Continuous Training of Large Scale Machine Learning Models

Behrouz Derakhshan[1], Tilmann Rabl[1,2], and Volker Markl[1,2]

[1]DFKI, Germany
firstname.lastname@dfki.de

[2]TU Berlin, Germany
firstname.lastname@tu-berlin.de

ABSTRACT

Many scientific and business applications rely on methods from machine learning in order to derive novel insight from large data sets. A data analytics process is a pipeline comprised of several steps, from source selection, data preparation, feature engineering, model building/training, to the deployment of the model in a production environment. Many of these steps are not automated, but require manual configuration by a data scientist or are semi-automatic at best. Once a machine learning model is trained on a dataset it is deployed into a system where it can answer prediction queries in a real-time and reliable fashion. While the system is running, new training data may become available, and the system incrementally updates the model. Furthermore, to ensure higher quality of prediction and better adaptation to changes in data distribution, the system also periodically retrain the model. However, retraining of models is 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, SGD updates the model based on a set of training items. Using this property, we eliminate the need for complete retraining by replacing it with a series of consecutive SGD iterations. We show that individual iterations of SGD are light-weight and can be executed while the system is answering prediction queries. Our experiments show that our deployment method updates models an order of magnitude faster, without degrading the quality of the model. Furthermore, models adopt faster to changes in data distribution using our deployment method.

Keywords

Machine Learning Model Management; Stochastic Gradient Descent; Machine Learning Systems

1. INTRODUCTION

Deploying and maintaining models is a crucial step in the lifecycle of a machine learning (ML) application. The deployment and serving of ML models in production has received very little attention by the research community despite the fact that it is the aspect that delivers the actual (business) value. In order to sustain the performance of a model in a dynamic environment, where data and thus models may change, we have to monitor the performance of the model in real-time during its deployment and update the model when necessary.

However, incremental updates alone are not enough to maintain model quality. When the data distribution changes quickly, the model will not adjust in a timely fashion. Furthermore, new batch datasets from external sources may become available while the system is running. Lastly, incrementally updating may not arrive at the same quality of model as a complete batch retraining. Therefore, 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. [24] provided an overview of landscape of existing machine learning systems. Most of the surveyed systems provide little to no support after the model has been trained. Only few systems [2, 11] support deploying and maintaining the models. Moreover, systems with support for model deployment are still lacking constant monitoring and fast and accurate updates of the machine learning models. Deployment and training of models is usually a manual process.

Figure 1 demonstrates the most common model deployment approach. 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 system updates the model incrementally. Not all of machine learning models support incremental updates. The ability to perform incremental updates is determined by the underlying optimization strategy and whether it allows incremental changes to the parameter of the model based on individual training items.

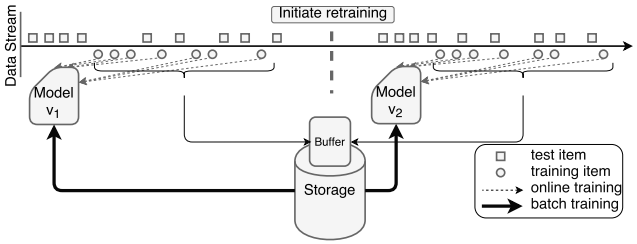


Figure 1: Current Model Deployment Method

Example application: to illustrate this model deployment approach, consider the task of predicting the click through rate (CTR) of online advertisements. Search engine providers use machine learning models to estimate the expected click rate of different advertisements. They train machine learning models based on the available data, which typically includes content of the page, search query, user information, content and meta-data of the available advertisements. Once the model is deployed, prediction queries of the form of ad requests arrive at the system. The model determines the advertisements that have the highest probabilities of being clicked by the user. Based on whether or not advertisements are clicked on, the system sends feedback in the form of new training observations and the model is updated accordingly. Furthermore, new batch training datasets in the form of user databases, new advertisements with their metadata and more web pages will become available as more companies employ the service of the search engine provider. In order to leverage the new data and reduce the error introduced by incremental learning, the model is periodically retrained using the entire data. Figure 2 demonstrates the CTR prediction example for a search engine provider.

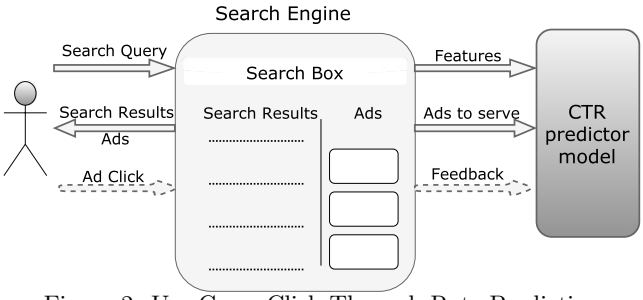


Figure 2: Use Case: Click Through Rate Prediction

Some emerging machine learning systems have tried to automate this process [11]. 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. As a result, they miss many opportunities for optimizing the training and deployment process.

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

- We define a system for machine learning models deployment and maintenance and provide a prototypical implementation
- We allow for incremental updates to the model, thus handling changes in the data distribution
- We eliminate offline batch retraining and replace it with a series of single iteration of SGD

Our experiments show that our approach improves the runtime of model training and deployment by an order of magnitude over the state of the art while retaining the quality of the model. The speedup is due to the fact that we are able to completely eliminate batch retraining. Our method can also 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. Section 3 introduces the design principles of our system. Section 4 discusses the architecture and components. In Section 5, we evaluate our system against different workloads and compare the performance of our method to other model deployment and maintenance approaches. Section 6 discusses related work. Finally, Section 7 presents our conclusion and future work.

2. STOCHASTIC GRADIENT DESCENT

Machine learning applies optimization methods in order to find the minimum of an 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 because every iteration has to inspect all the training items. Stochastic gradient descent [5] 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 because it only works with a sample of the data. We are leveraging this property of SGD and design our deployment system so that it executes one iteration of SGD at a time without interrupting the query answering component. In Section 5, we show that the time to execute a single iteration of SGD is insignificant.

2.1 Distributed SGD

To efficiently train machine learning models on large datasets, scalable techniques have to be employed. ~~As explained earlier,~~ SGD inherently works well with large amounts of data because it does not need to scan every data point during every iteration. However, when the dataset cannot fit into the memory of a single machine, it has to be distributed to multiple machines to run efficiently. In this situation, one common method is to distribute the gradient calculation tasks across nodes in a cluster, where each task allocated

to a node will work with a different part of the data. One problem of this approach is that a synchronization step is required before applying the updates to the model. This synchronization step slows down the SGD process, because after every iteration, all the updates have to be sent to a central process that updates the model. Experiments on distributed, asynchronous SGD show that the quality of the final model is not worse than the synchronized approach [30, 13].

2.2 Learning Rate Tuning

If we decide to go with April submission, try different learning rate tuning techniques

The learning rate or step size is a parameter of SGD, which controls the amount of change in model parameters during each iteration of SGD. It 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 until the model converges. However, in continuous and online learning due to possible changes in data distribution the parameters of a model may never converge. Adaptive learning rates have been used in online scenarios where the learning rate is adjusted automatically based on different criteria. Schaul et al. have proposed one such method were the learning rate can be inferred automatically based on the changes in the parameters of the model [32]. One advantage of their method is that it works well with non-stationary problems, where the distribution of the data is constantly changing. Based on observed changes in the data, one can adjust the learning rate to decrease (when approaching the optimum value) or 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 [38], clustering [6], neural networks [13], and matrix factorization [14]. Some examples of machine learning models that use SGD are:

Linear Classifiers are arguably the most common type of machine learning models built using the SGD optimization method. In our CTR prediction example described in Section 1, the logistic regression method is used to train models for predicting the click through rate [27]. Logistic regression models typically output a probability instead of a class label that indicates how likely an item belongs to a specific class [20]. In our 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 machines (SVM) represent another common class of classification models [36]. While logistic regression models aim to find parameters of a function that accurately fit the data points to the labels, SVM tries to separate the data points belonging to different classes. In many cases, both type of models work equally well, but depending on the data (whether it is linearly separable or not) and result requirements (predictions should be probabilities instead of the class labels) one method may be preferred to the other.

Matrix Factorization is a common method used in recommender systems [21]. Matrix factorization is used to derive the latent factors (e.g., for users and items) for recommender systems. It relies on the fact that each user and item

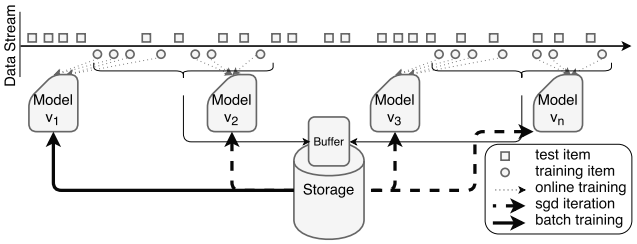


Figure 3: Continuous Training and Serving

can be described in a few dimensions (10 to 40 usually) based on the available ratings. These latent factors automatically capture the similarity of users and items based on the ratings provided by the users. Any unknown rating, hence can be predicted by computing the dot product of the user vector with the item vector. A scalable version of the algorithm was proposed by Gemulla et al. [15].

Neural Networks or deep learning – inspired by biological neural networks in the 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 successfully applied to training machine learning models. However, due to the slow training process and the lack of large amount of training data, they have not been used extensively in the machine learning community in the past. In the last decade, there was a drastic change due to several seminal publications. Hinton et al. proposed methods for speeding up the training of neural networks [19]. The ImageNet competition [31] in 2012 was won by a neural network proposed by Krizhevsky et al. where they significantly reduced the error rate [23]. The success of the Google's Deepmind team in achieving Neural Networks that were capable of defeating humans in the game of Go [34] and mastering Atari games [28] was also instrumental in popularizing neural networks in the machine learning community.

SGD is the ideal optimization algorithm for training neural networks since it works very well with large datasets (which are required for training neural networks). In fact, almost all of the recent work on neural networks use SGD for training them.

3. CONTINUOUS TRAINING AND SERVING

Our proposed deployment and maintenance system uses SGD as its underlying learning algorithm. As a result, it can update the model incrementally (one training item at a time) or use mini-batches of data (1 iteration of SGD). The core design principles of our deployment system are threefold. First, we incrementally update the model so that it can adapt to changes in the distribution of the incoming data. Second, we eliminate retraining and replace it with a series of consecutive iterations of SGD. And finally, we immediately use new batch datasets that are available to the system. Figure 3 shows how our deployment method works. First, using the existing data residing on disk, we train an initial model and deploy it into the system. The system receives prediction queries and training observations in a streaming fashion. The deployed model answers incoming prediction queries as soon as they are received. Once the system receives a training observation it updates the

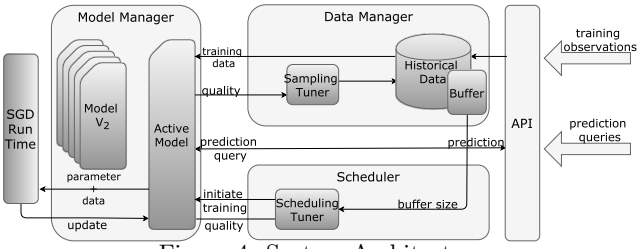


Figure 4: System Architecture

model incrementally. 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 when the system is not under heavy load. Each new iteration uses a random sample of the data in storage and the data in the buffer. Moreover, our system stores new batch datasets in the buffer (or the persistent storage unit) as soon as they become available. Any new scheduled iteration of SGD uses the newly introduced dataset to further train the model without requiring a new model to be retrained from scratch.

4. SYSTEM ARCHITECTURE

The proposed system comprises three main components; model manager, data manager and scheduler, and an independent SGD run-time. Figure 4 gives an overview of the architecture of our system and the interactions between its components. Incoming training observations are forwarded to the data manager. The data manager first stores the training observations in a buffer and then passes them on to the model manager. The model manager incrementally updates the model using the training observations. The model manager is also responsible for receiving prediction requests. Once it receives a request, it uses the latest version of the model to make a prediction and return the result to the user. Both scheduler and data manager components are constantly communicating with each other and with 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 the next iterations of SGD. Next, we explain each component of the system in more detail.

4.1 Scheduler

The scheduler component is responsible for scheduling of new iterations of SGD. 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. A new iteration of SGD is also executed when the system receives more training data than can be handled by the intermediate buffer. If the model is not updated with the new training items frequently, the quality decreases more rapidly, especially if the distribution of the data is changing. In our prototype, the scheduling rate is controlled by a user defined parameter, *max_buffer_size*. When the intermediate buffer's size reaches *max_buffer_size*, the scheduler executes a new iteration of SGD. It is important to note that the scheduling rate affects the quality of the model. In Section 5, we investigate the effect of scheduling rate on model quality. If 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 with 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 the model manager where the actual training and model updates happen. The data manager also communicates with the 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), the sampling rate can be adjusted. For example, when the data distribution is changing, a smaller sampling rate places more emphasis on the data that arrived recently. This is similar to the problem of concept drift where the distribution of the incoming data changes overtime. This renders 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. A 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 can 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 API of the model manager. The API is used to 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 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 may indicate a change in distribution of the data. As a result, reducing the sampling rate will put more emphasis on 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 [8] and Apache Spark [37] 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 [37] as our SGD run-time, but we plan incorporate Apache Flink in the complete version of the system. The model manager is the component responsible for communicating with the SGD run-time. In the current version of our prototype, the 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) [33]. ~~Therefore, the data manager should have support for HDFS. Both Flink and Spark provide out-of-the-box support for HDFS.~~

Name	#Users	#Items	#Ratings
Movie Lens 1M	6000	4000	1000000
Movie Lens 100k	1000	1700	100000

Table 1: Recommender System Datasets

Name	#Features	#Instances
Higgs	28	11000000
Susy	18	5000000
URL Reputation	3231961	2396130
Cover Type	52	581012
MNIST [25]	784	60000

Table 2: Classification Datasets

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 Setup

Environments: We evaluated our deployment method on two environments; local and distributed. We use a single node OS X machine with a quad-core Intel i7 and 16 GB of RAM for our local experiments. The distributed environment consists of 11 nodes (1 master, 10 slaves). Each node is running on a Intel Xeon 2.40 GHz 16 core processor and has 28 GB for dedicated memory for running our prototype.

Prototypes: we implemented two versions of our deployment method. Version 1 is implemented using python and uses some of the libraries available in scikit-learn [7]. This version is designed to work on a local environment. It gives a greater control over tuning the parameters of the system, since we are able to perform incremental updates one item at a time (as opposed to Spark's micro-batching). Version 1 supports neural networks (multi-layer perceptron) and recommender system (matrix factorization). Version 2 is implemented on top of Apache Spark [37]. It is using the SGD class of Spark's machine learning library, to execute the initial and incremental training. Version 2 works in both local and cluster environment and supports different types of linear models (SVM, Logistic Regression and Linear Regression).

Datasets: we use two different types of datasets in our experiments. To evaluate the recommender system we use Movie Lens [17] datasets, which are a collection of movie ratings. We use two versions of this dataset (100k and 1M), each with varying number of users, movies and ratings. Table 1 shows the details of Movie Lens dataset. We have sorted the Movie Lens 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. For these two datasets, we used an evolving test set to evaluate the performance of the deployment methods. 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, as a result captures the degree to which the models are adopting to the changes in the dataset.

To evaluate the classifier, we use a collection of binary classification datasets (mostly from UCI Machine Learning Repository[1]). Higgs and Susy [3] are physics datasets which are produced using Monte Carlo simulations. For these two datasets, the task is to distinguish between a signal process that produces a target particle (Higgs bosons particle for Higgs and supersymmetric particle for Susy) and a background process which does not. Cover Type [10] is a dataset used for predicting forest cover types from cartographic variables only. We use a binary class version of this dataset[2]. URL Reputation [26] is an collection of URLs and whether they are malicious or not. This dataset is collected over 120 days and features are anonymized, but correspond to lexical and host-based information gathered for each URL. MNIST [25] is a dataset of handwritten digits. Each image has 28x28 pixels and the labels are the 10 digits (0 to 9) the images are representing.

For all of the datasets we use 10% for initial training and 90% for online prediction/training. Except for URL Reputation and Movie Lens datasets that are timestamped, the order of the incoming data are arbitrary for the rest of the datasets.

Deployment 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 [11]. 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.

5.2 Implementation of ML Models

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

Recommender system: we implement a recommender system based on the matrix factorization using SGD [14]. Based on our initial experiments, we set the number of latent factors to 40. First, we train a model using the initial static data. To produce more accurate predictions, we have also included user, item, and global bias values as described in [21]. After the initial training, we deploy the model and use any new training observation to 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.

SVM Classifier: we implement a simple SVM classifier by extending the existing SVM classifier of Apache Spark. We implement the necessary methods described in section 4 for the SVM classifier to work with our deployment method. The extra methods are *update* which takes one (or a micro-batch in Spark Streaming) and incrementally update the model and *update_iteration* which takes a dataset and executes one iteration of SGD on the given data.

Neural Network: to evaluate our system on an image classification task, we implemented a multi-layer perceptron neural network using back propagation [9]. We set the number of hidden layers to 50 and use a softmax output function [4]. 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.

5.3 Tuning parameters

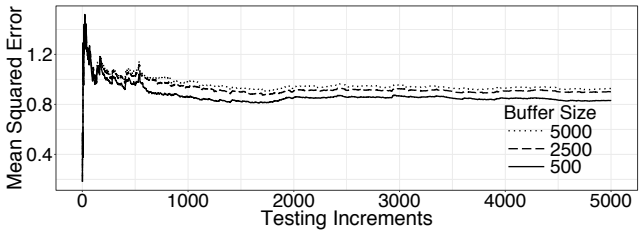
In section 3, we discussed how system parameters such as sampling and scheduling rate affect the performance and quality of the system. In this section, we analyze the effects of different sampling and scheduling rates on the system. Our goal, is to make these parameters adaptive, but for now we analyze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

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.

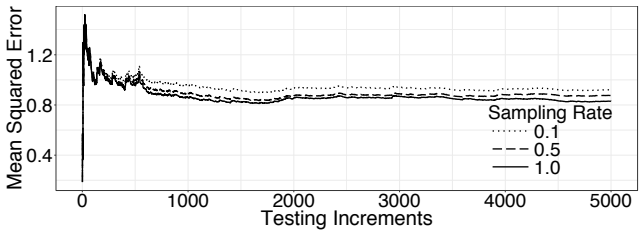
Figure 5a shows the mean squared error for different buffer sizes for Movie Lens 100k dataset. A smaller buffer size causes the scheduler to initiate training iterations more frequently. As a result, the underlying model is updated faster. However, the error rate is not decreasing linearly with the buffer 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 consid-

[1] http://archive.ics.uci.edu/ml/

[2] https://www.csie.ntu.edu.tw/~cjlin/libsvmtools/



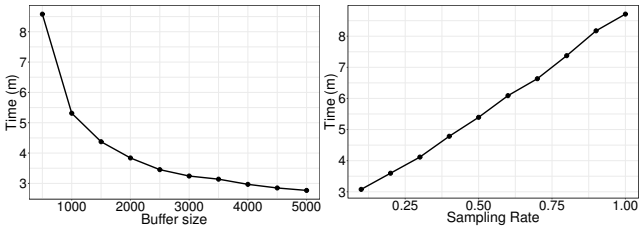
(a) Buffer size



(b) Sampling rate

Figure 5: Parameters effect on quality (Movie Lens 100K)

ering the effect the buffer size has on the running time. Figure 6a 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.



(a) (b)

Figure 6: Parameters effects on run time

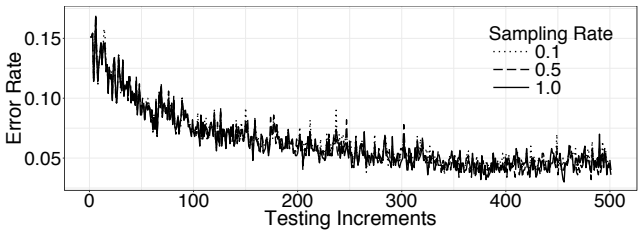
Dynamic scheduling: In production environments, the load on the system varies throughout the day. Therefore, a dynamic scheduling maximizes the performance of the system, by performing more frequent updates while there are more resources available for training. Moreover, since training and serving can 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 5b 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.

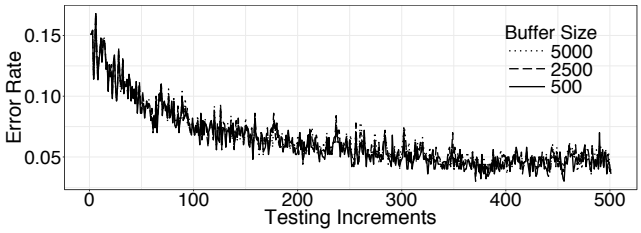
Figure 6b shows the effect of increasing sampling rate on running time. Using the entire historical data (sampling rate = 1.0) increases the running time 5 fold. Therefore, similar to scheduling rate, setting the sampling rate to smaller values will increase the performance substantially, while only slightly affecting the quality of the model.

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 applications. To demonstrate this, we perform the same set of experiments on the MNIST dataset. Figure 7b shows the effect of different sampling rates on the neural network classifer 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.

These two figures are just to show that these parameters dont always have the same effect .. should I include them?



(a) Buffer size



(b) Sampling rate

Figure 7: Parameters effect on quality (MNIST)

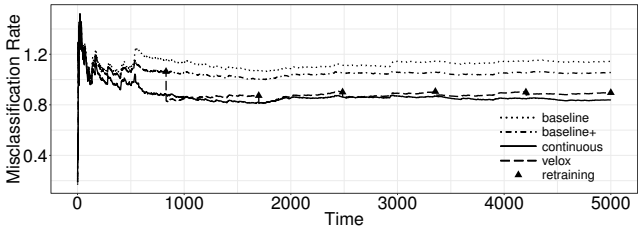
Figure 7a shows how buffer size affects the overall quality

of the neural network model. Similar to the 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.

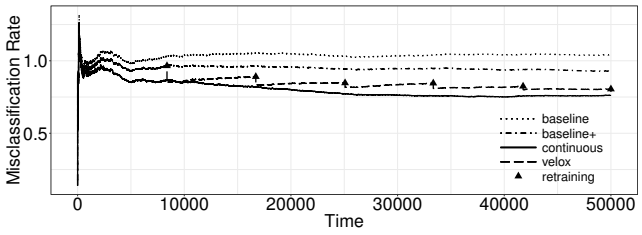
Based on our findings, we conclude that increasing the sampling and scheduling rate does not always affect the quality. In both the Movie Lens 100k and MNIST use cases the change in scheduling and sampling rate has small to no effect on the overall quality. However, the running time of the methods are 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.

5.4 Experiments in Local Environment

We evaluated our deployment method on the datasets described in this chapter. Our prototypes support two different methods of specifying the scheduling rate; buffer size and scheduling period. For recommender system experiments we use the buffer size to specify the scheduling rate (in Velox's case scheduling rate corresponds to offline retraining frequency).



(a) Movie Lens 100k

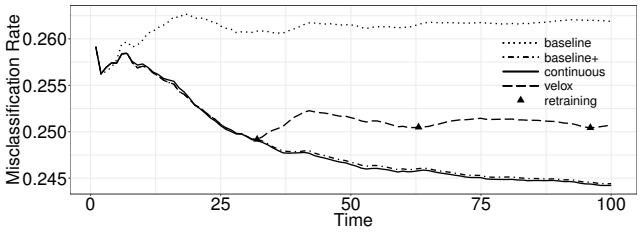


(b) Movie Lens 1M

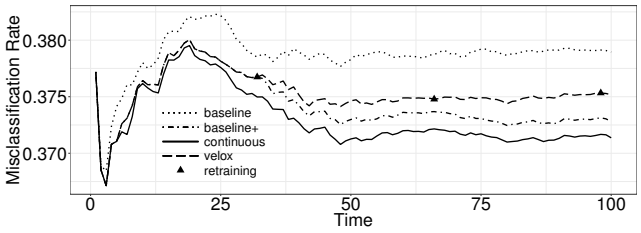
Figure 8: Recommender Systems

Figure 8a shows the mean squared error of the continuous, Velox, and several other methods on Movie Lens 100k dataset. The scheduling rate (buffer size) is set to 500 for Continuous method and 15,000 for Velox. All four methods perform an initial training, therefore, they have similar error rates in the beginning. However, the error rate of Baseline method starts to increase since it does not incrementally update the model. Baseline+ incrementally updates the model. As a result, it performs better than Baseline method overall. The error rate of Velox is similar to Baseline+ until the first scheduled retraining for Velox. After the retraining, there is a sudden drop in the error rate. For the remainder of the data, Velox performs as expected, once there is a retraining, the error rate immediately decreases. This decrease in error rate, is followed by a slow increase due the fact that Velox only incrementally updates the model until the next scheduled retraining, which as discussed earlier is not handling the change in the data distribution gracefully. Initial error rate of Continuous is similar to other methods. However, the error rate starts to decrease rapidly. The reason for the fast decrease in error rate is that immediately after the deployment of the model, Continuous method executes iterations of SGD. It consistently manages to adopt to the changes in data 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.

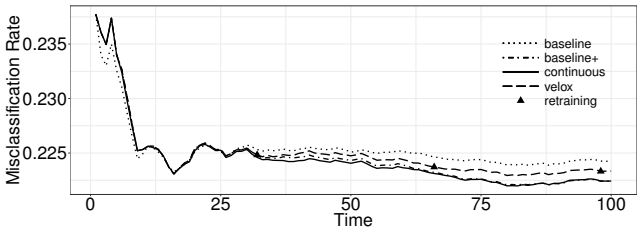
Figure 8b shows the mean squared error rate achieved by the implemented methods on Movie Lens 1M. Similar to Movie Lens 100k, the Continuous method has the lowest error rate among all the implemented methods. The difference in error rate between Continuous and Velox is even greater than the error rate 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) and Continuous method can adopt the changes in data distribution faster than other methods. 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 by Baseline+ , where due to the incremental updates of the model the error rate is lower.



(a) Cover Types



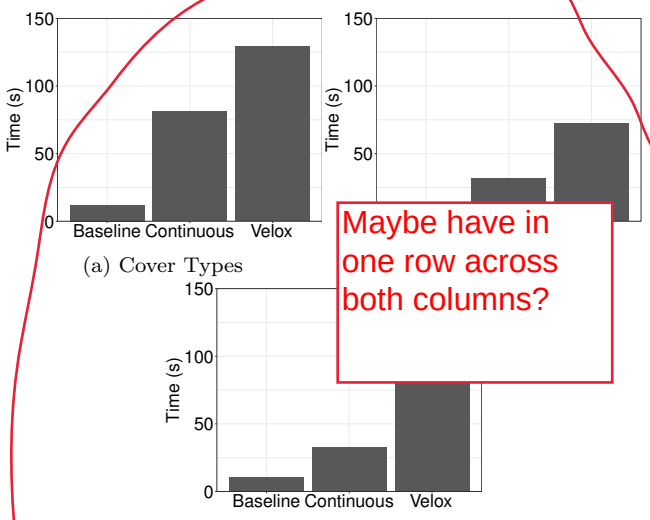
(b) Higgs Sample



(c) SUSY Sample

Figure 9: Classification Datasets

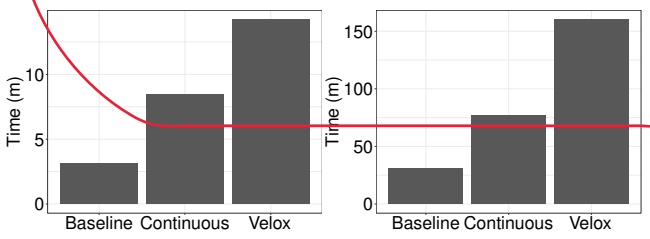
Figure 9 shows the misclassification rate over time of the deployment methods on several classification datasets. Contrary to the recommender system use case, the data used in these experiments is not time dependent and as a result the deployment methods behave differently. For Cover type (figure 12a), Continuous achieves the lowest error rate overall, although the difference with baseline+ is very small. Baseline method has the highest error rate overall because it is not performing incremental updates on the model. Interestingly, velox starts to perform worst after each retraining. This increase in error rate is the result of overfitting the model to the existing data. Higgs sample (figure 12b) error rate follows the same pattern as cover type. Continuous deployment achieves the lowest error rate followed by baseline+. After each retraining Velox's error rate slightly increases due to overfitting.



(a) Cover Types

(c) Susy Sample

Figure 10: Total training time for classification datasets



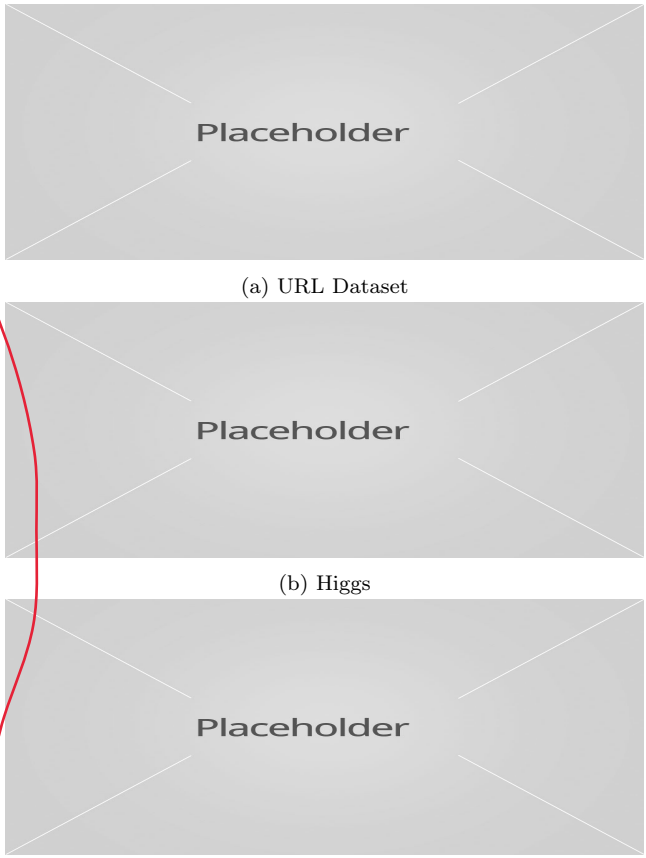
(a) Movie Lens 100k (b) Movie Lens 1M

Figure 11: Total training time for Recommender Systems

Figure 10 shows the total training time of deployment methods on local classification datasets. Figure 11 shows the total training time of deployment methods on local classification datasets. To measure this value, we calculated the total time that each method spent in training the model. We excluded the incremental training as the time is negligible. 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 total training time is 2 to 5 times smaller than Velox. 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 training 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.

5.5 Experiments in Distributed Environment

We evaluated our deployment system on larger workloads in a cluster environment.



(a) URL Dataset

(b) Higgs

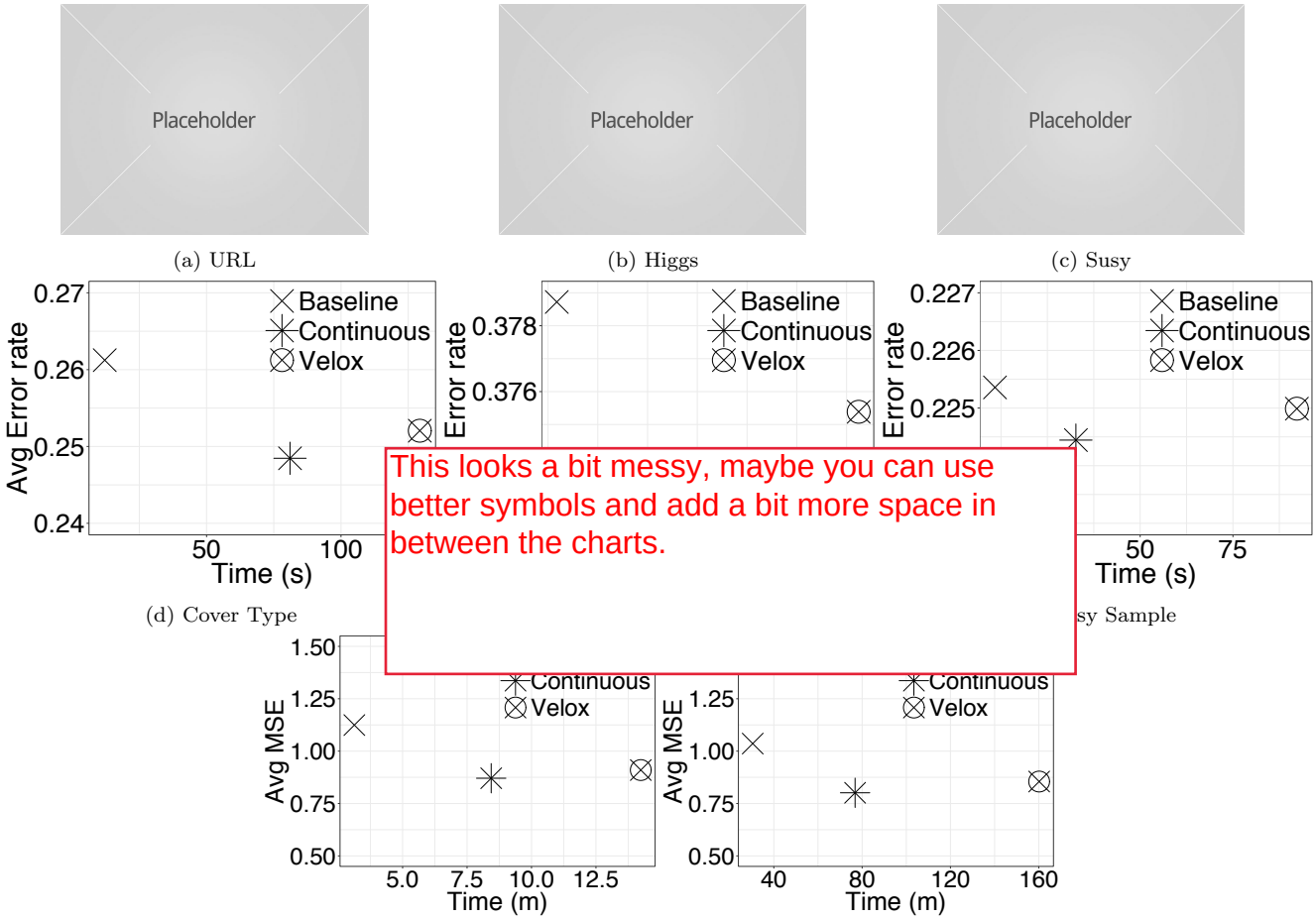
(c) SUSY

Figure 12: Large Classification Datasets

5.6 Discussion

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 [11], 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



(a) URL (b) Higgs (c) Susy

(d) Cover Type sy Sample

(g) Movie Lens 100K (h) Movie Lens 1M

Figure 13: Performance vs Quality

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. TensorFlow Serving provides mechanisms for real-time 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. Moreover, TensorFlow is designed to create and train only deep neural network models and does not work with other machine learning models. Our system supports incremental and batch updates to the model and automatically applies these updates to the model currently being served. Furthermore, our system can work with any machine learning model that uses stochastic gradient descent as optimization algorithm.

Velox is an implementation of the common machine learning serving practice [11], explained in Section 1. 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 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. 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 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, which both reduces the time spent on training and can produce an updated model swiftly. Moreover, our system makes use of new batch datasets as soon as they become available.

Clipper [12] 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 and for each prediction request, it uses the model with the highest confidence. However, it does not incrementally train the models in production which overtime leads to models becoming outdated. Our deployment method on the other hand, focuses on maintenance and constant updates of the models.

Weka [16], Apache Mahout [29], and Madlib [18] are systems that provide the necessary toolkits to train machine

learning models. All of these systems provide a range of machine learning training algorithms. However, they 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 [22] and TuPaq [35] 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. They focus on training high quality models by performing automatic feature engineering and hyper-parameter search. However, they only work with batch datasets and 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.

7. CONCLUSIONS

This paper presents an architecture for serving and continuous training of machine learning models. Most of the existing ML model serving systems do not provide support for continuous update [1, 12]. To ensure high quality models, some systems [11] perform incremental update and periodic retraining to ensure the quality of the model does not decrease. However, retraining is a resource intensive process which completely neglects any updates made to the model so far.

We propose a system for deploying and maintaining machine learning models that are trained using stochastic gradient descent optimization. In our system, we schedule iterations of SGD to run while the machine learning model is also answering prediction queries. After every iteration, the model is updated with ~~the~~ new parameters. The frequent updates~~,~~ help~~s~~ in adapting the model to ~~the~~ changes in data distribution. Moreover, our system is applicable to a wide range of machine learning models as demonstrated in our evaluation section.

Our experiments 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 show that not only continuous training requires less resources but it also produces models with lower error rates that adapt to the changes in data faster. ~~Moreover, c~~omparing 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] Pierre Baldi, Peter Sadowski, and Daniel Whiteson. Searching for exotic particles in high-energy physics with deep learning. *Nature communications*, 5, 2014.

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

[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] Lars Buitinck, Gilles Louppe, Mathieu Blondel, Fabian Pedregosa, Andreas Mueller, Olivier Grisel, Vlad Niculae, Peter Prettenhofer, Alexandre Gramfort, Jaques Grobler, Robert Layton, Jake VanderPlas, Arnaud Joly, Brian Holt, and Gaël Varoquaux. API design for machine learning software: experiences from the scikit-learn project. In *ECML PKDD Workshop: Languages for Data Mining and Machine Learning*, pages 108–122, 2013.

[8] 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.

[9] 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.

[10] Ronan Collobert, Samy Bengio, Yoshua Bengio, et al. A parallel mixture of svms for very large scale problems. *Neural computation*, 14(5):1105–1114, 2002.

[11] 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.

[12] 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.

[13] 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.

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

[15] 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.

[16] 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.

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

[18] 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.

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

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

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

[22] 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.

[23] 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.

[24] 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.

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

[26] 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.

[27] 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.

[28] 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.

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

[30] 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.

[31] 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.

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

[33] 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.

[34] 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.

[35] 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.

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

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

[38] 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.