~~A Novel Approach to~~ Continuous Training of Large Scale Machine Learning Models

Behrouz Derakhshan
behrouz.derakhshan@dfki.de

Tilmann Rabl
rabl@tu-berlin.de

Volker Markl
volker.markl@tu-berlin.de

ABSTRACT

Machine learning is increasingly pervasive in many business and scientific applications. Making machine learning models ready for serving in a productive environment is a process done in form of pipelines that include several manual, semi-automatic, and automatic steps. ~~Model deployment is the final stage in these pipelines.~~ Once a machine learning model is trained on an initial dataset it is deployed into a system where it can answer prediction queries in a realtime and reliable fashion. If new training data becomes available while the system is running, the model is updated. Currently, these updates are either performed incrementally or they require a complete retraining of the model offline. However, complete retraining of the model is often a time consuming and resource intensive process.

In this paper, we propose a novel deployment method for stochastic gradient descent (SGD)-based machine learning models. SGD is an iterative process, which works well with large data sets. In each iteration of SGD, the model is updated based on one or a sample (mini-batch) of the training items. Using this property of SGD, not only incremental learning can be applied but we also eliminate the need for offline retraining and replace it with continuous mini-batch updates. One iteration of SGD is lightweight and can be executed while the system is running without interrupting the normal flow. Our experiments show that our deployment method can achieve up to an order of magnitude faster ~~model update~~ without degrading the quality of the model and in some case even increasing the quality due to faster adoption to changes in data distribution.

Keywords

ML Model Management; Stochastic Gradient Descent; ML Systems

1. INTRODUCTION

Deployment and maintenance of machine learning models is a crucial step in machine learning applications' life cycle. However, it is one that has received very little attention. ML application's life cycle does not end with training. In fact deployment and serving of the models is the most important aspect that brings the actual business value. In order to gain actual value out of the machine learning models, they have to be monitored in realtime in production environment. These models have to be maintained and constantly updated incrementally to better fit the new data that arrives at the system. The models should also be deployed in an environment that is capable of handling high traffic. While the system is running, it is also possible that new data sets become available. In order to prevent the model's quality from degrading, periodic retraining should be performed to better fit the model with the data that has arrived at the system since the last training. Most of the current machine learning research focuses on training and providing tools to make model training and search easier. Kumar et al. [20] provided an overview of landscape of existing machine learning systems. Except in a very few cases [2, 7], most of the surveyed systems, focus on training of machine learning models and provide little to no support after the model has been trained. Moreover, systems with support for model deployment are still lacking constant monitoring and fast and accurate updates of the machine learning models. ~~Therefore,~~ the whole process of training and deployment of models is usually done manually. The most common approach is as follows; first a model is trained based on an existing dataset residing on disk, this model serves as the initial model. This model is then deployed to an environment where it can answer prediction queries arriving at the system. The system also receives feedback in the form of new training observations. Upon receiving a training observation, the model is updated incrementally. Incremental updates are only supported by certain type of machine learning models, where the underlying optimization strategy allows changes in the model based on individual training items. These incremental updates, however, are not enough to keep the model quality at an acceptable rate and after a while the quality may degrade. Moreover, new data sets from external sources may become available while the system is running which are combined with the existing data. At this point, a new model has to be retrained based on the entire available data and has to be redeployed again.

Example application: to illustrate this model deployment approach, consider ~~as an example~~ the task of predicting the click through rate (CTR) of online advertisements. Online advertising industry ~~is a multi-million dollar industry and CTR prediction is the machine learning problem at the~~

~~heart of this industry [23, 11]. In order to gain profit, search engines typically show a set of advertisements, based on the users' search queries and other related information (users' details, geographical and cultural information). However, typically advertisements are sold based on click rate and, therefore, the ad network provider is only payed if the advertisement was clicked.~~ An optimal strategy is to serve advertisements based on their expected click rate value. Machine learning models are used to find the expected click rate of advertisements before they are served to the user. A predictive model is created based on the available training data, which typically includes content of the page, search query, user information, content and meta-data of the available ad creatives. Once the model is deployed, prediction queries of the form of ad requests arrive at the system. The model proposes the ad creatives that have the highest probabilities of being clicked by the user. Feedback of whether advertisements are clicked on or not is send back to the system in the form of new training observations and the model is updated accordingly. Furthermore, new training data sources in the form user databases, new ad creatives with their metadata and more web pages will become available as more companies employ the service of the ad provider. In order to leverage the new data and reduce the error introduced by incremental learning, the model is periodically retrained using entire data. Figure 1 demonstrates the CTR prediction use case for a search engine provider.

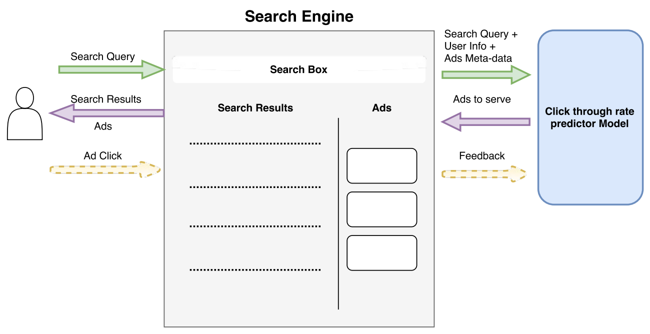

Figure 1: Use Case: Click Through Rate Prediction

Some emerging machine learning systems have tried to automate this process to some extend [7]. They automatically deploy the model into a production environment, monitor its quality, and initiate a retraining when required. However, they treat the underlying machine learning models as black boxes and as a result miss many opportunities for optimizing the training and deployment process.

Our key observation is that, by utilizing the underlying optimization algorithm (stochastic gradient descent), the model training process can be seamlessly executed along side the serving and query answering component. Based on this observation, we present a new model for a system, that supports deployment, maintenance, and incremental and fast batch updates of machine learning models. We are making the following contributions:

- a system for machine learning models deployment and maintenance
- elimination of offline batch retraining and introduction of uninterrupted updates to the model while the system is running
- incremental updates and handling of changes in data distribution

Our experiments show that we are able to achieve the same quality with similar systems while completely eliminating the offline batch training, which reduces the running time by an order of magnitude. ~~We have experimented on time dependent datasets where distribution of the data changes over time and we have shown tha~~t our method can adapt to changes in the distribution faster than the existing methods.

The rest of this paper is organized as follows. Section 2 describes the underlying optimization method ~~used in our system~~. Section 3 introduces the design principles of our system. Section 4 discusses the architecture and component ~~of the system.~~ In Section 5, we evaluate our system against different workloads and compare the performance of our method to other model serving and maintenance approaches. Section 6 discusses ~~the~~ related work. Finally, Section 7 presents our conclusion and future work.

2. STOCHASTIC GRADIENT DESCENT

In machine learning, optimization methods are used to find the minimum of the objective function (referred to as the loss function) by calculating the gradient of the function at different data points and update the function parameters based on the gradient values. A common optimization method is gradient descent, an iterative process where in every iteration the entire training data set is used to calculate the gradient value. One drawback of gradient descent is that in presence of large datasets it will become very slow since every iteration has to inspect all the training items. Stochastic gradient descent (SGD) is an approximation of the gradient descent method. Similar to gradient descent, it is an iterative process but in each iteration it operates on one element (or a sample of elements) at a time. It calculates the gradient at a single element and updates the parameters of the model accordingly. Although it converges after a higher number of iterations, the overall convergence time is lower (sometimes by orders of magnitude) than gradient descent. Each iteration of SGD can be executed in a short amount of time since it only works with a sample of the data. We are leveraging this property of SGD and design our system so that it executes one iteration of SGD at a time while the system is running. In Section 5 we show that time to execute a single iteration of SGD is insignificant. Moreover, since a single iteration is a lightweight process, it can be executed in separate threads and will not interrupt the prediction query answering component of the system.

2.1 Distributed SGD

The amount of available data has been increasing exponentially in recent years. To efficiently train machine learning models on large datasets, new techniques have to be employed. As explained earlier, SGD inherently works well with large amount of data since it does not need to scan every data point in every iteration. However, the problem arises when the dataset cannot fit into the memory of a single machine and ~~it~~ has to be distributed. In this situation, one common method is to distribute the gradient calculation tasks across the nodes in the cluster, where each task will work with a different part of the data. One problem in this scenario is that a synchronization step is required before applying the updates to the model. The synchronization step slows down the SGD process, because after every iteration, all the updates are sent to a central process that updates the model. [26, 8] proposed distributed asynchronous SGD and their experiments show that the quality of the final model is not any worse than the synchronized approach.

2.2 Learning Rate Tuning

The learning rate or step size parameter plays an important role in the convergence of SGD. The most common technique is to set the learning rate to a small value, and slowly decrease the value in every iteration. This, however is not possible in case of continuous and online learning. Adaptive learning rates have been used in online scenarios where based on different criteria the learning rate is adjusted automatically. Schaul et al. [28] have proposed one such method were the learning rate can be inferred automatically based on the changes in the weight parameters. One advantage of their method is that it works well with non-stationary problems, where the distribution of the data is constantly changing. By examining the changes in the data the learning rate will decrease (when approaching the optimum value) and increase (when the distribution of the data has changed).

2.3 Machine Learning Models based on SGD

SGD is one of the most common optimization methods for training machine learning models on large datasets. It has been used in classification [34], clustering [4], neural networks [8], and matrix factorization [9]. In our prototype, we have implemented linear classifiers and matrix factorization models. Some examples of machine learning models that use SGD are:

Linear Classifiers are arguably the most common type of machine learning models build using SGD optimization method. In our CTR prediction use case described in section 1, logistic regression method is used to train models for predicting the click through rate [23]. Logistic regression models typically output a probably ~~instead of a class label [16].~~ This probability indicates how likely an item belongs to a specific class. In our ~~CTR prediction~~ example, for every available advertisement, the click probability is predicted and depending on how many advertisements will be displayed, the ones with highest probabilities are selected. Support vector machine (SVM) [31] is another common classification model. While logistic regression model aims to find parameters of a function that accurately fit the data points to the labels, SVM tries to find a hyperplane that separates the data points belonging to different classes. In many cases, both type of models work equally well, but depending the data and its complexity one method may be preferred to the other.

Matrix Factorization is a common method used in recommender systems [17]. Matrix factorization is used to derive the latent factors (users and items) for recommender systems. It relies on the fact that each user and item can be described in a few dimensions (10 to 40 usually) based on the available ratings. These latent factors have automatically capture the similarity of users and items based on the ratings provided by the users. Any unknown rating, hence can be predicted by simply using a dot product of the user vector with item vector. In our prototype, we used the SGD implementation of matrix factorization by Simon Funk [9]. A scalable version of the algorithm was proposed by Gemulla et al. [10].

Neural Networks or deep learning inspired by biological neural networks in brain are used to learn and approximate complex functions. They have been used for more than half a century to model functions and have been used in training machine learning models. However, due to slow training process and lack of large amount of training data they have not been used extensively in machine learning community. Past decade saw a change with publication of several nominal works. Hinton et al. [15] proposed methods for speeding up the training of neural networks. ImageNet competition [27] in 2012 was won by a neural network proposed by Krizhevsky et al. [19] where they significantly reduced the error rate. Success of Google Brain team [32, 24] was also instrumental to popularizing neural networks in machine learning community.

SGD is the core run-time of our model deployment system. SGD's near constant iteration running time and its fast adoption to changes in data distribution makes it ideal for both large datasets and real-time environments. In our prototype, we are using a simple implementation of distributed SGD with synchronization and a constant learning rate. However, the SGD run-time is decoupled from the rest of the components as will be explained in section 4, therefore more robust versions of distributed SGD and learning rate adaption methods can be used in the system without affecting other components.

3. CONTINUOUS TRAINING AND SERVING

As described in Section 2, stochastic gradient descent is an iterative process where in each iteration one or a sample of the data is selected randomly and the gradient is calculated based on this sample. After each iteration the parameters of the model are updated. As more iterations are executed, the model becomes more accurate. Since SGD can train a model using both individual and (mini-)batches of data items, it is an idea optimization methods for deployment environments. Initial models can be trained over large batches data. After the initial training, these models can be deployed into a production environment where they will answer prediction queries arriving at the system. As mentioned in section 1 most systems incorporate a feedback mechanism where more training items will become available over time. With arrival of each new training item, SGD can incrementally update the model. In our CTR use case, initial training is performed over the available click behavior of the users. This data is gathered over a long period of time by observing what ads have been click by each specific users. After the model is deployed, prediction queries that contains users' and requested pages' information are send to the model. Based on this data and the properties of the available ads creatives, the model proposes the best set of ads to be shown to the user. The system collects feedback based on whether the user clicked on any of the given set of ads. If an ad is click by a user, this information is sent back to the system as a new training item which will be used to incrementally update the model. However, if an ad is not clicked by a user, it is not a strong indication of user's negative taste towards a specific ad. Therefore, training items with negative labels

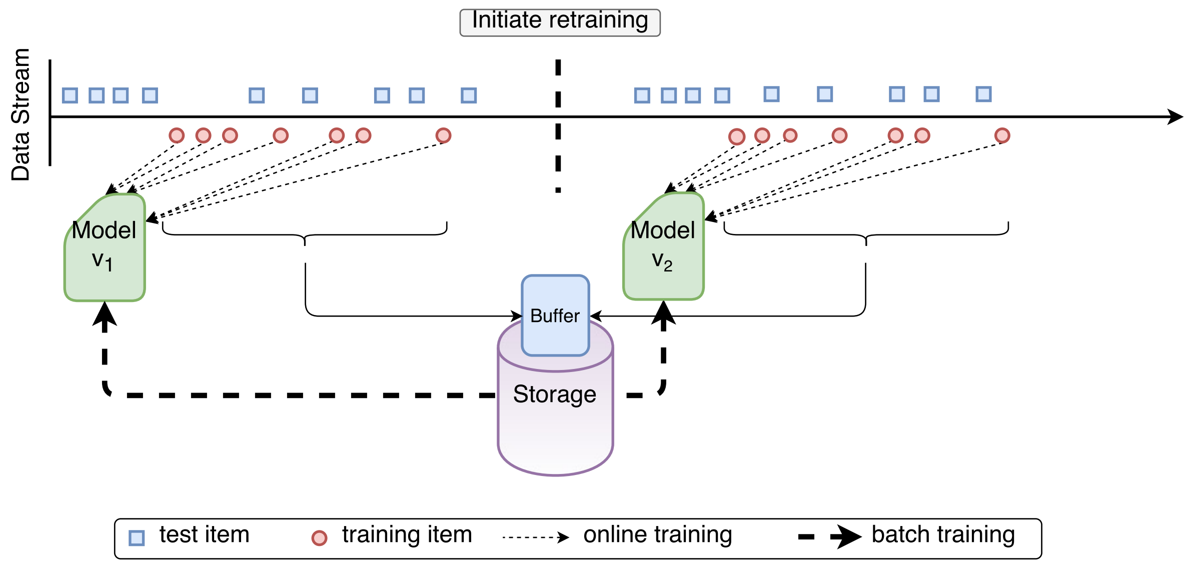

Figure 2: Velox Work Flow

are typically only generated if an ad has a very low (close to zero) click ratio for a very long period of time.

3.1 State of the Art

Velox [7] is an state of the art system which has implemented this deployment model. It uses Apache Spark to train an initial model using the available training data. Once the model is trained, it is deployed into a distributed environment, where prediction queries are answered in real time. When new training data arrives at the system the model is incrementally updated. The incoming training data is also stored in a persistent storage. To ensure the model is performing well, its quality is monitored over time. Velox periodically retrains the model from scratch. This retraining can be triggered when the size of new training data has surpasses a predefined limit, or the quality of the model has gone below a user defined threshold or simply after a fixed period of time defined by the user. Figure 2 depicts how Velox works. A buffer is used to store the incoming training observations, which will be combined with the historical data during the next retraining phase. Although Velox continuously monitors and updates the underlying model, it has 3 drawbacks:

- Complete retraining discards all the current model parameter
- Process of retraining on full dataset is a time consuming process that requires a lot of resources
- When new datasets are introduced, a complete retraining has to be performed in order to update the model

While the model is being retrained, incremental updates are turned off. Moreover, newly introduced datasets will not have any effect on the model until the next retraining. This negatively affects the quality of the predictions produced by the system. To address these problems, we introduced a new deployment system which we describe next.

3.2 Continuous Deployment

Our key observation is that, by utilizing SGD's properties, the model training process can be seamlessly executed along side the serving and query answering component. Similar to Velox, models are incrementally updated. However, unlike Velox, the retraining process is completely eliminated and instead is replaced by a series of consecutive 1-iteration of SGD optimization. Since each iteration of SGD is light and fast, it can be executed while the system is serving prediction requests.

Figure 3 shows how our proposed system works. An initial model is trained using the data residing on disk. Once the model is deployed, it receives prediction queries and training observations in a streaming fashion. Incoming prediction queries are answered as soon as they are received. Once a training observation is received the model is incrementally updated. The system also keeps track of incoming training observations and adds them to an intermediate buffer. A scheduler component, will schedule a new iteration of SGD based on the rate of incoming training observations. The scheduler can also decide to run an iteration if the system is idle or the load on the system is not heavy. Each new iteration will use a random sample of the data in storage and the data in the buffer. As more training observations becomes available the model is updated further. Moreover, new datasets can be stored in the buffer (or persistent storage unit) as soon as they become available, and used immediately to further train the model without requiring a new model to be retrained from scratch. In our deployment system, we are addressing the issues with current methods have. First of all, the models are never discarded and new iterations of SGD are applied directly to the current version of the model. The retraining process is replaced by a series of independent SGD iterations. Therefore, incremental updates are not turned off and effect of new datasets are visible in the model within a small time frame.

4. SYSTEM ARCHITECTURE

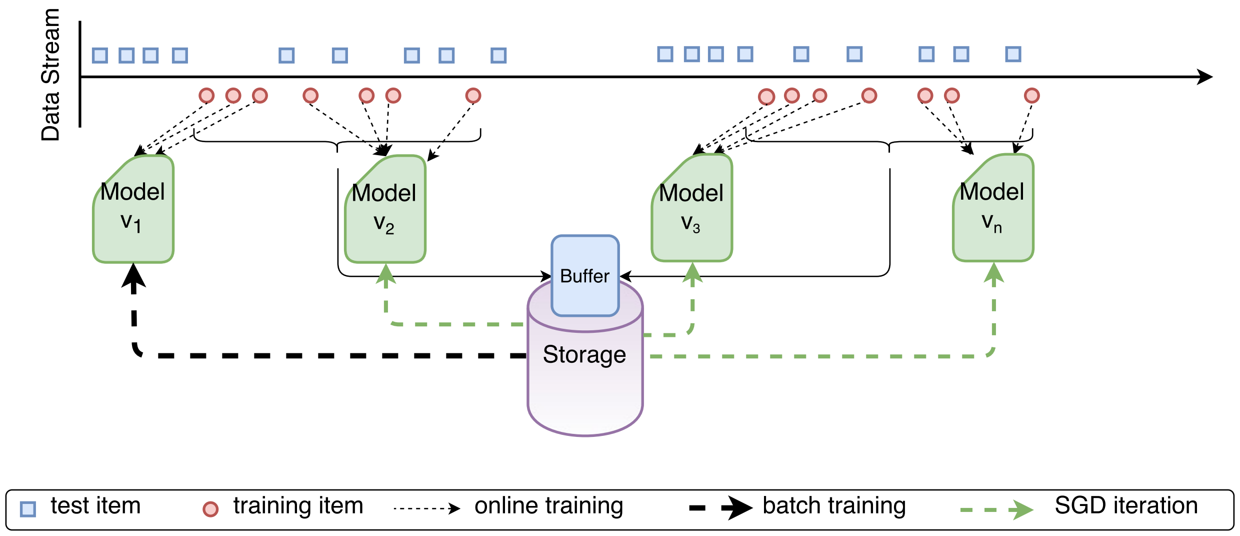

Figure 3: Continuous Training and Serving

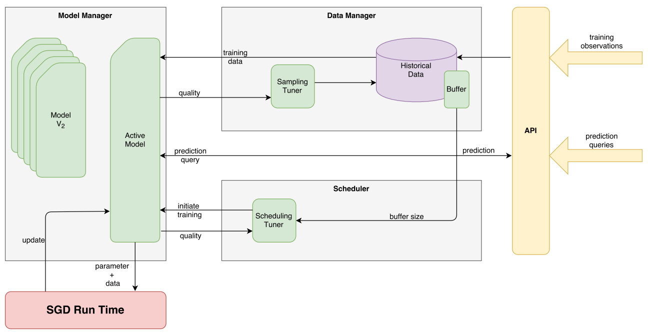

Figure 4: System Architecture

The proposed system comprises three main component; model manager, data manager and scheduler and an independent SGD run-time. Figure 4 shows an overview of the architecture of our system and the interactions between its components. Incoming training observations are forwarded to the data manager where they first stored in a buffer and then passed on to the model manager in order to incrementally update the model. Prediction requests are directly forwarded to the model manager, where based on the latest available model a prediction is made and the result is returned to the to the user. Both scheduler and data manager components are constantly communicating with each other and the model manager to obtain the latest statistics such as the model quality and buffer size, which in turn helps in tuning the scheduling and sampling rate for next iterations of SGD. Next, we explain each component of the system in more detail.

4.1 Scheduler

Scheduler component is responsible for scheduling of new iterations of SGD. An iteration of SGD can be scheduled at any point in time. Intuitively, the best time to execute an iteration is when the system is not under heavy load. This will help in utilizing the system's resources as well as keeping the model up to date at all time. ~~Another situation that~~ a new iteration of SGD should be executed ~~is~~ when the system is receiving a lot of training items and the intermediate buffer is getting full. If the model is not updated with the new training items frequently, the quality will go down more rapidly, especially if the distribution of the data is changing. Training iterations and model serving are disjoint and not affected by each other. As soon as the training iteration is completed the model is updated. This property allows the scheduler to execute new iteration of SGD in a separate thread without interrupting the serving module. The scheduling rate however affects the quality of the model. Each new iteration slightly increases the quality of the model. The increase in quality is even more when new training observations have arrived at the system.

In Section 5, we investigate the effect of scheduling rate on model quality. In case no new training data is available, the model parameters will eventually converge and any further training iterations will not have any effect on the model quality. Therefore, the scheduler component has to communicate with the model manager in order to detect whether the model parameters have converged and stop further iterations until more training data becomes available.

4.2 Data Manager

In order to execute an iteration of SGD, we need to combine the training data that arrives at the system in real-time and the data stored on disk. The data manager is responsible for storing the incoming training observations in an intermediate buffer. When a new training iteration is scheduled, the data manager accesses the historical data stored on disk and provides a sample. The data from the sample and the data in the buffer are merged together to create the dataset for next training iteration. The data manager provides access to this dataset for model manager where the actual training and model updates happen. The data manager also communicates with scheduler in order to inform it when the intermediate buffer is becoming full and a new training iteration is required.

The created data set consists of the data inside the buffer and a sample of the historical data as described earlier. The sampling rate, therefore, is a parameter that has to be configured. It can be pre-configured to a constant value based on the application. However, using the feedback from the system's model manager (Section 4.3), sampling rate can be adjusted. For example, when the data distribution is changing more rapidly a smaller sampling rate will put more emphasis on the data that arrived in the system recently. This is similar to the problem of concept drift where the distribution of the incoming data changes overtime. This will render historical data less important and as a result a smaller sample of the historical data (or none at all) will give more importance to the data in the buffer and help the model to adopt faster to the concept drift. However, if there is no concept drift in the data, a larger sampling rate will increase the quality of the model after a training iteration. Another effect that the sampling rate has on the system is the training iteration running time. Larger sampling rate increases the running time of each training iteration as more data has to be processed. In Section 5, we investigate the effects of different sampling rates on both the quality and performance of the system.

Moreover, new data sets can be registered in data manager. In our current prototype, new data sets first have to be stored on disk, and data manager should be informed of the data path. Newly available data sets are used in the subsequent SGD iterations.

4.3 Model Manager

An important part of the system is the model manager component. It is responsible for storing the model, answering prediction queries and performing incremental and batch updates to the model. Listing 1 shows the APIs of the model manager. They are used to both interact with other components as well as end-users of the system. The scheduler component uses *update* and *update_iteration* to instruct the model manager to perform incremental or batch updates (one iteration of SGD) to the model. Upon a new prediction query, the *predict* method is called to provide the end-user with the label of the given input.

Listing 1: Model Manager API

```
def update(x,y)

def update_iteration(X,Y)

def predict(x): Label

def error_rate(X_test, Y_test): Double
```

The *error_rate* method returns the error rate of the model using the provided test dataset. As described earlier, constant monitoring of the quality is required in order to adjust the scheduling and sampling rate. When the error rate is stagnating, this could mean that the model has converged using the existing data, therefore, the model manager informs scheduler not to schedule any new iterations until new training observations have arrived at the system. Similarly as explained in Section 4.2, an increase in the error rate could indicate a change in distribution of the data. As a result, reducing the sampling rate will put more emphasis on the recent data (in the intermediate buffer) and help adopt the model to the changes in the distribution.

The model manager also keeps track of the changes that are made to the model. The model is updated both through incremental learning and training iteration. The model manager creates snapshots of the model in two different scenarios; after a series of incremental updates are made and after each training iteration. This versioning of the models is essential. When there is a rapid change in the distribution of the incoming data (a sudden concept drift) or when there are anomalies in the data, it is sometimes necessary to revert back to a version before the change in distribution occurred. In case of concept drift, new training iterations should be scheduled that only use the data in the buffer and in case of an anomalies in the data, they have to be identified and discarded before any further model updates could happen.

4.4 SGD Run-Time

All components of our model serving system described so far are not limited to any specific run time. We have decoupled the components from the actual run time of the system. As described earlier the underlying optimization method is SGD and any run time capable of performing incremental and batch SGD updates efficiently are suitable options for our system. Apache Flink [5] and Apache Spark [33] are distributed data processing platform that work with data in memory and have support for iterative algorithms, which makes both of them ideal options for our SGD run-time. In our current prototype, we are using Apache Spark [33] as our SGD run-time, but we plan incorporate Apache Flink in the complete version of the system. Model manager is the component responsible for communicating with the SGD run-time. In the current version of our prototype, model manager requests Spark to perform both incremental and batch updates to the model, both of which are supported by the built in machine learning library of Spark. The choice of run-time for SGD slightly influences the data manager as well. In our prototype, historical data is stored on Hadoop Distributed File System (HDFS) [29]. Therefore, the data manager should have support for HDFS. Both Flink and Spark provide out-of-the-box support for HDFS.

5. EVALUATION

In this section, we evaluate the performance of our system using various datasets. We report both the quality (error rate) and performance of our proposed method.

5.1 Experimental setup

We run our experiments on a single node with a quad-core Intel i7 and 16 GB of RAM. To evaluate our proposed method, we are using 2 different datasets.

MovieLens [13]: to analyze the performance of our method on time varying datasets, we use the MovieLens 100K and 1M datasets. They consists of movie ratings by users during different time periods. For both of the datasets, we use 10% for initial learning and 90% for online learning. We have set the learning rate to 0.001 for all of the experiments on MovieLens dataset. We have sorted the MovieLens data set based on the ratings timestamp and items are examined one by one according to their timestamp. This is to evaluate how each of the implemented methods react to changes in the distribution of the data. An evolving test set was also created, where the index of the test items are first drawn uniformly at random from the entire data set. For *movie-lens-100k*, the total size of the test set is 5,000 and for *movie-lens-1M* it is 50,000. We start with an empty test set, when we reach the index of a test item, we add that to the current test set and calculate the error rate based on the current test set. Each error calculation (called a test cycle), as a result captures the degree to which the models are adopting to the changes in the dataset.

MNIST [21]: MNIST is a dataset of handwritten digits. it consists of 60000 28x28 color images in 10 classes. We use a neural network to train a model to classify the images in the data set. Similar to MovieLens, 10% are used for initial training and 90% for online learning. Note that, in this dataset, the order is irrelevant and we assume no change in distribution of the data occurs throughout the system's lifetime.

5.2 Methods

In this section, we briefly describe the methods we have implemented.

Baseline is the naive and simplest deployment model. After a model is a trained from the initial data, it is used throughout the lifetime of the application without any incremental or batch learning.

Baseline+ is similar to the baseline approach with the added incremental learning. After the initial model is deployed, it is constantly updated based on new training items that arrive at the system.

Velox is an implementation of the common deployment scenario described in Section 1 It is based on the the proposed system by Crankshaw [7]. After the initial model is deployed, the model is incrementally updated with every new training item. A full retraining of the model is performed once a certain amount of training observations have been received.

Continuous is the implementation of our proposed method. Similar to Velox and Baseline+ once, an initial model is first trained using the existing data. New training data is used to incrementally update the model. Based on the rate of incoming training data, a scheduler component triggers new iteration of SGD. The data used in this new iteration of SGD consists of the new data that arrived in the system since the last scheduled iteration plus a sample of the existing historical data.

Static training is a simple training of a static model over the entire dataset. Static training is only used to establish a baseline for comparison of the running time of different methods. It is only applicable in cases where the entire dataset is available.

5.3 Implementation of ML Models

We have implemented two different machine learning models to evaluate our system. In this section, we briefly explain each implemented methods.

Recommender system: we have implemented a recommender system based on the matrix factorization method described in Section 2. Based on our initial experiments, we have set the number of latent factors to 40. A model is first trained from the static data. To produce more accurate predictions, we have also included user, item, and global bias values as described in [17]. After the initial model is trained and deployed, any new training observations incrementally update the model. Training observations are of the form $(user\_id, item\_id, rating)$. Based on the training observation, the bias values as well as the factors are updated for the specified user and item.

Neural Network: to evaluate our system on an image classification task, we implemented a multi-layer perceptron neural network using back propagation [6]. We set the number of hidden layers to 50 and use a softmax output function [3]. Similar to recommender system case, a network is first trained on the static data. The training observations are of the form of (X, y), where X is a vector of 784 dimension (1 for each pixel) and y is the digit the image is representing.

5.4 Tuning parameters

In section 3, we discussed how system parameters such as sampling and scheduling rate affect the performance and quality of the system. In this section, we are analyzing the effects of different values of sampling and scheduling rate on the system. Our goal, is to make these parameters adaptive, but for now we analyze their effect on the running time and mean squared error.

Scheduling rate: This parameter specifies how often a new iteration of SGD should be scheduled. In our prototype, the scheduling rate is governed by a parameter called buffer size, which dictates how many new items should be stored in the buffer before a new iteration of SGD is executed. Ultimately, the decision to schedule new iterations is made by the system based on the availability of resources. Executing one iteration of SGD even using the entire data is not a resource heavy process, and can easily be done in parallel with the serving component of the system. This results in a paradigm where both training and serving can happen simultaneously.

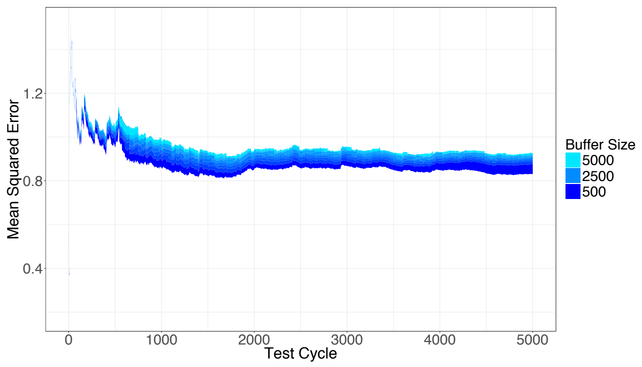

Figure 5: Movie Lens 100k buffer size vs mean squared error

Figure 5 shows the mean squared error for different buffer sizes for *movie-lens-100k* dataset. An smaller buffer size causes the scheduler to initiate training iterations more frequently. As a result, the underlying model is updated faster. However, the error rate is not decreasing linearly with the buffer size. Further analysis show that once the model is update more frequently, it slowly starts to converge and any further training has little to no effect on the overall quality of the model Therefore, increasing the scheduling rate only decreases the error rate up to a point, after which increasing the scheduling rate has no effect on the overall quality of the model. This is extremely important, specially when considering the effect the buffer size has on the running time. Figure 6 shows the running time on *movie-lens-100k* dataset using different buffer sizes. Increasing the buffer size from 500 to 5000 decreases the running time by a factor of 5 while as described before the error rate is only increased slightly. Therefore, depending on the application, we can set the buffer size to bigger values in order to increase the performance of the system without affecting the quality of the final model substantially.

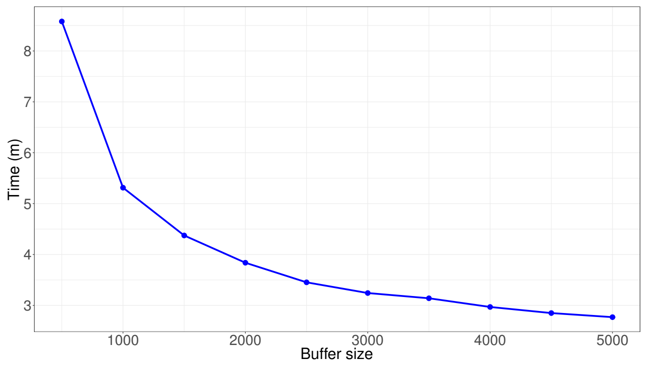

Figure 6: Movie Lens 100k buffer size vs time

Dynamic scheduling: In production environments, the load on the system varies throughout the day. Therefore, a dynamic scheduling maximizes the performance of the system, by performing more frequent updates while there are more resources available for training. Moreover, since training and serving can be done in parallel, we can perform training in the background and only update the weights when the training iteration is over.

Sampling rate: In each iteration of SGD, as described in Section 3, the data inside the buffer and a sample of the historical data is used to update the model. In this section, we investigate the effect of the sampling rate on the model quality and running time of the system. Figure 7 shows that larger sampling rate increases the quality of the model, but similar to scheduling rate, the decrease in error rate is negligible considering the effect it has on running time. This is again, caused by the same phenomena, where the model after training on bigger sample rates start to converge faster and as a result bigger sample sizes will not have an effect on the quality.

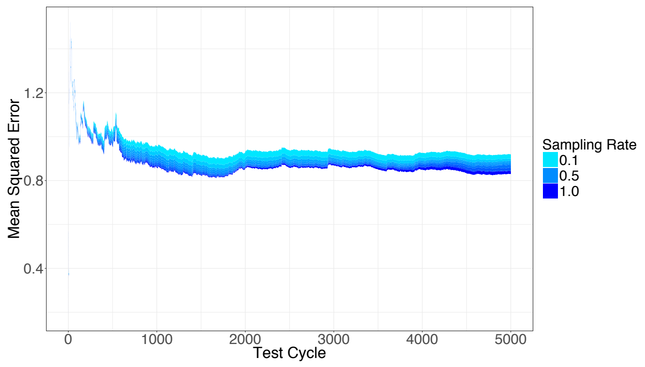

Figure 7: Movie Lens 100k sample rate vs mean squared error

Figure 8 shows that effect of increasing sampling rate on running time. Using the entire historical data (sampling rate = 1.0) increases the running time 5 fold. Therefore, similar to scheduling rate, setting the sampling rate to smaller values will increase the performance substantially, while only slightly affecting the quality of the model.

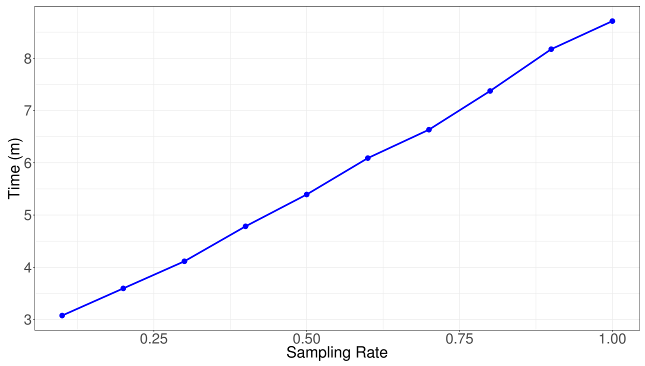

Figure 8: Movie Lens 100k sampling rate vs time

Tuning parameters based on error rate: As described earlier, tuning the parameters are heavily influenced by application type. Both the underlying machine learning model and the dataset can have a big effect on the selection of sampling rate and scheduling rate. In the recommender system use case, due to changes in incoming data distribution, we saw that bigger sample rates and higher scheduling rates have an effect (although small) on the quality of the model. This, however, may not be the case for other use cases. To demonstrate this, we perform the same set of experiments on the *MNIST* dataset. Figure 9 shows the effect of different sampling rates on the neural network model for *MNIST* data. Interestingly and contrary to the results we achieved for *movie-lens-100k* the difference in error rates for different sampling rates are almost indistinguishable from each other. While the difference in error rates for *movie-lens-100k* was small, but it is still much greater than the difference in error rates for neural networks. We believe this is caused by how neural networks behave. Increasing sampling rate, causes similar data items to be used repeatedly in each training iteration and based on our experiments neural networks are not affected by this oversampling and, therefore, the results are almost similar with different sampling rates. Moreover, in this experiment, the number of parameters of multi-layer perceptron is far less than the number of parameters of the matrix factorization model for *movie-lens-100k*. This causes the neural work to converge faster and, therefore, it is not affected by more training, unless new training observations arrive at the system.

Figure 10 shows how buffer size affects the overall quality of the neural network models achieved. Similar to sampling rate case, the error rates of the models are not affected by the scheduling rate. New training observations that exist in the buffer have the maximum effect on the model's quality, since they are becoming available to the model for the first time. As the scheduling rate increases, the number of new training observations remain the same, and only the historical data is used more frequently to train the model. Since neural networks do not gain much benefit by revisiting the same items, increasing scheduling rate has no effect on the overall quality.

Based on our findings, we conclude that increasing the sampling and scheduling rate does not always have an effect

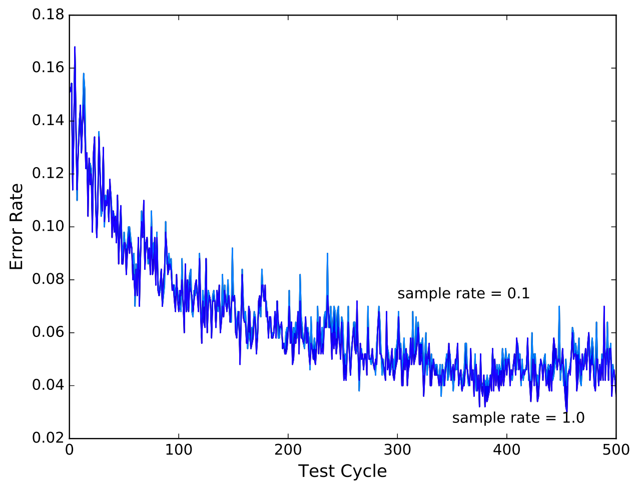

Figure 9: MNIST sample rate vs error rate

on the quality. In both the *movie-lens-100k* and *MNIST* use cases the change in scheduling and sampling rate has small to no effect on the overall quality. However, the running time of the methods are heavily influenced by these parameters. Setting these parameters to small values decreases the running time considerably and save computation resources regardless of the type of model the system is serving.

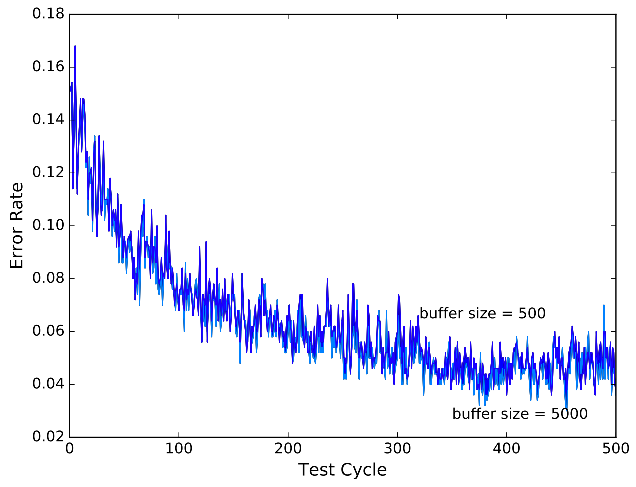

Figure 10: MNIST buffer size vs error rate

5.5 Comparison Under Constrained Resources

Both, the velox and the continuous model use different strategies to update the model. To perform a fair comparison of the two, we are evaluating them under similar resource constraints. We configured both of the methods' frequency of training in such a way that the total running time on the *movie-lens* and *MNIST* datasets are similar. To achieve this, we have fixed the total number of epochs (SGD iterations) for both methods. For velox, this value is equal to the number of retraining multiplied by number of iterations in each retraining. For the continuous model, it is total number of scheduled iterations throughout the lifetime of the system. Therefore, the continuous method schedules a new iteration every 500 items for *movie-lens-100k* and 5,000 items for *movie-lens-1M* and velox method schedules a retraining every 15,000 items for *movie-lens-100k* and 150,000 items for *movie-lens-1M*. Figure 11 shows the

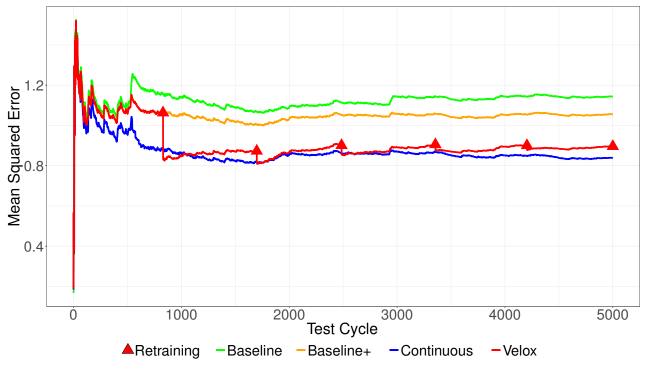

Figure 11: Mean squared error on MovieLens 100k

mean squared error of the continuous, velox, and several other methods on *movie-lens-100k* dataset. Baseline model achieves the worst performance among all the existing methods. Baseline model's quality is similar to Baseline+, Continuous, and Velox methods in the beginning. Since all four methods perform an initial retraining, the error rate stays low in the beginning, but Baseline quality starts to degrade since it does not include any incremental or batch retraining. Baseline+ continuous to perform better than naive method until 3,000 test cycles. However, afterwards a small change in distribution of the data causes the model trained using Baseline method to perform worst since it cannot adopt to changes in the data and the quality continues to degrade as more new test items are examined. Velox method performs similar to Baseline+ method until the first retraining is executed. There is a sudden drop in the error rate after the first retraining. For the remainder of the data points, Velox performs as expected, once there is a retraining, the error rate immediately decreases. This decrease in error rate, however, is followed by a slow increase due the fact that only incremental updates are being made to the model. Finally, Continuous method has the lowest error rate among all the implemented methods. It consistently manages to adopt to the changes and decrease the error rate where the distribution of the incoming data is stable. As expected, except for immediately after a full retraining, the Continuous method always performs better than Velox.

Figure 12 shows the mean squared error rate achieved by the implemented methods on *movie-lens-1M*. Similar to *movie-lens-100k*, the Continuous method has the lowest error rate among all the implemented methods. The difference in error rate between Continuous and Velox is even greater than the error rate for *movie-lens-100k*. We believe a bigger shift in data distribution (data in *movie-lens-1M* is gathered from a longer period of time) makes the Continuous method to consistently adopt faster to the changes in data distribution. Velox's error rate follows the same trend as in the case of *movie-lens-100k*. After every retraining, the error rate first drops then slowly increases until the next retraining. Similar to *movie-lens-100k* dataset, the error rate of Baseline is highest among all the implemented methods, followed

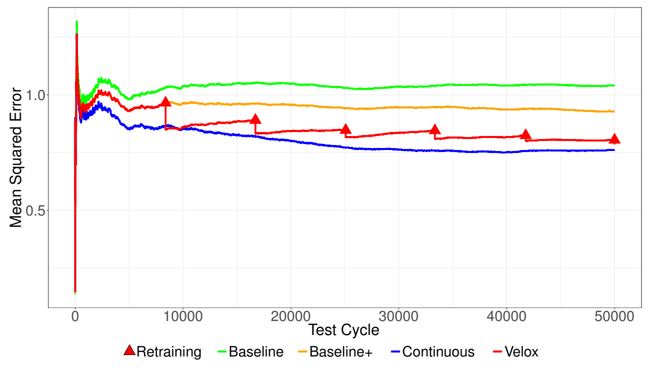

Figure 12: Mean squared error on Movie Lens 1M

by Baseline+ , where due to the incremental updates of the model the error rate is consistently lower.

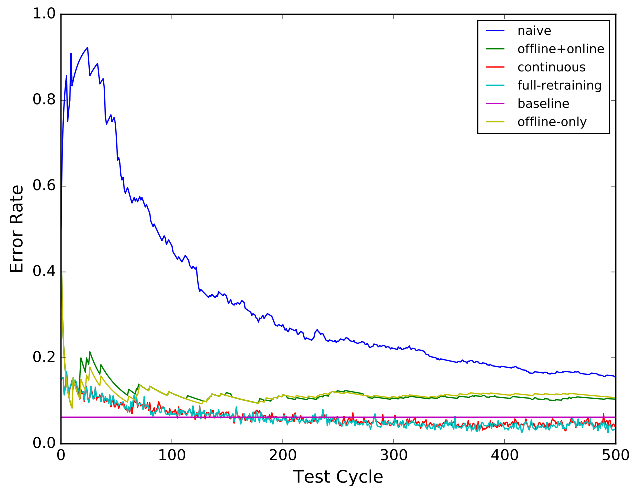

Figure 13: Error rate on MNIST

Figure 13 shows the error rate (misclassification rate) of the implemented methods on *MNIST* dataset. For this workload, the implemented methods behave quite differently from previous experiments and that is due to dataset characteristics and the underling model properties. In order to achieve a high accuracy, neural networks require a lot of training data. This explains why the naive method performs poorly. Its error rate starts to decrease as more training data becomes available, but it is still far inferior to all the other methods. The gap in error rate between offline+online and offline-only methods is much smaller than that of movie lens datasets. Unlike the movie lens datasets, *MNIST* does not contain a shift in distribution and that is why both the offline+online and offline-only methods' performance is rather similar. The Baseline+ method performs slightly better because of the extra training items that the underlying neural network is exposed to. Velox and Continuous methods behave similarly as well and unlike Movie Lens datasets, Velox has a consistent error rate, that is the error rate does not decrease after a retraining of the model. Similar to offline+online's case, since there is no change in the distribution of the data, the error rate does not degrade after a retraining. The continuous method manages to decrease the error rate as more training iterations are performed. As more training iterations are executed, the underlying neural network's parameters converge and as a result the error rate on the test set decreases. Both Continuous and Velox start to perform better than the baseline after a few training iterations (retraining in Velox's case). This is, as explained earlier because of neural network's ability to perform better when more training data are available to it.

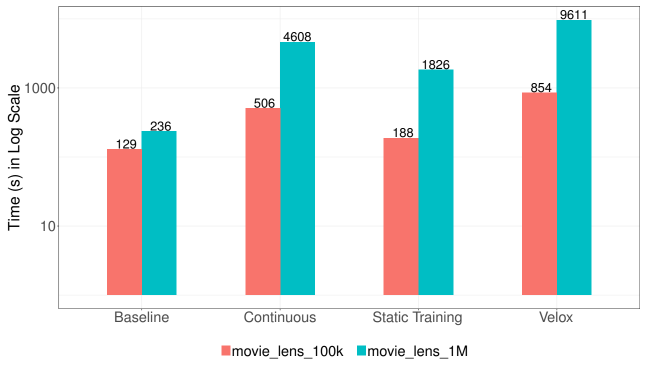

Figure 14: Running time on different Workloads

Figure 14 shows the running time of the implemented methods on *movie-lens-100k*, *movie-lens-1M* and *MNIST*. As expected, Baseline have the lowest running times on both datasets. This is because they do not support batch updates to the model. Static training method's running time is only affected by data size. When dataset size big, each iteration of SGD will examine more data and the model will converge slower as there are a lot more data to examine. Continuous method's running time is smaller than Velox by almost an order of magnitude in all workloads. Although we have configured both methods to use similar amount of resources (same number of iterations), continuous method still manages to achieve a consistent and lower error rate in a much faster time. This is because a full retraining incurs a much higher overhead than continuously training the method. Another reason for the difference in running time is that as the size of the data increases the time for running a new iteration in continuous method stays roughly the same where as the time for a full retraining increases exponentially. The running time of the Baseline+ is not included as it is similar to Baseline, since they both have include initial training and have to examine individual items arriving at the system one at a time.

6. RELATED WORK

Traditional machine learning systems focus on training and management of models and leave the task of deployment and maintenance to the users. It has only been recently that some systems, for example Velox [7], TensorFlow Serving [1], and LongView [2] have proposed architectures to support model deployment and query answering as well. 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, LongView only works with batch data and does not provide support for realtime queries. As a result it does not support incremental learning neither. Our system supports both realtime and incremental learning. TensorFlow Serving provides mechanisms for realtime queries, deployment and version control of machine learning models. It has out-of-the-box support for models created using TensorFlow and it provides several interfaces for users to deploy their custom models. However, it does not provide incremental updates to the model. Contrary to our system, models have to be retrained outside of the system and redeployed to TensorFlow Serving once the training is done. Our system supports, incremental and batch updates to the model and automatically applies these updates to the model currently being served. Velox [7] is an implementation of the common machine learning serving practice, explained in section 1. Velox supports incremental learning and can answer prediction queries in realtime. It also eliminates the need for users to manually retrain the model offline and redeploy it again. It is integrated into the *Berkeley Data Analytics Stack (BDAS)*, and uses the components of BDAS such as Tachyon [22] and Spark [33]. After the initial training of a model, it is deployed to Velox where prediction queries are answered in realtime and incremental updates are made to the model. Velox also monitors the quality of the model using a validation set and once the error rate has gone beyond a predefined threshold it initiates a complete retraining of the model using Spark. Figure 2 depicts how Velox works. Velox uses Stochastic Gradient Descent as its underlying training algorithm. As described in Section 2, even the batch updates to the model can be performed in a series of independent iterations. However, Velox retrains the model offline from scratch. This has two drawbacks; any updates that has been done to the model so far will be discarded, and the process of retraining on full data set requires a lot of resources. Our system uses the underlying properties of SGD to fully integrate the training process into the system's lifeline and eliminate the need for complete retraining of the model.

Weka [12], Apache Mahout [25], and Madlib [14] are systems that provide the necessary toolkits to train machine learning models. All of these systems provide a range of machine learning training algorithms. They, however, do not provide any management, before or after deployment of these models. Our proposed system focuses on models trainable using Stochastic Gradient Descent and as a result is able to provide management of the models both in training and after deployment. MLBase [18], and TuPaq [30] are machine learning 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. However, once models are created, they have to be deployed and used for serving manually by the users. Our system, on the contrary, is designed for automatic deployment and continuous training.

7. CONCLUSIONS

This paper presents an architecture for deployment and continuous training of machine learning models. The architecture utilizes the properties of stochastic gradient descent optimization method. SGD is an iterative optimization process where in each iteration a small sample of the data is used to update the machine learning model, ~~however, it typically works with static datasets~~. Our proposed architecture uses the properties of ~~the~~ SGD and applies it to long running, integrated training and deployment processes. Every successive iteration of SGD is scheduled to run while the machine learning model is being used to answer prediction queries. After every iteration, the model is updated with the new parameters. SGD is one of the most common optimization techniques for large datasets and has been successfully used in different domains such as neural networks, recommender systems, regression, and clustering models. This makes our architecture applicable for a large range of machine learning models beyond the ones demonstrated in the evaluation.

To demonstrate our system's ability to incorporate different machine learning models and workloads, we implement two use cases: recommender system and Image classifier. The recommender system is tested on different workloads and the results show that not only it scales well but also it adopts to changes in data distribution by continuously updating the model. Image classifier use case as well shows how we can implement and use neural networks within our system to create a model that not only can answer prediction requests quickly, but is also able to make incremental and batch updates to further increase the model quality without creating much overhead. Our experiments also shows the continuous training of the machine learning models is much faster than full retraining of them, which is the most common approach in deployment and maintenance of machine learning models. We showed that not only continuous training requires less resources but it also produces models with lower error rates that adopt faster to changes in the dataset. Moreover, comparing to simple techniques such incremental learning or initial batch training it produces a model with higher quality.

In future work, we will explore other optimization strategies such as batching of prediction queries, caching of query results, and more advanced methods for sampling of historical data in order to investigate their effect on the performance of the system.

8. REFERENCES

[1] Martın Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, et al. Tensorflow: Large-scale machine learning on heterogeneous distributed systems. *arXiv preprint arXiv:1603.04467*, 2016.

[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] Christopher M Bishop. Pattern recognition. *Machine Learning*, 128, 2006.

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

[5] Paris Carbone, Stephan Ewen, Seif Haridi, Asterios Katsifodimos, Volker Markl, and Kostas Tzoumas. Apache flink: Stream and batch processing in a single engine. *Data Engineering*, page 28, 2015.

[6] Ronan Collobert and Samy Bengio. Links between perceptrons, mlps and svms. In *Proceedings of the twenty-first international conference on Machine learning*, page 23. ACM, 2004.

[7] 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.

[8] 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.

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

[10] Rainer Gemulla, Erik Nijkamp, Peter J Haas, and Yannis Sismanis. Large-scale matrix factorization with distributed stochastic gradient descent. In *Proceedings of the 17th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 69–77. ACM, 2011.

[11] Thore Graepel, Joaquin Q Candela, Thomas Borchert, and Ralf Herbrich. Web-scale bayesian click-through rate prediction for sponsored search advertising in microsoft's bing search engine. In *Proceedings of the 27th international conference on machine learning (ICML-10)*, pages 13–20, 2010.

[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] F Maxwell Harper and Joseph A Konstan. The movielens datasets: History and context. *ACM Transactions on Interactive Intelligent Systems (TiiS)*, 5(4):19, 2016.

[14] 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.

[15] Geoffrey E Hinton, Simon Osindero, and Yee-Whye Teh. A fast learning algorithm for deep belief nets. *Neural computation*, 18(7):1527–1554, 2006.

[16] David W Hosmer Jr, Stanley Lemeshow, and Rodney X Sturdivant. *Applied logistic regression*, volume 398. John Wiley & Sons, 2013.

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

[18] 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.

[19] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. In *Advances in neural information processing systems*, pages 1097–1105, 2012.

[20] Arun Kumar, Robert McCann, Jeffrey Naughton, Jignesh M Patel, Timothy E Babros, Randall James Hunt, Kathryn Koski, John C Strikwerda, Bruce A Wade, Robert Bruce Arnold, et al. A survey of the existing landscape of ml systems. *UW-Madison CS Tech. Rep. TR1827*, 2015.

[21] Yann LeCun and Corinna Cortes. MNIST handwritten digit database. 2010.

[22] Haoyuan Li, Ali Ghodsi, Matei Zaharia, Scott Shenker, and Ion Stoica. Tachyon: Reliable, memory speed storage for cluster computing frameworks. In *Proceedings of the ACM Symposium on Cloud Computing*, pages 1–15. ACM, 2014.

[23] 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.

[24] Tomas Mikolov, Kai Chen, Greg Corrado, and Jeffrey Dean. Efficient estimation of word representations in vector space. *arXiv preprint arXiv:1301.3781*, 2013.

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

[26] Benjamin Recht, Christopher Re, Stephen Wright, and Feng Niu. Hogwild: A lock-free approach to parallelizing stochastic gradient descent. In *Advances in Neural Information Processing Systems*, pages 693–701, 2011.

[27] Olga Russakovsky, Jia Deng, Hao Su, Jonathan Krause, Sanjeev Satheesh, Sean Ma, Zhiheng Huang, Andrej Karpathy, Aditya Khosla, Michael Bernstein, Alexander C. Berg, and Li Fei-Fei. ImageNet Large Scale Visual Recognition Challenge. *International Journal of Computer Vision (IJCV)*, 115(3):211–252, 2015.

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

[29] 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.

[30] 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.

[31] Ingo Steinwart and Andreas Christmann. *Support vector machines*. Springer Science & Business Media, 2008.

[32] Ilya Sutskever, Oriol Vinyals, and Quoc V Le. Sequence to sequence learning with neural networks. In *Advances in neural information processing systems*, pages 3104–3112, 2014.

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

[34] 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.