads with the highest score to display to the user. Based on the action of the user (click or no click), new training data, containing the original prediction request and the label (+1 for a clicked ad and -1 for a non-clicked ads) will be generated which is stored with the existing training data (similar to the Feedback Generator in the Figure). maybe another use case (IoT or Steel coil flatness prediction)

Many existing platforms, such as Velox [8], Clipper [9], Laser [1], and TensorFlow Extended [3], provide support for deployment of machine learning pipelines. Since the online training of the models and pipelines may not be adequate to ensure accurate predictions, these platforms, either automatically or manually, facilitate the training and re-deployment of the pipelines and models [8]. In many cases, the amount of training data is very large, thus training a new pipeline may take hours (or even days) [3]. During the pipeline training, incoming prediction requests are answered by the old pipeline and new real-time training are accumulated. By the time the pipeline training data is over, enough data has been accumulated to prompt for a new round of pipeline training. As a result, in the current deployment platforms, the deployed pipelines are out of date which results in lower prediction accuracy.

We propose a deployment platform that continuously updates the pipelines (thus providing up-to-date pipelines) using a combination of the historical and incoming data. Our deployment platform updates the pipeline using the incoming feedback data (we call this the real-time training data) similar to how online machine learning algorithms work and performs small batch updates using the historical data. Our solution offers two key optimizations.

Proactive training. Instead of fully training a new pipeline using the historical training data, we continuously update the existing pipeline using small sampled batches of the historical data. The deployment platform forwards each batch of the data through the pipeline and performs a partial model update using this batch. The updated pipeline is immediately ready for answering prediction queries. Our experiments show that proactive training of the pipeline achieves more accurate predictions over time and requires fewer resources when compared to the full pipeline retraining.

Online Statistics Computation and Data Materialization Our deployment platform updates the pipeline using the real-time training data by employing advanced online machine learning algorithms (such as Adagrad [**?**]), before storing it for the proactive training. Since the real-time training data is traveling through the pipeline during the online training phase, we compute statistics required by the pipeline, update the pipeline components, and transform and materialize the data before storing it with the rest of the historical data. As a result, during the proactive training, the data processing and the feature processing steps of the pipeline are skipped and the materialized data is directly used in the model trainer component of the pipeline.

In summary, our contributions are:

- A platform for continuously training deployed machine learning pipelines that adapts to the rate of the incoming data.
- Proactive training of the deployed pipelines that frequently updates the pipeline in-place using a combination of the historical and the real-time data and increases the prediction accuracy when compared with state of the art.
- Efficient pipeline training by online statistics computation and data materialization, thus guaranteeing the availability of up-to-date pipelines for answering prediction queries.

The rest of this paper is organized as follows: Section 2 describes the details of our continuous training approach. In Section 3, we introduce the architecture of our deployment system. In Section 4, we evaluate the performance of our continuous deployment approach. Section 5 discusses the related work. Finally, Section 6 presents our conclusion and the future work.

2 CONTINUOUS TRAINING AND SERVING

The underlying optimization algorithm we utilize for continuously training the deployed model is Stochastic Gradient Descent (SGD). Using SGD enables us to make frequent updates to the model, thus increasing the quality and the freshness of the deployed model. SGD algorithm has several parameters and in order to work effectively, they have to be tuned. In this section, we first describe the details of SGD and its parameters and our approach in tuning these parameters for our platform. Then we describe how we take advantage of the properties of SGD to implement our proactive training, online statistics computation, and data materialization. Another important aspect of any deployment platform is the ability to monitor the quality of the deployed model. We present our method for evaluating the quality of the deployed model and how we guarantee high-quality models. In the last part of this section, we describe how utilizing our continuous training approach improves the efficiency of online advertising example described in Section 1.

2.1 Stochastic Gradient Descent

Stochastic Gradient Descent (SGD) is an optimization strategy utilized by many machine learning algorithms for training a model. SGD is an iterative optimization technique where in each iteration, a sample of the data is used to make updates to the model. SGD is suitable for large datasets as it does not require scanning the entire data in every iteration [5]. SGD is used in different machine learning tasks such as classification [31, 19], clustering [6], and matrix factorization [15, 11]. It is also widely used in neural networks for training the networks on large datasets [10]. Prominent applications of SGD in neural networks are the work of Google Deepmind team that managed to train neural networks that defeat humans in the game of Go [25] and mastering Atari games [20].

In our example application, a logistic regression model is trained using the SGD optimization method [19]. In logistic regression, the goal is to find the weight vector (w) that maximizes the conditional likelihood of labels (y) based on the given data (x) in the training dataset:

$$w^* = \text{argmin}_w ln(\prod_{i=1}^{N} P(y^i | x^i, w))$$

where N is the size of the training dataset. To use SGD to find the optimal w, we start from initial random weights and in each step make small updates based on the gradient of the loss function:

$$w^{t+1} = w^t + \eta \sum_{i \in S} x^i (y^i - \hat{P}(Y^i = 1 | x^i w))$$

where η is the learning rate parameter and S is the random sample in the current iteration. The algorithm continues until convergence (when the weight vector does not change after an iteration).

Learning Rate. A very important parameter of stochastic gradient descent is the learning rate. The learning rate controls the degree of change in the weights in each iteration of SGD. The most basic approach of tuning the learning rate is setting it to an initial small value and after each iteration decrease the value by a small factor. In complex and high-dimensional problems, this simple tuning approach for the learning rate is not effective [23]. Adaptive learning rate methods such as, Momentum [22], Adam [14], RMSPROP [27], and AdaDelta [30] have been proposed. These methods automatically adjust the learning rate value in every iteration and speed up the convergence time. Moreover, some of the adaptation methods perform per coordinate modification [23, 27, 30]. This is important because not all the parameters of the weight vector contribute the same way and some of the parameters change more rapidly during the training process.

Sample Size. Another parameter of stochastic gradient descent is the sample size. SGD is guaranteed to converge to a solution regardless of the sample size. However, the sample size can greatly affect the time that is required to converge. Two extremes of the sample size are 1 (every iteration only considers 1 data item) and N (every iteration consider the entire data set, similar to normal batch gradient descent). Setting the sample size to 1 increases the model update frequency, however, it also results in noisy updates. Therefore, more iterations are required for the model to converge. Using the entire data in every iteration leads to more stable updates and as a result, the number of iterations required for the model to converge is fewer. However, each iteration takes more time as more data has to be processed. The most common approach is to set the sample size to a value small enough so that each sample can be processed quickly but large enough so the updates are not noisy (called a mini-batch gradient descent).

Distributed SGD. To efficiently train machine learning models on large datasets, scalable techniques have to be employed. SGD inherently works well with large amounts of data because it does not need to scan every data point during every iteration. However, for very large datasets, SGD has to perform many iterations in order to converge. To decrease the running time, large datasets can be distributed among multiple nodes, where each node will compute the gradients on a subset of the data in parallel. After the initial computation of the gradients, they are all sent to one node where the final gradients are computed.

2.2 Proactive Training

We use the iterative nature of SGD in the design of our continuous training process. After the initial model is deployed, new iterations of SGD can be performed on a combination of the existing and new data. Once the gradients are computed, we update the deployed model. However, the two parameters of SGD (learning rate and sample size) play an important role. They have to be tuned to increase the efficiency of the training. Choosing a very small sample size may result in inaccurate updates and as a result, degrade the quality of the deployed model. On the other hand, a very large sample size leads to a lengthy training process and less frequent model updates. Similarly, learning rate should be adapted accordingly. We view proactive training as an extension of the offline batch training. Therefore, the process of choosing the best sample size and learning rate adaptation technique is similar to static training. Different hyperparameter tuning techniques are proposed for finding the best set of hyperparameters. Most common and simplest approaches are grid search and random search [4]. We use a grid search over the initial training data to find the best hyperparameters (in our system, learning rate adaptation technique and sample size). Once the initial model is trained and deployed, the same set of parameters are used for the proactive training.

Scheduling rate. An extra parameter of proactive training is the scheduling rate. In offline training, iterations of SGD are executed one after the other until convergence. In proactive training, the scheduling rate defines the frequency of SGD iteration execution. The scheduling rate plays an important role as it directly affects the freshness of the deployed model. However, a high scheduling rate results in many frequent SGD iterations which incur an overhead on the deployment system as it is using a lot of resources. A small scheduling rate also affects the model freshness. To increase the efficiency of the system a scheduler component is designed that is tasked with scheduling new iterations of SGD. Similar to learning rate tuning, we use an adaptive approach to adjust the scheduling rate. We describe a method for tuning the scheduling rate based on the rate of the incoming training data. The scheduling rate is increased as the rate of the incoming training data increases and vice versa. This helps in adapting the model to the new training data.

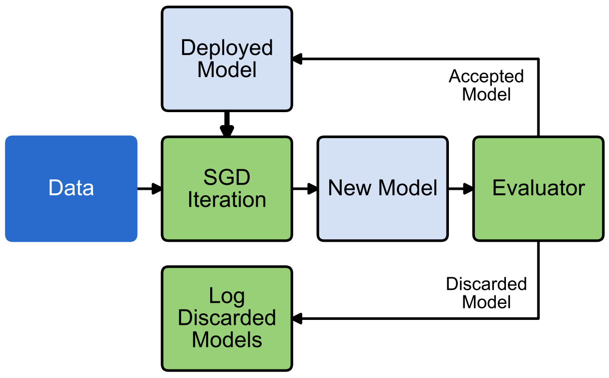


Figure 2: Model Evaluation

2.3 Online Statistics Computation and Data Materialization

Before training the machine learning model, the training dataset has to be processed by the pipeline. Different components of the pipeline require statistics over the dataset to be calculated before they process the data. Computing these statistics require scans of the data. In our deployment platform, we provide a mechanism for online computation and update of the statistics required for different components of the pipeline. In our prototype, we implemented a standard scaler, a missing value imputer, and a one-hot encoder. The above components require the mean, the standard deviation, and for the one-hot encoder, the hash table of the unique categorical parameters in every feature column. Our deployment platform also accommodates custom pipeline components. When new training data arrives at the system, the platform directs the data to the corresponding component so they update their underlying statistics and data structures. Computing the required statistics online reduces the model training time as computing these statistics offline is time-consuming. Every pipeline component needs to transform the data when it is updating its statistics. The component then passes the transformed data to the next component. Once every pipeline component updates their statistics, the resulting transformed data is ready for model training. We provide an optional feature that allows materialization and storage of the transformed data on disk. As a result, during the next SGD iterations, the system skips the preprocessing steps of the pipelines and directly accesses the transformed data to train the model. Storing the transformed data significantly reduces the model training time.

Dynamic model size. Depending on the type of the pipeline components, the size of the final model may need to be adjusted during the serving of the model. For example, one-hot encoding and data bucketization both may generate new features after processing new training data. After every statistics update, we analyze the changes made in the pipeline. If any of the changes result in an increase in the model size, we dynamically adjust the model size in the next proactive training.

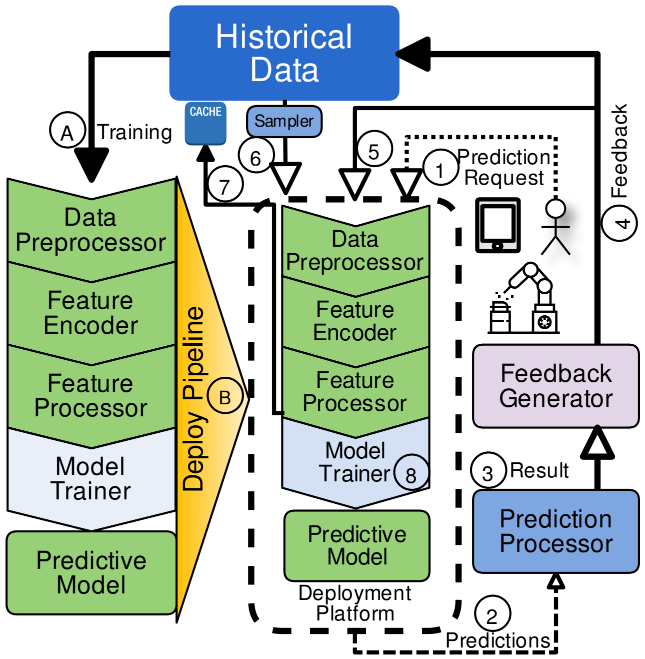


Figure 3: Ads Serving Continuous Training

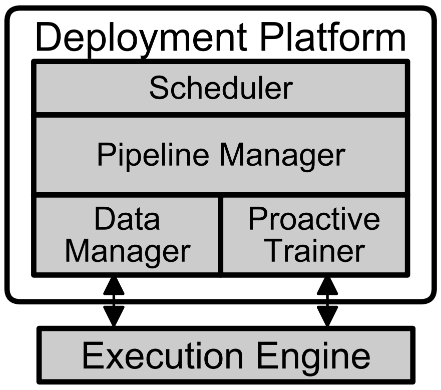


Figure 4: System Architecture

2.4 Model Stability

To ensure that SGD iterations do not degrade the quality of the model, a model evaluator is used to assess the quality of the model. Figure 2 shows the process of model evaluation. Every iteration of the SGD, uses the latest deployed model as an initial starting point and updates the model based on the training data. The evaluator assesses the quality of the model using an evaluation dataset. If the quality of the model has degraded, the update is discard and the model is logged. The engineers or administrators of the system can study the log in order to investigate any potential issues with machine learning pipeline or the incoming training data.

2.5 Improved Example Application

Figure 3 shows how our deployment approach improves the example application described in Section 1. After the training of the initial model Ⓐ, the model and the pipeline are deployed to the deployment environment Ⓑ. Prediction requests are sent by the user to the deployment environment ① where based on the current model for each ad a score is predicted ②. Based on the score a few ads are shown to the user ③. Depending on whether or not the user clicks on them, feedback is sent back to the deployment system ④. Similar to current approaches the data is stored in the click log database. However, contrary to the existing methods, the data is routed to the deployment platform immediately. The deployment platform forwards the training data to the pipeline components to compute the required statistics ⑤. The deployment platform periodically samples the historical data ⑥. Based on this sample and the new training data, an iteration of SGD is performed which updates the deployed logistic regression model. If the new model is accepted, it is redeployed ⑦. New prediction requests that arrive at the system will be answered by the new model. In the new workflow, the deployment platform continuously updates the pipeline and the deployed model without requiring a full retraining over the click log dataset. The deployment platform ensures that the model is always up-to-date and the information about new users or new ads are taken into account in the next iteration of stochastic gradient descent. This increases the model freshness without affecting the model quality.

3 DEPLOYMENT PLATFORM

Our proposed deployment platform comprises four components; pipeline manager, data manager, scheduler, and proactive trainer. Figure 4 gives an overview of the architecture of our system and the interactions among its components. At the center of the deployment platform is the pipeline manager, which monitors the deployed pipeline, processes training data and prediction requests, and continuously trains the model. The data manager and the scheduler enable the pipeline manager to perform proactive training. The proactive trainer component executes iterations of SGD on the deployed model. The proposed design decouples the components of the platform from the execution engine, which is responsible for make the model available for receiving and answering prediction requests. This design enables us to switch the execution engine without requiring major changes to the deployment platform.

This Figure can be removed

3.1 Scheduler

The scheduler component is responsible for scheduling a new proactive training. The scheduler communicates with pipeline manager to instruct when to execute the proactive training. The scheduler allows for two types of scheduling mechanism, namely *static* and *dynamic*.

The static scheduling uses a user-defined parameter that specifies the interval between executions of the proactive training. This is a simple yet useful mechanism for use cases that require periodical updates of the model when the interval is known a priori (for example, every minute or hour). The dynamic scheduling tunes the scheduling interval based on the rate of the incoming prediction, prediction latency, and the time that it takes to execute the proactive training. The scheduler uses the following formula to compute the time when to execute the proactive training:

$$T' = S * T * pr * pl$$

where T' indicates the time in seconds when the next iteration is scheduled to run, T is the execution time of the last proactive training, pl and pr are the average prediction latency and the average number of prediction requests per second. S is the slack parameter. Slack is a user-controlled parameter that provides a hint to the scheduler about the possibility of surges in the incoming prediction requests and training data. A large slack value (≥ 10) forces the scheduler to schedule the next proactive training in a conservative manner, allocating most of the resources of the deployment platform to the query answering component. A small slack value ($10 \geq S \geq 2$) enables the scheduler to schedule the next proactive training using a shorter time interval. As a result, the deployment platform allocates more computing resources for training the pipeline. A slack value of smaller than 2 is not recommended as it may: imprecise language increase the latency of prediction answering component.

T is measured by the scheduler component itself while pr and pl are measured by the deployment platform. Using the formula to schedule the next proactive training, the scheduler ensures that all the prediction requests that arrived during the last proactive training and all new prediction requests are answered before a new iteration is scheduled. Moreover, the scheduler assumes that the entire resources of the computing cluster are being used by the proactive trainer and therefore the prediction answering component is completely blocked while the proactive training is being executed. By removing the above assumption, proactive training can be scheduled to execute more frequently.

Rework or remove

3.2 Data Manager

The data manager component is responsible for storage of historical data, processing the incoming training data, and providing the proactive trainer with a new batch of training data for every iteration.

Historical data is typically large and may not fit in the memory or disk of a single machine. The data manager handles the communication with the underlying storage unit. When new training data becomes available, the data manager appends the incoming data to the existing historical data. Moreover, the incoming training data is forwarded to the pipeline manager to update the statistics required by the pipeline components.

When a new proactive training is scheduled to run, the data manager is responsible for providing the batch of training data. The data manager provides two different sampling approaches. A simple random sampling technique that creates a random sample from the entire historical data and a time-based sampling technique that creates a random sample from a given time interval. The data manager also allows for the interval to be of length zero, which indicates no sampling will be performed. After the sample is assembled, the recent training data is appended to the sample and the combination is provided to pipeline manager for next proactive training process.

The time-based sampling approach enables the proactive trainer to train the pipeline using more recent training data. In many of the real world use cases (e.g., e-commerce and online advertising), this is an important feature, as historical data may lose their importance and should not be used in training the pipeline. If older data are irrelevant to the training of the pipeline, setting the time interval to a small value adapts the model to the newer data faster. However, sometimes the incoming training data is not time-dependent (e.g., image classification of objects). In these scenarios, setting the interval length in time-based sampling to a larger or even using the simple random sampling of the entire historical data may be more beneficial. In our experiments, we investigate the effect of the different sampling approaches on the quality of the model.

To increase the performance of the sampling operation, we utilize a data partitioning technique. Upon arrival of new training data, the data manager assembles a partition of the data and creates an index for the partition using the average of the timestamps of the data that resides in the partition. If the underlying execution engine allows for in-memory data processing, the data manager stores the partition in memory. This allows quick looks ups when performing the sampling, as data belonging to specific time intervals can be accessed directly. When time-based sampling technique is used, the data manager uses samples the partitions instead of the individual data points.

Rework the sentence

Based on the specified time interval, the data manager creates a list of the data partitions that belong to the specific time interval. Data manager then samples from the list of the partitions that belong to the specified time interval. The partitions are assembled to create the final sample data. This sample is then passed to the pipeline manager and used for the next proactive training.

The data manager also allows for new training datasets to be registered while the model is being served. The new dataset is merged with the existing historical data and immediately becomes available for next proactive training processes.

3.3 Pipeline Manager

The pipeline manager is the most important component of the system as it loads the offline trained pipeline, continuously trains the pipeline after the deployment, evaluates the model update before applying the changes to the deployed model, and exposes the model to answer prediction queries.

Once a pipeline is deployed into the platform, the pipeline manager monitors the pipeline. The scheduler component informs the pipeline to execute the next proactive training. The pipeline manager then requests the data manager to provide the training dataset for the next proactive training. Once the training dataset is received, the pipeline manager provides the proactive trainer component with the current model parameter and the training dataset. Once the training is over, the proactive trainer sends the updated model back to pipeline manager. To ensure the quality of the model has not dropped, the pipeline manager uses an independent evaluation set to evaluate the quality of the model. If the quality of the model does not degrade, the pipeline manager replaces the existing model with the new one.

When new training data arrives at the system, the data manager forwards the data to the pipeline manager. The pipeline manager directs the data through the pipeline one component at a time, where each component will receive the data, update their statistics, transform the data, and finally pass the transformed data to the next component. The model training component of the pipeline is skipped as the model is updated separately in the proactive trainer component. If the data materialization is enabled, the data processed by the pipeline is sent back to the data manager to be stored with the rest of the materialized data. The data manager also forwards prediction requests to the pipeline manager. Similar to training data, the pipeline manager also sends the prediction request through the pipeline to perform the necessary data preprocessing. Using the same pipeline to process both the training data and prediction requests guarantees that the same set of transformations are applied to both types of data (training and prediction requests) and prevents inconsistencies between training and serving that is a common problem in model deployment [3]. After the prediction request is processed, the pipeline manager uses the model to make a prediction.

3.4 Proactive Trainer

The proactive trainer is responsible for training the model by executing iterations of SGD. It is also responsible for tuning the learning rate parameter of SGD. In training process, the proactive trainer receives a training dataset and the current model parameters from the pipeline manager, then performs one iteration of SGD and returns the updated model to the pipeline manager. Although iterations are independent of each other, the proactive trainer needs to store the necessary information for computing the learning rate for next iterations. The proactive trainer is the only component that is tightly coupled with the execution engine as it directly executes the code on the engine. Therefore, separate implementations have to be provided for the different execution engines.

3.5 Execution Engine

All of the components of our deployment platform described so far, except for the proactive trainer, are decoupled from the execution engine. In our deployment platform, any data processing platform capable of processing data both in batch mode (for continuous training) and streaming mode (answering prediction requests) is a suitable execution engine. Apache Spark [28] is a distributed data processing platforms that can support both stream and batch data processing. It works with data in memory and on disk which speeds up the execution of the proactive training.

Current Prototype. In our current prototype, we are using Apache Spark [28] as the execution engine. The data manager component uses Hadoop Distributed File System (HDFS) for storing the historical data [24]. We also leverage some of the components of the machine learning library in Spark to implement the proactive trainer. To enable real-time prediction answering we use Spark streaming [29]. Spark streaming allows us to define the parallelism parameter (*pn*) and extract prediction requests rate and latency (*pr* and *pl*). The scheduler uses these parameters to schedule a new proactive training.

This paragraph is more relevant to Evaluation Section

4 EVALUATION

To evaluate our continuous training approach, we perform several experiments. We first describe the setup of our experiments including the computing cluster, the deployed pipeline, and how we simulate a real production environment by streaming a large real-world dataset through our deployment platform. We discuss the effect of different parameters (learning rate adaptation, sampling strategy, and scheduling policy) on the quality and training time of the model. Then, we discuss the effects of proactive training on the quality and freshness of the model and compare them to a model that is trained periodically. Finally, we evaluate the training time of the continuous training approach and the effects of online statistics computation and data materialization optimizations on the training time.

4.1 Setup

We evaluate our deployment method in a distributed environment consists of 21 nodes (1 master, 20 slaves). Each node is running on an Intel Xeon 2.40 GHz 16 core processor and has 28 GB of dedicated memory for running our prototype. We use Apache Spark 2.2.0 running on Hadoop 2.7. Each executor node has 16 task slots (a total of 320 slots).

To demonstrate the deployment platform, we designed the following machine learning pipeline and simulation:

Criteo Pipeline. The Criteo pipeline consists of 5 operations: input parser, missing value imputer, standard scaler, one hot encoder, and logistic regression model trainer. The Terabyte Criteo click log dataset is used for benchmarking algorithms for clickthrough rate (CTR) prediction [7]. It contains 24 days of user click logs. The dataset contains 13 numerical and 26 categorical features. In all of our experiments, we are using the data from the first 3 days (Day 0 to Day 2) of the Criteo dataset. Day 0 is used for the initial offline training of the pipeline. The data from Day 1 and Day 2 are used as streaming data sources. To evaluate the quality of the pipeline, we use a sample of Day 6 to compute the logistic loss.

Criteo Data Simulation. We simulate a production environment by streaming 2 days of the Criteo dataset. The data from each day is divided into 1440 smaller batches and stored on disk. Each batch represents one minute of data. We use spark streaming to read the data files one by one and stream them through the deployment platform. All the experiments are using Day 1 and Day 2 as streaming sources unless specified otherwise.

4.2 Learning Rate Adaptation Method

In Section 2.1, we discussed the importance of learning rate tuning for training a model using the Stochastic Gradient Descent (SGD) optimization method. Proactive training is an extension of SGD, and therefore the process of tuning the learning rate adaptation method is not different from tuning it for an offline SGD training.

To find the best learning rate adaptation algorithm, we first train a model using SGD optimization algorithm for 500 iterations using Adadelta, RMSprop, and Adam, three of the state-of-the-art learning rate adaptation techniques. After the training, the models (and the pipelines) trained with different learning rate adaptation techniques are deployed. We use the Day 0 of the Criteo data to investigate the effect of the learning rate adaptation techniques on the Criteo pipeline. Figure 5 shows the logistic loss error rate of different learning adaptation techniques. During the SGD training phase, we capture the logistic loss on the evaluation dataset after 20, 40, 80, 160, 320, and 500 iterations of training.

Adadelta performs very poorly during the offline training phase. The Criteo dataset is a complex and high-dimensional data, where features are a mix of numerical and categorical variables. Since categorical features are not standardized, Adadelta is not able to effectively tune the learning rate for a mix of standardized and non-standardized features. Similar to the offline SGD training, Adadelta performs poorly for proactive training after the model is deployed. In fact, the error rate of the model, starting from the 20th iteration to end of Day 1, stays almost constant throughout the experiment.

Unlike Adadelta, RMSprop reduces the error rate on the evaluation dataset through the offline SGD training. Before the deployment, the error rate of the model is dropped to 0.48. Similarly, after the deployment, proactive training of the model using RMSprop reduces the error rate by 18%.

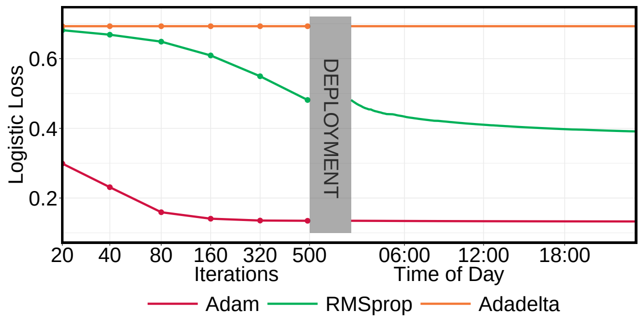


Figure 5: Learning rate adaptation techniques for Criteo pipeline

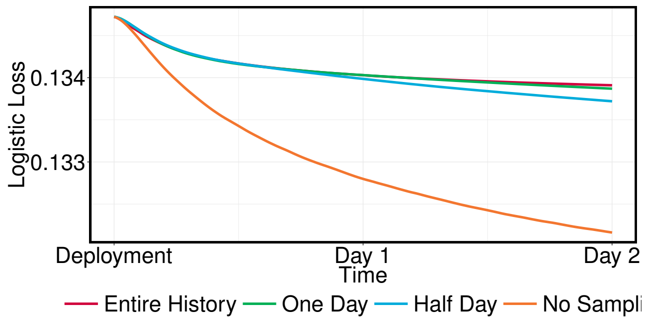


Figure 6: Effect of different sampling modes on quality

Adam has the best performance among the evaluated learning rate adaptation techniques. Using Adam, the model fully converges after 500 iterations of SGD during the offline training phase. After the deployment, the error rate is further reduced by 1.4%. While the reduction in error rate for Adam is smaller than RMSprop, Adam still outperforms RMSprop after a day of proactive training of the model.

This experiment shows that process of selecting the learning rate adaption technique for proactive training is similar to the process of selecting it for the offline SGD training. A learning rate adaptation technique that performs best during the offline training of the model also has the best performance for proactive training, when the model is deployed.

Even though Adam performs best for the Criteo pipeline, this does not indicate that Adam is the best learning rate adaptation technique for proactive training for every other pipeline and model. For every pipeline and dataset, the users have to evaluate the performance of the different learning rate adaptation techniques during the offline training of the model. The method that performs best during the offline training also has the best performance for the proactive training.

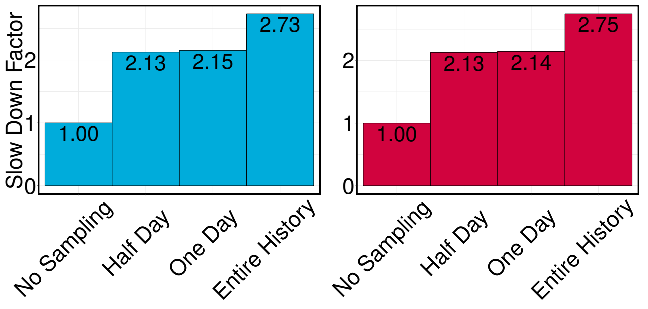
4.3 Sampling Methods

In this section, we analyze the effect of different sampling techniques on both the quality and the training time of the pipeline. We use a sampling rate of 0.1% for all of the experiments where a sampling is performed. This sampling rate is chosen to be equal to the sampling rate used during the initial offline SGD training of the model.

Figure 6 shows how different sampling modes, namely random sampling, time-based sampling, and no sampling, affect the quality of the model in the Criteo pipeline. In both random sampling and time-based sampling modes, first the data is sampled and then the new training data that has arrived at the system recently is appended to this sample and used in the proactive training. In no sampling mode, only the recently arrived data is used in the proactive training. In random sampling approach, the entire historical data is used for creating the sample. In this scenario, the logistic error rate decreases in a very slow manner over time. Since the deployed model is already fully trained on the historical data, using the historical data in the continuous training of the model does not have a big impact on the model quality.

The time-based sampling has a larger impact on the quality of the model. We evaluated the quality of the model for one day and half day time intervals. Using an interval of one day, the time-based sampling decreases the error rate by 0.03% more than when using the simple random sampling (entire historical data). Moreover, decreasing the interval length further, results in a model with lower logistic error rate. In our experiment, using a time interval of half day for the time-based sampling results in an error rate that is 0.14% smaller than when using the simple random sampling. Reducing the time interval in the time-based sampling limits the data in the sample to the more recently generated data, which allows the model to fit to more unseen and more time-relevant data. In the Criteo pipeline, disabling sampling completely has the biggest impact on the error rate. By not sampling the data, the error rate of the model is decreased by 1.9% over the two days deployment period.

This experiment shows that the sampling time interval has a big impact on the quality of the deployed model. For Criteo pipeline, the dataset has a stable distribution, which stays the same throughout the course of the experiment. As a result, limiting the training to the more recent data exposes the model to newer and unseen data which results in bigger changes (toward convergence) in the weights of the model (e.g., no sampling). When the distribution of the incoming training data is stable, using more historical data to continuously train the model has little effect as the combination of historical and new data dampens the effects of the new data on the model (e.g., sampling from the entire history). As a result, the improvement in the model convergence is very small.



(a) Total time

(b) Data processing time

Figure 7: Slowdown factor of different sampling approaches

Figure 7 shows the effect of different sampling operations on the training time of the pipeline. Figure 7a shows that using the time-based sampling (intervals of one and half a day) and simple random sampling (the entire history) increases the total training time by a factor of 2.13 to 2.73 compared to when no sampling is performed. However, the slowdown in training time is not due to the sampling operations. To demonstrate this, for each sampling method, we also capture the amount of time the deployment platform spends in processing the data after the sample is provided by the data manager. Data processing includes applying the pipeline transformations and training the model. Figure 7b show the slowdown in data processing time when using different sampling methods. The slowdown factor for both data processing and the total time is almost identical. Therefore, the increase in time is not due to the sampling operation. After the sampling is performed, more data will be processed by the proactive trainer, which increases the total time.

This demonstrates the ability of the data manager to provide samples from the data without incurring overhead on the deployment platform. The data manager uses a partitioning technique to store the incoming data in a manner that searching and sampling them does not require a scan of the data. When a sample is required, the data manager directly samples from the list of the data partitions that fall within the range of the required time interval.

4.4 Scheduling Policy

In this section, we analyze the scheduling policy of our deployment platform. In our prototype, we simulated 2 days of continuous training of Criteo data using Apache Spark. Since the streaming component of Apache Spark requires a fixed interval for executing mini batches, we analyze the effect of our scheduling policy analytically.

Figure 8 shows the actual execution time of every proactive training throughout the simulation. The execution time of the proactive training ranges from 23 to 53 seconds. In order for the scheduler component to effectively schedule proactive training, it requires the prediction latency, prediction throughput

Is it rate or throughput? Clipper uses the latter

, and a user-defined slack parameter. In our estimation, we use a slack parameter of 10. We estimate the throughput and latency based on the time it takes for the deployment platform to predict the labels of the evaluation dataset. The evaluation dataset contains 2 million data points. The deployment platform is queried using the evaluation dataset every minute and requires 15 seconds to return the predictions in the worst case scenario (when the evaluation dataset is stored on disk). This amounts to a latency of $7 * 10^{(-6)}$ seconds (7.5 micro seconds) and a throughput of 34,000 requests per second.

Based on above parameters, the scheduler computes the scheduling intervals for every execution of the proactive training.

Using the slack parameter, we can guide the scheduler to increase or decrease the scheduling intervals of the proactive training. The slack parameter allows for the deployment platform to accommodate surges in the incoming prediction requests and new training data. In scenarios where sudden surges are expected (e.g., online stores), we recommend a large slack parameter (recommended value is 10).

No exp. validate this

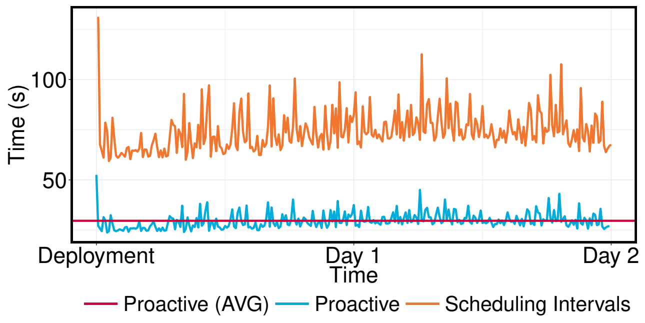


Figure 8: Analysis of scheduling policy

4.5 Model Freshness

We measure the model freshness by two metrics: rate of proactive training execution and rate of new features. The rate of the proactive training execution is determined by the scheduling rate. Performing more frequent training results in models that can adapt to changes in the data more rapidly.

To evaluate the model freshness in terms of the rate of new features, we use the feature encoder to determine the number of new features that arrive at the system. In the Criteo pipeline, the feature encoder is used to transform the categorical features into binary indicator variables.

Figure 9 shows the feature size over time for the first 5 days after deployment of the Criteo pipeline. The initial training data (Day 0) only contains a small portion of all the unique categorical features of the Criteo dataset. The rate of incoming new features is close to 30,000 per minute and around 45 million new features are generated everyday.

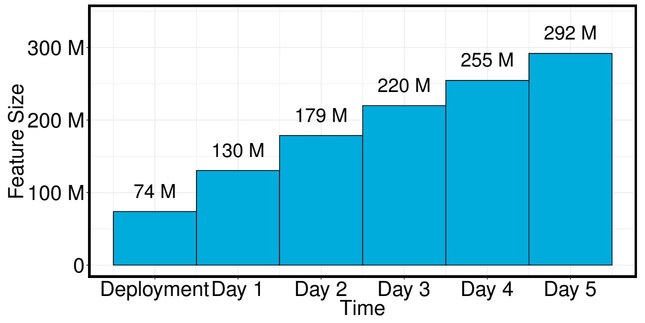


Figure 9: Criteo categorical feature size over time

Using our continuous training approach, we update the pipeline as soon as the new features become available. During the next scheduled proactive training, the model is updated using these new features. As a result, the deployed pipeline is able to answer prediction queries that may contain the same set of features more accurately. Using a daily training approach, any unseen features that arrive at the system are dropped before a prediction is made.

4.6 Proactive Training

In this section, we evaluate the quality of the deployed model. Figure 10 shows the logistic loss of the continuous training and periodical approaches on the Criteo pipeline.

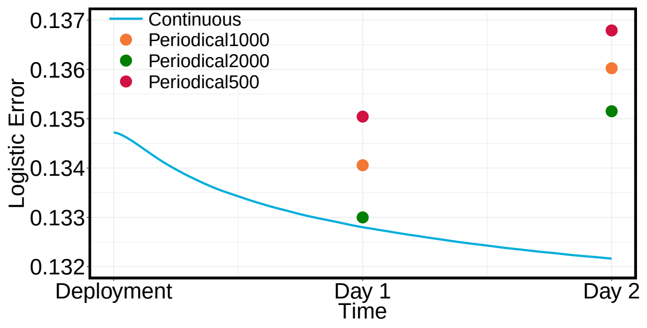


Figure 10: Model quality for different deployment approaches

After the deployment, the continuous training updates the pipeline and trains the model using the incoming training data. The incoming data contains both unseen training observations and new features, as described in experiment 4.5. The continuous training approach trains the model over this data within a short period of time which results in a higher quality and fresher model when compared to the periodical training approach.

To measure the performance of the periodical training approach, we train the model at the end of every day of the simulation. The initial training and periodical training use the same set of parameters. We use Adam learning rate adaptation technique and a sampling rate of 0.1 for training the initial and periodical training of the pipeline.

The initial deployed model converges after 500 iterations. However, after Day 1, the amount of the stored data is doubled, and training the model for 500 iterations results in a model that has an error rate of 0.002 higher than the initial model. Due to the larger number of data points, the periodical training has to train the model longer in order to achieve a lower error rate. To decrease the error rate of the model further, we train the model for up to 2000 iterations. Our experiment shows that continuous training still outperforms periodical training after 2000 iterations. The error rate of the model trained using continuous training is 1.6%, 0.9%, and 0.15% lower than a model trained using the periodical training with 500, 1000, and 2000 iterations respectively at the end of Day 1 of the simulation.

We observe a similar behavior for the second periodical training. Because of the large training dataset after the second day of the simulation, the model requires a longer training period to converge. After two days of training, the continuous training approach results in an error rate that is smaller than the error rates of a model trained using 500, 1000, and 2000 iterations by 3.4%, 2.8%, and 2.2%.

This experiment shows that the continuous training of a machine learning pipeline decreases the error rate while still producing fresher models when compared to the periodical training of the pipeline.

It has to be noted that the poor performance of periodical training after each day, can be alleviated by more advanced training methods or better parameter selection for the underlying SGD optimization algorithm. In this experiment, we demonstrate that using the same set of parameters (learning rate and sample size), we are able to achieve a lower error rate by using the continuous training approach. In a real deployment scenario, when the quality of the model degrades after the periodical training, either the model is discarded or it is trained for a longer period of time using a more sophisticated training algorithm, while the existing model continuous to answer prediction requests.

4.7 Total Training Time

In this section, the total training time for continuous and periodical deployment approaches are measured. The total training time includes the time spent in pre-processing the training data using the pipeline components and training the model. For periodical training, we train the model for 500 iterations even though the experiment in Section 4.6 demonstrated that the error rate of such a model is larger than the continuous approach by 1.6%.

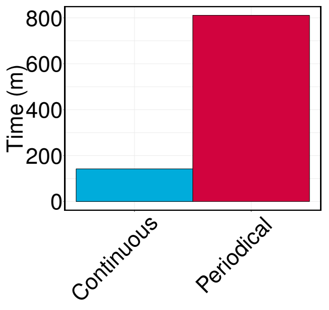
Figure 11a shows the total training time of the Criteo pipeline for different deployment approaches. Using continuous training, the time spent in data preprocessing and model training is smaller than that of periodical training by a factor of 5. This is due to the large amount of redundant data preprocessing and model training that exists in the periodical training approach. In the periodical training, the underlying pipeline is trained from scratch every day, which includes ingesting the data, performing the data transformation steps of the pipeline and finally training the model using the transformed data. However, in continuous training, the pipeline is incrementally updated when new data arrives at the system.

The total training time can be further reduced in the continuous training approach by switching on the online statistics update and materialization optimizations. Figure 11b shows the effect of each optimizations on the continuous training approach. By enabling online statistics update, the pipeline components are updated when new training data becomes available. Therefore, the proactive trainer does not need to update pipeline components and proceeds to transform the data directly. Enabling the online statistics update optimization reduces the total training time by a factor of 3. Moreover, enabling both online statistics and materialization allows the proactive trainer to skip the data processing part of the pipeline completely and directly proceeds to the model training, which only accounts for a small fraction of the pipeline. Our experiment shows that enabling both online statistics update and materialization reduces the total training time by a factor of 18, from 142 minutes of training, when no optimizations are used, to 8 minutes, when both optimizations are switched on.

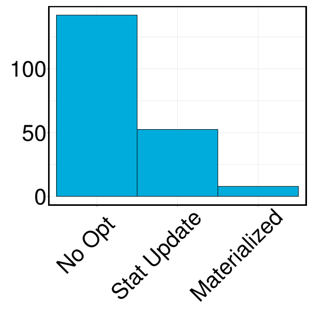
One possible problem with the materialization optimization is the amount of space required to store the materialized dataset. Depending on the type of the data processing, the size of the materialized data may increase. In our experiments, the input data is a combination of Integer and String values, however, after the pre-processing steps of the Criteo pipeline, the data is transformed to large vectors of Double. As a result, the size of the materialized data increases by a factor of 2.

4.8 Discussion

Our experiments show that our continuous training approach outperforms periodical training of deployed models and pipelines. By using proactive training we manage to reduce the average error rate by 1.6%. The frequent updates that the continuous training approach applies to the deployed model is the main reason for the reduction



(a) Deployment Approaches



(b) Optimizations

Figure 11: Total training time for different deployment approaches (with optimizations enabled)

in error rate. As the amount of the existing data increases after each time interval, periodical training requires more training iterations and more advanced techniques to train a model with an acceptable error rate. Continuous training does not face the same issue and the error rate of the model decreases when using the continuous training approach, as demonstrated by the experiments. The continuous training approach enables the model to adapt to the recently unseen data faster and allows the model to be updated with new features. In contrast to the continuous training, in periodical training, new features are only discovered after each training interval (e.g., 1 day for the Criteo pipeline). As a result, the deployment platform discards the newly available features when answering prediction requests until the next training.

The continuous training approach reduces the total training time by a factor of 5 after two days of training. Moreover, the online statistics computation and data materialization optimizations reduce the total training time by 2 orders of magnitude over the state-of-the-art deployment approaches. After the model is deployed, the training time for the continuous training approach stays constant as the frequency of proactive training remains the same. However, periodical training has to process larger datasets at the end of each time interval which leads to the large difference in total training time between continuous and periodical training.

Proactive training is an extension of the SGD optimization algorithm, where during each scheduled training an optional sample of the historical data and the newly available training data are combined and used to train the deployed model in-place. Therefore, tuning proactive training is similar to the process of tuning the SGD algorithm. In our experiments, we show that the learning rate adaptation technique that works best with SGD during the offline training also results in the lowest error rate when used in the proactive training.

We demonstrate how different sampling approaches (simple random sampling, time-based sampling, and no sampling) affect the quality of the model. To perform efficient time-based sampling, the data manager uses a partitioning technique that stores the incoming data in partitions and assigns timestamps to each partition. Our experiments show that using the data partitioning technique, we can effectively provide samples from different time intervals without incurring an overhead on the deployment platform. However, while the sampling operations themselves do not incur any overhead, the extra amount of data that is generated as the result of the sampling increases the proactive training time. To alleviate the issue, we propose a scheduling policy that dynamically adapts to both the rate of the incoming prediction requests and the time required for executing a proactive training. We show that our scheduling policy can effectively execute the proactive training and adapt to the changes in the rate of the incoming data, the prediction latency, the proactive training time, and the sudden data surges.

5 RELATED WORK

Traditional machine learning systems focus solely on training models and leave the task of deploying and maintaining these models to the users. It has only been recently that some systems, for example LongView [2], Velox [8], Clipper [9] , and TensorFlow Extended [3], have proposed architectures that also consider model deployment and query answering.

LongView integrates predictive machine learning models into relational databases. It answers predictive queries and maintains and manages the models. LongView uses techniques such as query optimization and materialized view selection to increase the performance of the system. However, it only works with batch data and does not provide support for real-time queries. As a result it does not support incremental learning. In contrast, our system is designed to work in a dynamic environment where it answers prediction queries in real-time and incrementally updates the model when required.

Velox is an implementation of the common machine learning serving practice. Velox supports incremental learning and can answer prediction queries in real-time. It also eliminates the need for users to manually retrain the model offline and redeploy it again. Velox monitors the error rate of the model using a validation set. Once the error rate exceeds a predefined threshold, Velox initiates a complete retraining of the model using Spark. This deployment method, however, has three drawbacks; retraining discards updates that have been applied to the model so far, the process of retraining on full data set is resource intensive and time consuming, and new datasets introduced to the system only influence the model after the next retraining. Our approach differs, as it exploits the underlying properties of SGD to fully integrate the training process into the system's lifeline. This eliminates the need for completely retraining the model and replaces it with consecutive SGD-iterations. Moreover, our system can train the model on new batch datasets as soon as they become available.

Clipper is another machine learning deployment system that focuses on producing higher quality predictions by maintaining an ensemble of models. It constantly examines the confidence of each model. For each prediction request, it uses the model with the highest confidence. However, it does not incrementally train the models in production, which over time leads to models becoming outdated. Our deployment method on the other hand, focuses on maintenance and continuous updates of the models.

TensorFlow Extended (TFX) is a platform that provides continuous training and deployment of machine learning models. TFX automatically stores new training data, performs analysis and validation of the data, retrain new and fresh models, and finally redeploy the new models. However, data analysis and validation and model retraining are done periodically on batch datasets. As a result, TFX targets use cases that typically require daily updates to the model as the overhead of performing more frequent training and data analysis is too high. TFX provides warmstarting optimization to speed up the process of training new and fresh models. Our continuous training method can be used as a replacement of the continuous training component of TFX. By exploiting the properties of SGD optimization technique, our continuous training method can provide fresher models (seconds to minutes instead of several hours or days) without increasing the overhead.

Weka [12], Apache Mahout [21], and Madlib [13] are systems that provide the necessary toolkits to train machine learning models. All of these systems provide a range training algorithms for machine learning methods. However, they do not provide any management, before or after the models have been deployed. Our proposed system focuses on models trainable using stochastic gradient descent and as a result is able to provide model management both during training and deployment time.

MLBase [16] and TuPaq [26] are model management systems. They provide a range of training algorithms to create machine learning models and mechanism for model search as well as model management. They focus on training high quality models by performing automatic feature engineering and hyper-parameter search. However, they only work with batch datasets. Once models are trained, they have to be deployed and used for serving manually by the users. Our system, on the contrary, is designed for deployment and maintenance of already trained models.

6 CONCLUSIONS

We propose a deployment platform for continuously training machine learning pipelines and models. After a machine learning pipeline is designed and initially trained on a dataset, our platform deploys the pipeline and makes it available for answering prediction queries.

We propose a training approach, called proactive training, that uses the combination of historical data and newly arrived data to train the deployed pipeline. Proactive training eliminates the need for training new models from scratch. To guarantee a model with an acceptable error rate, current deployment approaches require the model and the pipeline to be trained completely from scratch which is a time-consuming and resource-intensive process. As a result of the lengthy training process, fresh models cannot be available to the users. Proactive training addresses the trade-off between model quality and model freshness and manages to provide fresh models without sacrificing the quality. Moreover, our online statistics optimization and materialization reduce the training time by a factor 18. We propose a modular design that enables our deployment platform to be integrated with different scalable data processing platforms. We implemented a prototype using Apache Spark to evaluate the performance of our deployment platform on large datasets. In our experiments, we develop a machine learning pipeline to process and train a logistic regression model over the Criteo click log datasets. Our experiments show that our continuous training approach reduces the total training time by a factor 5, without optimization, and by 2 orders of magnitude, with optimizations enables, when compared to the periodical training of the pipeline. We demonstrate, how our approach addresses the model freshness requirement, by showing that new features in the Criteo dataset are discovered and used to train the model immediately after they are sent to the deployment system. Moreover, we discuss the process of tuning the deployment approach, showing that the same set of parameters that are selected for training the initial pipeline can be used after deployment, during the continuous training.

In the future work, we will integrate more complex machine learning pipelines (e.g., neural networks) into our deployment platform and investigate the effect of concept drift and anomaly on our deployment platform.

REFERENCES

[1] Deepak Agarwal, Bo Long, Jonathan Traupman, Doris Xin, and Liang Zhang. Laser: A scalable response prediction platform for online advertising. In *Proceedings of the 7th ACM international conference on Web search and data mining*, pages 173–182. ACM, 2014.

[2] Mert Akdere, Ugur Cetintemel, Matteo Riondato, Eli Upfal, and Stanley B Zdonik. The case for predictive database systems: Opportunities and challenges. In *CIDR*, pages 167–174, 2011.

[3] Denis Baylor, Eric Breck, Heng-Tze Cheng, Noah Fiedel, Chuan Yu Foo, Zakaria Haque, Salem Haykal, Mustafa Ispir, Vihan Jain, Levent Koc, et al. Tfx: A tensorflow-based production-scale machine learning platform. In *Proceedings of the 23rd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 1387–1395. ACM, 2017.

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

[5] Léon Bottou. Large-scale machine learning with stochastic gradient descent. In *Proceedings of COMPSTAT'2010*, pages 177–186. Springer, 2010.

[6] Leon Bottou, Yoshua Bengio, et al. Convergence properties of the k-means algorithms. *Advances in neural information processing systems*, pages 585–592, 1995.

[7] Olivier Chapelle. Terabyte Criteo Click Logs. http://labs.criteo.com/2013/12/download-terabyte-click-logs-2/, 2013. [Online; accessed 25-September-2017].

[8] Daniel Crankshaw, Peter Bailis, Joseph E Gonzalez, Haoyuan Li, Zhao Zhang, Michael J Franklin, Ali Ghodsi, and Michael I Jordan. The missing piece in complex analytics: Low latency, scalable model management and serving with velox. *arXiv preprint arXiv:1409.3809*, 2014.

[9] Daniel Crankshaw, Xin Wang, Giulio Zhou, Michael J Franklin, Joseph E Gonzalez, and Ion Stoica. Clipper: A low-latency online prediction serving system. *arXiv preprint arXiv:1612.03079*, 2016.

[10] Jeffrey Dean, Greg Corrado, Rajat Monga, Kai Chen, Matthieu Devin, Mark Mao, Andrew Senior, Paul Tucker, Ke Yang, Quoc V Le, et al. Large scale distributed deep networks. In *Advances in neural information processing systems*, pages 1223–1231, 2012.

[11] Simon Funk. Netflix update: Try this at home, 2006.

[12] Mark Hall, Eibe Frank, Geoffrey Holmes, Bernhard Pfahringer, Peter Reutemann, and Ian H Witten. The weka data mining software: an update. *ACM SIGKDD explorations newsletter*, 11(1):10–18, 2009.

[13] Joseph M Hellerstein, Christoper Ré, Florian Schoppmann, Daisy Zhe Wang, Eugene Fratkin, Aleksander Gorajek, Kee Siong Ng, Caleb Welton, Xixuan Feng, Kun Li, et al. The madlib analytics library: or mad skills, the sql. *Proceedings of the VLDB Endowment*, 5(12):1700–1711, 2012.

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

[15] Yehuda Koren, Robert Bell, Chris Volinsky, et al. Matrix factorization techniques for recommender systems. *Computer*, 42(8):30–37, 2009.

[16] Tim Kraska, Ameet Talwalkar, John C Duchi, Rean Griffith, Michael J Franklin, and Michael I Jordan. Mlbase: A distributed machine-learning system. In *CIDR*, volume 1, pages 2–1, 2013.

[17] Mu Li, David G Andersen, Jun Woo Park, Alexander J Smola, Amr Ahmed, Vanja Josifovski, James Long, Eugene J Shekita, and Bor-Yiing Su. Scaling distributed machine learning with the parameter server. In *OSDI*, volume 1, page 3, 2014.

[18] Justin Ma, Lawrence K Saul, Stefan Savage, and Geoffrey M Voelker. Identifying suspicious urls: an application of large-scale online learning. In *Proceedings of the 26th annual international conference on machine learning*, pages 681–688. ACM, 2009.

[19] H. Brendan McMahan, Gary Holt, D. Sculley, Michael Young, Dietmar Ebner, Julian Grady, Lan Nie, Todd Phillips, Eugene Davydov, Daniel Golovin, Sharat Chikkerur, Dan Liu, Martin Wattenberg, Arnar Mar Hrafnkelsson, Tom Boulos, and Jeremy Kubica. Ad click prediction: a view from the trenches. In *Proceedings of the 19th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD)*, 2013.

[20] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin Riedmiller. Playing atari with deep reinforcement learning. *arXiv preprint arXiv:1312.5602*, 2013.

[21] Sean Owen, Robin Anil, Ted Dunning, and Ellen Friedman. *Mahout in Action*. Manning Publications Co., Greenwich, CT, USA, 2011.

[22] Ning Qian. On the momentum term in gradient descent learning algorithms. *Neural networks*, 12(1):145–151, 1999.

[23] Tom Schaul, Sixin Zhang, and Yann LeCun. No more pesky learning rates. *ICML (3)*, 28:343–351, 2013.

[24] Konstantin Shvachko, Hairong Kuang, Sanjay Radia, and Robert Chansler. The hadoop distributed file system. In *2010 IEEE 26th symposium on mass storage systems and technologies (MSST)*, pages 1–10. IEEE, 2010.

[25] David Silver, Aja Huang, Chris J Maddison, Arthur Guez, Laurent Sifre, George Van Den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Veda Panneershelvam, Marc Lanctot, et al. Mastering the game of go with deep neural networks and tree search. *Nature*, 529(7587):484–489, 2016.

[26] Evan R Sparks, Ameet Talwalkar, Michael J Franklin, Michael I Jordan, and Tim Kraska. Tupaq: An efficient planner for large-scale predictive analytic queries. *arXiv preprint arXiv:1502.00068*, 2015.

[27] Tijmen Tieleman and Geoffrey Hinton. Lecture 6.5-rmsprop: Divide the gradient by a running average of its recent magnitude. *COURSERA: Neural networks for machine learning*, 4(2):26–31, 2012.

[28] Matei Zaharia, Mosharaf Chowdhury, Michael J Franklin, Scott Shenker, and Ion Stoica. Spark: cluster computing with working sets. *HotCloud*, 10:10–10, 2010.

[29] Matei Zaharia, Tathagata Das, Haoyuan Li, Timothy Hunter, Scott Shenker, and Ion Stoica. Discretized streams: Fault-tolerant streaming computation at scale. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, pages 423–438. ACM, 2013.

[30] Matthew D Adadelta Zeiler. An adaptive learning rate method. arxiv preprint. *arXiv preprint arXiv:1212.5701*, 2012.

[31] Tong Zhang. Solving large scale linear prediction problems using stochastic gradient descent algorithms. In *Proceedings of the twenty-first international conference on Machine learning*, page 116. ACM, 2004.