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

A data analytics process is a pipeline comprised of several steps, from source selection, data preparation, feature engineering, to model training. Once the model is trained, it is deployed into a system where it can answer prediction queries reliably and in real-time. Current deployment systems perform online training and periodical batch retraining to maintain the quality of the model. However, retraining of models is a time-consuming and resource-intensive process.

We propose a novel deployment method for continuously training stochastic gradient descent (SGD)-based machine learning models. We utilize the iterative nature of SGD and replace the retraining process 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 deployment method updates models two to five times faster than the current deployment methods, without decreasing the quality of the model.

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 model quality as a complete, batched retraining. Therefore, in order to prevent the model's quality from degrading, the model should be retrained periodically using the data that has arrived at the system since the last training.

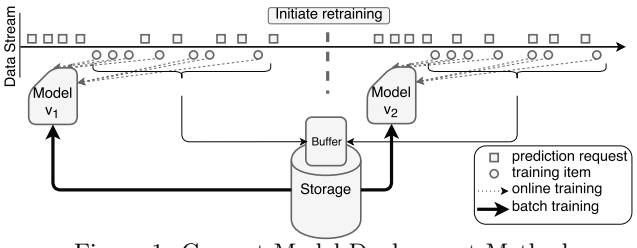


Figure 1: Current Model Deployment Method

Most of the current machine learning research focuses on training and providing tools to make model training and search easier. Kumar et al. provided an overview of existing machine learning systems [25]. They find that most of the surveyed systems provide little to no support after the model has been trained. Only few systems support deploying and maintaining the models [2, 11]. However, these are still lacking continuous monitoring and fast and accurate updates of the model.

Figure 1 demonstrates the most common model deployment approach. First, an initial model is trained based on an existing dataset residing on disk. Deploying this model to a production environment enables it to answer prediction queries arriving at the system. During production, the system continuously receives feedback in the form of new training observations. This enables the system to update the model incrementally. Note that not every machine learning model supports incremental updates. The underlying optimization strategy determines whether a model can be updated incrementally.

Example application: to illustrate this model deployment approach, consider the task of predicting the click through rate (CTR) of online advertisements. Figure 2 demonstrates the CTR prediction example for a search engine provider. 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 metadata of the available advertisements. Once such a 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 ob-

servations. The deployment system incrementally updates the underlying prediction model based on the new training observation. However, incremental updates alone are not enough to maintain the quality of the CTR prediction model. As a result, the deployment system periodically retrains the model using the entire data. This retraining may take hours or even days depending on the size of the data. Our proposed deployment method eliminates the need for this time-consuming retraining.

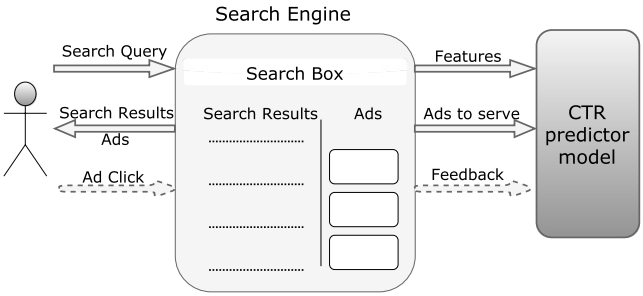


Figure 2: Use Case: Click Through Rate Prediction

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. Our contributions are as follows:

- We define a flexible system for machine learning models deployment and maintenance that allows any SGD-based machine learning model to be incorporated. We provide a prototypical implementation with several different machine learning model types
- We allow for incremental updates to the model, thus handling changes in the data distribution
- We eliminate the lengthy offline batch retraining and replace it with a series of single iteration of SGD without affecting the final quality of the model

Our experiments show that our approach improves the runtime of model training and deployment two to five times 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. Furthermore, while the deployment system is serving the model, new batch training datasets may become available. Our proposed deployment method is capable of utilizing the new datasets to further train the model without delay.

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

2. RELATED WORK

Traditional machine learning systems focus solely on training models and leave the task of deploying and maintaining these models to the users. It has only been recently that some systems, for example Velox [11], TensorFlow Serving [1], and LongView [2] have proposed architectures that also consider model deployment and query answering. LongView integrates predictive machine learning models into relational databases. It answers predictive queries and maintains and manages the models. LongView uses techniques such as query optimization and materialized view selection to increase the performance of the system. However, it only works with batch data and does not provide support for real-time queries. As a result it does not support incremental learning. In contrast, our system is designed to work in a dynamic environment where it answers prediction queries in real-time and incrementally updates the model when required. 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 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 have to be redeployed to TensorFlow Serving once the training is complete. 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 error rate of the model using a validation set. Once the error rate exceeds a predefined threshold, Velox initiates a complete retraining of the model using Spark. This deployment method, however, has three drawbacks; retraining discards updates that have been applied to the model so far, the process of retraining on full data set is resource intensive and time consuming, and new datasets introduced to the system only influence the model after the next retraining. Our approach differs, as it exploits the underlying properties of SGD to fully integrate the training process into the system's lifeline. This eliminates the need for completely retraining the model and replaces it with consecutive SGD-iterations. Moreover, our system can train the model on new batch datasets as soon as they become available.

Clipper [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. For each prediction request, it uses the model with the highest confidence. However,

it does not incrementally train the models in production, which over time leads to models becoming outdated. Our deployment method on the other hand, focuses on maintenance and continuous updates of the models.

Weka [17], Apache Mahout [30], and Madlib [19] are systems that provide the necessary toolkits to train machine learning models. All of these systems provide a range training algorithms for machine learning methods. However, they do not provide any management, before or after the models have been deployed. Our proposed system focuses on models trainable using stochastic gradient descent and as a result is able to provide model management both during training and deployment time.

MLBase [23] and TuPaq [36] are model management systems. They provide a range of training algorithms to create machine learning models and mechanism for model search as well as model management. They focus on training high quality models by performing automatic feature engineering and hyper-parameter search. However, they only work with batch datasets. Once models are trained, they have to be deployed and used for serving manually by the users. Our system, on the contrary, is designed for deployment and maintenance of already trained models.

3. STOCHASTIC GRADIENT DESCENT

Machine learning minimizes an objective function (often referred to as the loss function). Usually, this requires us to calculate 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 each iteration uses the entire training data set to calculate the gradient value. One drawback of gradient descent is that in presence of large datasets it performs very slow. Stochastic gradient descent [5] is an approximation of the gradient descent method. Similar to gradient descent, it is an iterative process. However, in each iteration it calculates the gradient at a single element (or a small sample) 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 [5]. Each iteration of SGD executes 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 performs one iteration of SGD at a time without interrupting the query answering component. In Section 6, we show that the overhead from executing a single iteration of SGD is very small and does not affect the query answering component.

3.1 Distributed SGD

To efficiently train machine learning models on large datasets, scalable techniques have to be employed. SGD inherently works well with large amounts of data because it does not need to scan every data point during every iteration. However, for very large datasets, SGD has to perform many iterations in order to converge. To decrease the running time, large datasets can be distributed among multiple nodes, where each node will compute the gradients on a subset of the data in parallel. One drawback of this approach is that a synchronization step is required before applying the updates to the model, which slows down the optimization process. To alleviate this, several asynchronous SGD methods are proposed [32, 13]. Experiments using these methods show that the quality of the produced model is similar to the synchronized SGD approach.

3.2 Machine Learning Models based on SGD

SGD is a common optimization methods that has been used in classification [41], 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 the CTR prediction example of Section 1, we use logistic regression to train the model for predicting the click through rate [28]. Logistic regression models typically output a probability instead of a class label that indicates how likely an item belongs to a specific class [21]. In our example, logistic regression predicts the click probabilities for all the available advertisements and the ones with the highest probabilities are displayed. Support vector machines (SVM) represent another common class of classification models [37]. While logistic regression aims 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 types of models work equally well. However, depending on the data (whether it is linearly separable or not) and result requirements (probability values or class labels) one method may be preferred to the other.

Matrix Factorization is a common method used in recommender systems [22]. 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 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. Hence, any unknown rating 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. [16].

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 [20]. The ImageNet competition [33] in 2012 was won by a neural network proposed by Krizhevsky et al. where they significantly reduced the error rate [24]. The success of the Google's Deepmind team in achieving Neural Networks that were capable of defeating humans in the game of Go [35] and mastering Atari games [29] 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 recent work on neural networks uses SGD for training them.

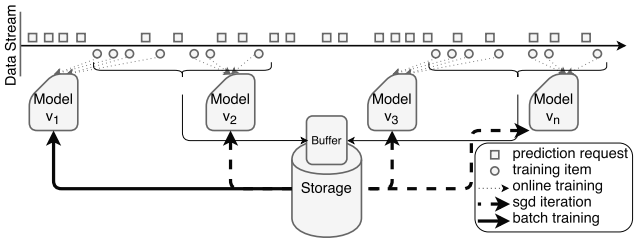


Figure 3: Continuous Training and Serving

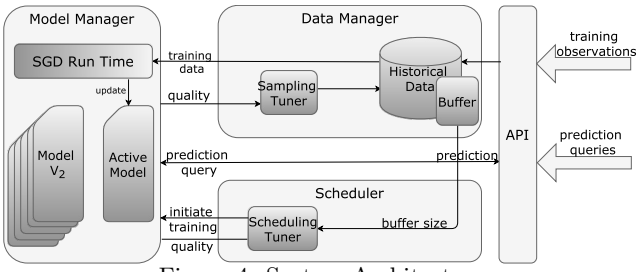


Figure 4: System Architecture

4. 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 model incrementally. The system also keeps track of incoming training observations and adds them to an intermediate buffer. A scheduler component, triggers new iterations 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 further scheduled iteration of SGD incorporates the new data without requiring the model to be retrained from scratch.

5. 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 among its components. The data manager first stores the incoming 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 returns the result to the user. Both the scheduler and the 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. This in turn helps us to tune the scheduling and sampling rate for future iterations of SGD. Next, we explain each component of the system in more detail.

5.1 Scheduler

The scheduler component is responsible for scheduling new iterations of SGD. Intuitively, the best time to execute an iteration is when the system is not under heavy load. 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. This decrease in the quality is more rapid 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 6, 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.

5.2 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. Upon a new training iteration, the data manager accesses the historical data stored on disk and provides a sample.

Different sampling strategies can be used to provide the sample. In our current prototype, the data manager uses a simple unified random sampling method to generate this sample. More advanced methods, such as Reservoir [39] or weighted random sampling can also be used to generate the sample. Reservoir sampling is typically used to generate samples from large datasets that do not fit in memory, whereas weighted random sampling is used when data elements have different weights. In an online machine learning scenario, recent items are more important for training the model and are assigned a bigger weight than older items. Therefore, weighted random sampling can generate samples that can contribute to the training of a better model.

The data from the sample and the data in the buffer are merged to create the dataset for next training iteration. The data manager provides access to this dataset for the model manager in order to further train the model. 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 is a system 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 5.3), 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 6, 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.

5.3 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, 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 5.2, an increase in the error rate may indicate a change in distribution of the data. As a result, reducing the sampling rate will place more emphasis on recent data (in the intermediate buffer) and help adapt 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 SGD-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. Moreover, in case of anomalies in the data, they have to be identified and discarded before any further model updates could happen.

5.4 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. Any system capable of performing incremental and batch SGD updates efficiently are suitable options for our system. Apache Flink [8] and Apache Spark [40] are distributed data processing platforms 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 [40] as our SGD run-time.

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 types of updates 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 the Hadoop Distributed File System (HDFS) [34].

6. 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.

6.1 Setup

Environments: We evaluate 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 an Intel Xeon 2.40 GHz 16 core processor and has 28 GB of dedicated memory for running our prototype.

Prototypes: We implement two versions of our deployment method. Version 1 is implemented in python and utilizes the scikit-learn machine learning library [7]. This version works in local environments. It supports incremental updates to the model, one item at a time (as opposed to the micro-batching of Apache Spark). As a result, we are able to investigate the effects of scheduling and sampling rates more accurately. Version 1 supports neural networks (multi-layer perceptron) and recommender algorithms (matrix factorization). In our evaluations, we use Version 1 to

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
URL Reputation	3231961	2396130
Cover Type	52	581012
MNIST	784	60000
SEA	3	60000
Adult	123	48882

Table 2: Classification Datasets

investigate the effect of scheduling and sampling rate on the running time and quality of the models. Version 2 is implemented on top of Apache Spark [40]. It uses the SGD implementation available in the machine learning library of Apache Spark. This version works in both the local and cluster environment and supports different types of linear models (SVM, Logistic Regression, and Linear Regression).

Datasets: We perform our experiments on two different types of datasets. To evaluate the recommender system we use 2 versions (100k ratings and 1M ratings) of the Movie Lens datasets, which are collections of movie ratings [18]. Table 1 shows the details of the Movie Lens dataset. We have sorted the Movie Lens dataset based on the timestamp of the ratings. Every item is examined one by one according to their timestamp. This is to evaluate how each of the implemented methods react to the changes in the distribution of the data. For these two datasets, we use an evolving test set to evaluate the performance of the deployment methods. We start with an empty test set. Upon receiving new test items, we add them to the test set and calculate the error on the current test set (we call this a test increment). For Movie Lens 100k, the total size of the test set is 5,000 and for Movie Lens 1M, the test size is 50,000.

To evaluate the classifier, we use a collection of classification datasets (mostly from the UCI Machine Learning Repository[1]). Higgs is a physics datasets which is produced using Monte Carlo simulations [3]. For this dataset, the task is to distinguish between a signal process that produces a target particle (Higgs bosons) and a background process which does not. Cover Type is a dataset used for predicting forest cover types from cartographic variables [10]. We use a binary class version of this dataset[2]. URL Reputation is a collection of URLs and the task is to identify malicious URLs [27]. This dataset is collected over 120 days. The features of this dataset are anonymized, but they correspond to lexical and host-based information gathered for each URL. MNIST is a dataset of handwritten digits [26]. Each image has 28x28 pixels and the labels are the 10 digits (0 to 9) the images are representing. SEA is an artificially generated dataset that has been injected with concept drift (change in data distribution) [38]. Attributes are floating point values between 0 and 10 and there are a total of 4 concepts (15,000 instances in each concept). Adult is a dataset extracted from a census database [31]. It contains information such as education level, occupation, relationship status, and salary of individuals from multiple countries. The task in this dataset is to predict whether an individual earns more than $50K a year. For all of the datasets we use 10% of the data for initial training and 90% for online prediction/-training. Except for URL Reputation and the Movie Lens datasets that are timestamped, the order of the incoming data are arbitrary for the rest of the datasets. For all of the classification datasets, we report the prequential error when receiving a new training instances [15] .

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

Baseline is the naive and simplest deployment model. In Baseline, we train a model from the initial data. We use this model throughout the lifetime of the application without any further incremental or batch training.

Baseline+ is similar to Baseline with the added incremental learning. After the initial model is deployed, it is incrementally updated with 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 new training item. Velox also retrains the model periodically using the entire training data.

Continuous is the implementation of our proposed method. Similar to Velox and Baseline+, a model is trained using the initial data. Continuous also updates the model incrementally with new training data. Based on the rate of incoming training data, the scheduler component triggers new iterations 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.

6.2 Implementation of ML Models

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. To produce more accurate predictions, we include user, item, and global bias values as described by Simon Funk [22]. After the initial training, we deploy the model and use the new training observations 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; *update* and *update_iteration* as described in Section 5. *Update* takes one (or a micro-batch in Spark Streaming) item and incrementally update the model. *Update_iteration* 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 implement 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]. 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.

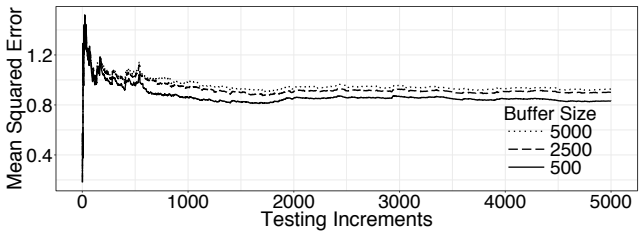
[1] `http://archive.ics.uci.edu/ml/`

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

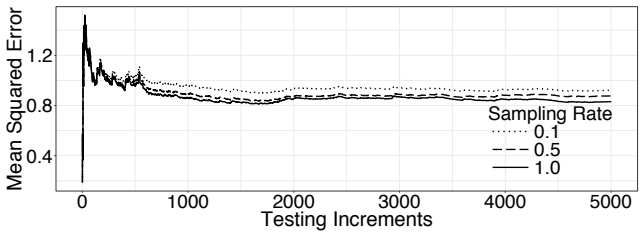
6.3 Tuning parameters

In Section 4, 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.

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. Executing one iteration of SGD, even using the entire data, is not a resource heavy process, and can be executed simultaneously with the serving component of the system.



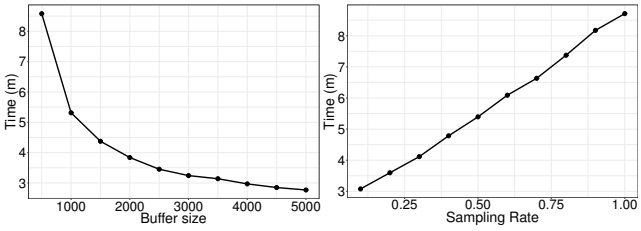
(a) Buffer size



(b) Sampling rate

Figure 5: Effect of Sampling and Scheduling Rate on Quality (Movie Lens 100K)

Figure 5a shows the mean squared error for different buffer sizes for Movie Lens 100k. A smaller buffer size forces the scheduler to initiate training iterations more frequently. As a result, the system updates the underlying model more often. However, the error rate is not decreasing linearly with the buffer size. Further analysis shows that the more frequent the model updates are, the faster the model converges and any further training has little to no effect on the overall quality. This is extremely important, specially when considering the effect of the buffer size on the running time. Figure 6a shows the running time on Movie Lens 100k using different buffer sizes. Increasing the buffer size from 500 to 5000 decreases the running time by a factor of 5 while the MSE is only decreased 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)

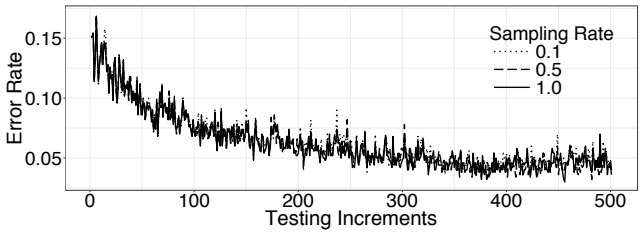
Figure 6: Effect of Sampling and Scheduling Rate on Running time (Movie Lens 100K)

Dynamic scheduling: In production environments, the load on the system typically varies throughout the lifetime of the application. 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 a background process and only update the weights when the training iteration is over.

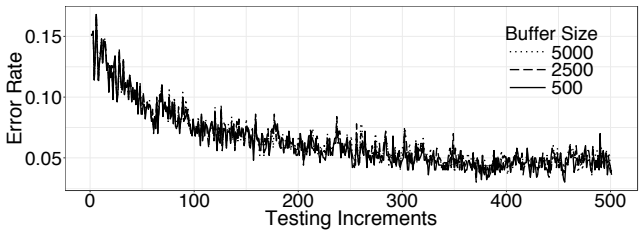
Sampling rate: In each iteration of SGD the data inside the buffer and a sample of the historical data is combined to update the model. In this section, we investigate the effect of the sampling rate on the model quality and the running time of the system. Figure 5b shows that a larger sampling rate increases the quality of the model. However, similar to the scheduling rate, the decrease in error rate is negligible considering the effect it has on running time. This is caused by the same phenomena, where the model after training on bigger sample rates start to converge faster. As a result, bigger sample sizes do not have a considerable effect on the quality.

Figure 6b shows the effect of increasing the sampling rate on the running time. Increasing the sampling rate from 0.1 to 1.0, increases the running time by a factor of 5. Therefore, similar to the 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: The underlying machine learning model and the dataset have big effects on the selection of sampling rate and scheduling rate. In the recommender system use case, due to the changes in the incoming data distribution, we see that bigger sample rates and higher scheduling rates have an effect (although small) on the quality of the model. However, this is not be the case for every application. To demonstrate this, we perform the same set of experiments on the MNIST dataset. Figure 7a shows the effect of different sampling rates on the neural network classifier model for MNIST. Contrary to the results we achieved for Movie Lens 100k, the error rates for different sampling rates are very similar. This is caused by how neural networks behave. Increasing the sampling rate causes similar data items to be used repeatedly in consecutive training iterations. Neural networks are not affected by this oversampling, therefore the results are almost similar with different sampling rates. Moreover, in this experiment, the number of parameters of the 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. Therefore it is not affected by more training, unless new training observations arrive at the system.

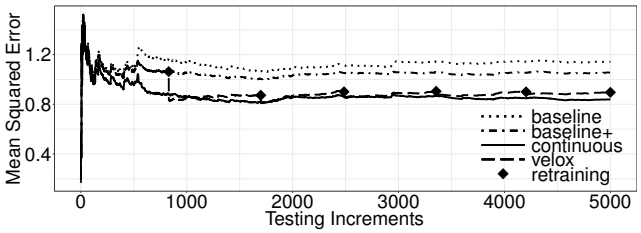


(a) Sampling rate

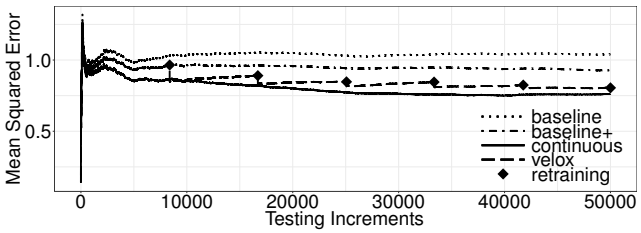


(b) Buffer size

Figure 7: Effect of Sampling and Scheduling Rate on Quality (MNIST)



(a) Movie Lens 100k



(b) Movie Lens 1M

Figure 8: Mean Squared Error of Recommender Systems

Figure 7b shows how the buffer size affects the overall quality of the neural network model. Similar to the sampling rate case, the error rate of the model is not affected by the scheduling rate. New training observations that exist in the buffer have the maximum effect on the quality of the model 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 the 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 datasets, the change in scheduling and sampling rate have 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.

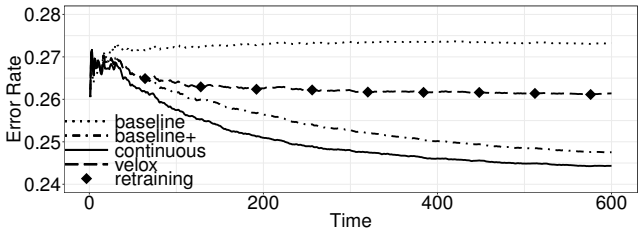
6.4 Experiments in the Local Environment

In this set of experiments, we investigate the error rate and training time for the different deployment methods on the smaller datasets in the local environment. For all of the datasets, the sampling rate of Continuous is set to 0.2.

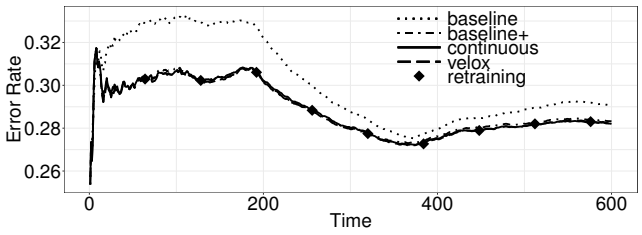
Figure 8a shows the mean squared error for Continuous, Velox, Baseline+, and Baseline on Movie Lens 100k. The scheduling rate (buffer size) is set to 500 for Continuous and 15,000 for Velox. All four methods perform an initial training, therefore they have similar error rates in the beginning. However, the error rate of Baseline gradually increases since it does not incrementally update the model. Baseline+ incrementally updates the model. As a result, it has a lower error rate than Baseline. 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. After every retraining the error rate immediately decreases. This decrease in the error rate is followed by a slow increase. The increase is 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 the 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 executes iterations of SGD. It consistently manages to adopt to the changes in data and decreases the error rate. As expected, except for immediately after a full retraining of Velox, Continuous always performs better than all the other methods.

Figure 8b shows the mean squared error rate achieved by the implemented methods on Movie Lens 1M. Similar to Movie Lens 100k, Continuous 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. This is because the change in data distribution in Movie Lens 1M is greater than Movie Lens 100k (data in Movie Lens 1M is gathered from a longer period of time). Since Continuous adapts to the changes in distribution faster than Velox, it maintains a lower error rate throughout. The error rate of Velox 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, the error rate of Baseline is the highest among all the implemented methods, followed by Baseline+ , where due to the incremental updates of the model the error rate is lower.

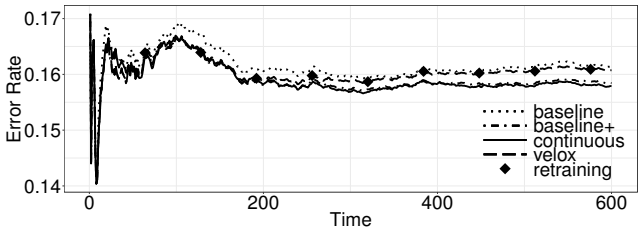
Figure 9 shows the misclassification rate over time for the implemented deployment methods on several classification datasets. Contrary to the recommender system use case, the data used in these experiments is not time dependent (except for SEA) and as a result the deployment methods behave differently. For Cover type (Figure 9a), Continuous achieves the lowest error rate overall, followed by Baseline+. Baseline method has the highest error rate because it is not performing incremental updates on the model. The error



(a) Cover Types



(b) SEA



(c) Adult

Figure 9: Error rate of the SVM Classifiers

rate of Velox slightly increases after the first retraining and it never recovers from this increase. By performing a retraining, the underlying model in Velox overfits to the existing data. As a result, the test error on incoming data does not decrease the same way that it does in the case of Baseline+ and Continuous. The error rate for all of the deployment methods, except for Baseline is very similar on SEA (Figure 9b). Continuous still perform slightly better than Baseline+ and Velox, although with a very small margin. Figure 9c shows the error rate of different deployment methods on Adult dataset. In this dataset, the different deployment methods perform rather similarly. Baseline has the highest error rate overall, however after the 6th retraining the difference between the error rate of Velox and Baseline becomes smaller. This is, similar to the Cover types dataset, due to the over training of the underlying model in Velox. Baseline+ and Continuous maintain a steady error rate overall, with Continuous slightly outperforming Baseline+.

Figure 10 shows the total training time for the deployment methods on local datasets. To measure this the total training time, we calculate the total time that each method spent in training the model. We excluded the incremental training time as both Velox and Continuous perform incremental training as part of their process. The total training time of Baseline is equal to the initial training time. Both Continuous and Velox perform the same initial training, as a result, their total training time cannot be smaller than Baseline. For the classification datasets (Figure 10a, 10b, 10c), Continuous and Velox are approximately 25 and 50 times slower than Baseline. This is expected, since Baseline only trains the model once. Moreover, this training is executed over a smaller dataset (10% of the entire data). For recommender system experiments, the difference in total training time is smaller. In our python prototype, the entire dataset is first loaded into the memory and then it is streamed to the deployment system one by one. As a result, both Continuous and Velox can train the model faster. In all of the evaluated datasets, the total training time of Continuous is 2 to 5 times smaller than that of Velox. This is because a full retraining incurs a 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 stays approximately the same, whereas the time for a full retraining increases exponentially.

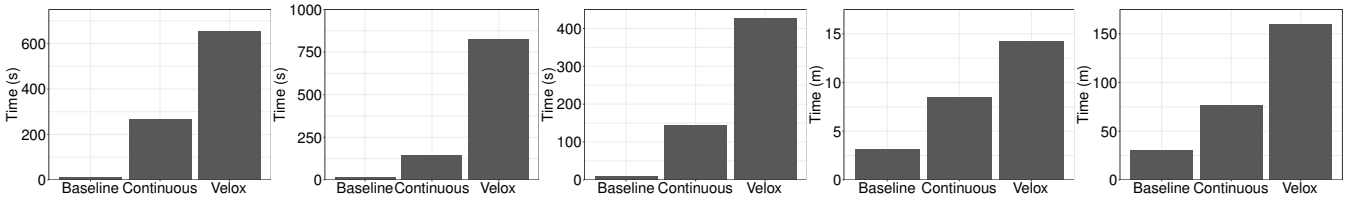
6.5 Experiments in the Distributed Environment

In this set of experiments, we investigate the performance of the deployment methods on large datasets in the distributed environment. Figure 11 shows the error rate of different deployment methods on classification datasets. Since the URL Reputation data was gathered over 120 days, it is very likely that the data contains changes in the distribution. This concept drift is reflected in Figure 11a. The error rate of Baseline is increasing with a steady rate as it cannot adapt to the changes in the data. All the other three methods perform similarly in the beginning. However, after the 4th retraining the error rate of Velox increases slightly. Similar to the previous experiments, Velox is overfitting to the data after a few retraining processes and as a result, it does not recover from the increase in the error rate. Continuous and Baseline+ maintain a lower error rate throughout with Continuous performing slightly better than Baseline+.

Figure 11b shows the error rate on the Higgs dataset. Similar to the local classification datasets, the error rate for Baseline is the highest among all of the deployment methods. Velox, Baseline+, and Continuous maintain the same error rate until the first retraining of Velox, after which due to possible overfitting, the error rate of Velox slightly increases. Continuous performs slightly better than Baseline+ in the beginning, however, the underlying models of both deployment methods slowly converges. Therefore, the error rates of both of the deployment methods become similar.

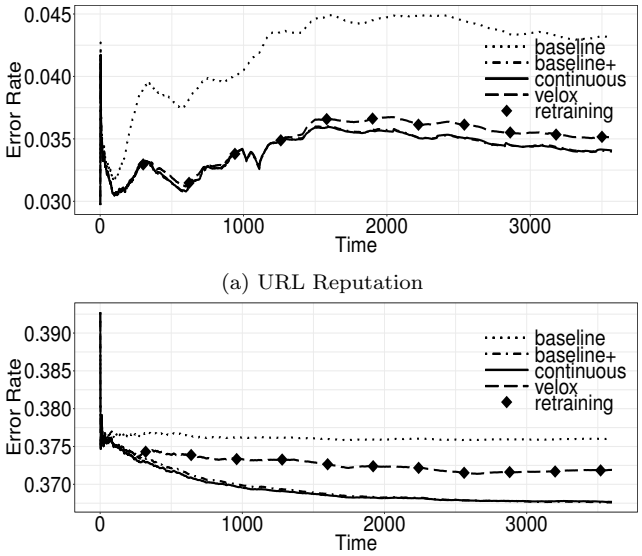
Figure 12 shows the total training time for the URL Reputation and Higgs experiments. Similar to local datasets, Baseline has the smallest total training time. However, the difference between the training time of Baseline and other methods is not as large. In our local experiments, the entire resources of the computer were being consumed to perform the training. However, in the distributed environment, the cluster is underutilized in the beginning as the amount of data is small. As more data arrives at the system, the utilization of the cluster increases. However, the training time for Continuous and Velox does not increase linearly with size. Therefore, the total training time for Continuous is only 5 to 10 times larger than that of Baseline (instead of 20 to 25 times for the local datasets) and the total training time for Velox is 20 to 25 times larger than that of Baseline (instead of 50 to 55 times for the local datasets).

6.6 Discussion



(a) Cover Types (b) SEA (c) Adult (d) Movie Lens 100k (e) Movie Lens 1M

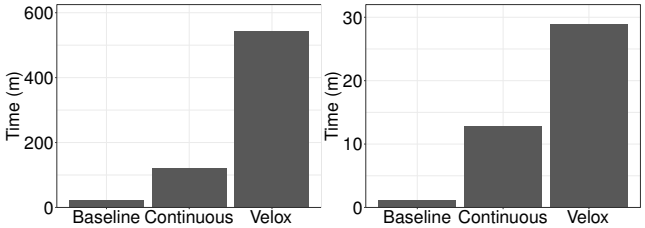
Figure 10: Total Training Time



(a) URL Reputation

(b) Higgs

Figure 11: Error rate of the SVM Classifiers



(a) URL Reputation (b) Higgs

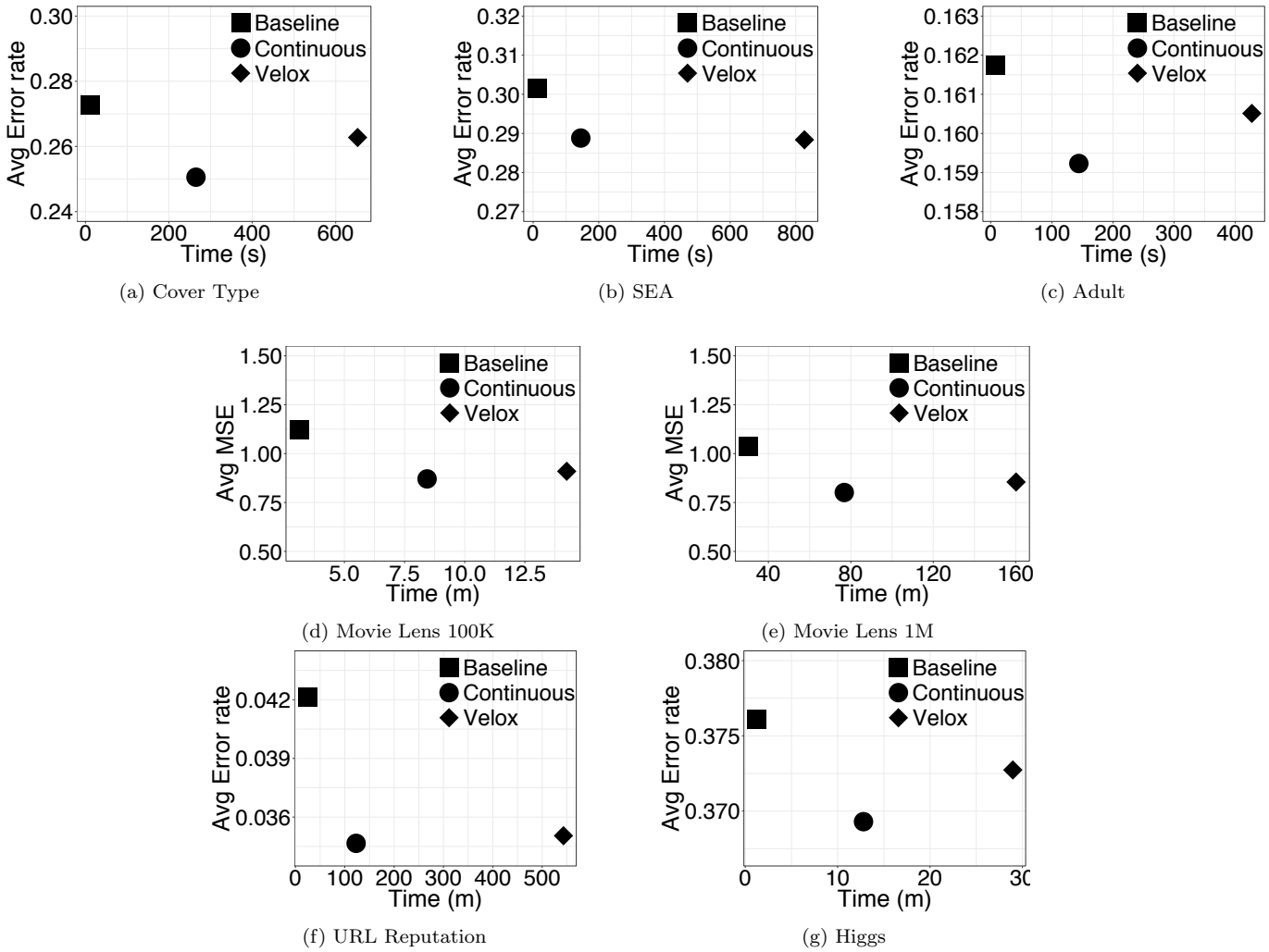
Figure 12: Total Training Time

Figure 13 shows the total training time and average error rate of each of the three different deployment methods (Baseline, Continuous, and Velox) on different datasets. For all of the evaluated datasets, Continuous achieves the lowest average error rate (for the classification datasets) and average mean squared error (for the recommender system datasets).

The difference in average error rate (or average MSE) between Velox and Continuous is smaller for datasets that contain concept drift (Figure 13b, 13d, 13e, and 13f). Since both Velox and Continuous perform incremental and batch training of the model after it is deployed, they both manage to handle the concept drift in the data. However, Continuous is able to adapt to the changes faster as it is continuously updating the model by executing iterations of SGD. Since the retraining process is resource intensive, Velox cannot execute it frequently. As a result, the underlying model in Velox cannot adapt to the changes in the distribution as quickly as Continuous. The performance of Baseline is very poor in datasets with concept drift. This is expected, as the underlying model in Baseline is only trained on an initial dataset. Therefore, it is not capable of adapting to the changes in the data.

Continuous performs very well for datasets with no concept drifts (Figure 13a, 13c, and 13g) as well. For the Cover Type and Higgs datasets, Baseline does not converge on the initial dataset, therefore it has a higher error rate than Velox and Continuous. Velox performs better than Baseline on Higgs and Cover Type, however, retraining on these two datasets causes the underlying model to overfit to the existing data. As a result, the performance of the model is decreased after each retraining. This leads to an overall lower average error rate than that of Continuous.

As expected, Baseline has the lowest training time for all the datasets since it only trains the model only once on the initial datasets. The total training time for Continuous is 2 to 5 time smaller than Velox for every datasets. This has a big impact on prediction latency and accuracy. In our current prototype, we do not address the problem of the trade off between prediction latency and accuracy. In our prototype, both prediction and model updates are managed by the same node. As a result, the prediction component is paused until the SGD iteration (or retraining in the case of Velox) is executed. Therefore, the system always answers each prediction query using the latest version of the model, although with a much greater delay. However, in an actual model deployment system, model training and prediction answering are typically executed on separate nodes (or threads). Therefore, any prediction request arriving at the system is answered immediately, although not with the latest version of the model. Since the retraining time for Velox is large, a considerable percentage of prediction requests are always answered by an older version of the model. As a result, while the system is executing a retraining, the error rate will continue to rise until the retraining is finished. For example in the URL Reputation dataset, the average time of each retraining is 68 minutes in Velox, whereas the average time of each iteration of SGD is 1.5 minutes. This means that in worst case scenario, the model that is answering a prediction request in Velox is outdated by 1 hour. Our continuous deployment method reduces this delay since it is updating the underlying model constantly using smaller batches of data.



(a) Cover Type (b) SEA (c) Adult

(d) Movie Lens 100K (e) Movie Lens 1M

(f) URL Reputation (g) Higgs

Figure 13: Total Training Time vs Quality

7. CONCLUSIONS

We propose a system for deploying and maintaining machine learning models that are trained using the SGD optimization method. Our deployment approach eliminates the need for offline retraining of the model, without affecting the quality. In our system, we schedule iterations of SGD to run while the machine learning model is answering the prediction queries. After every iteration, the model is updated with the new parameters. The frequent updates help in adapting the model to changes in the data distribution. Moreover, our system is applicable to a wide range of machine learning models as demonstrated in our evaluation.

Our experiments show that the continuous training of the machine learning models is much faster than full retraining, 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. Comparing to simple techniques such incremental learning or initial batch training, our technique produces a model with higher quality.

In future work, we will explore more advanced methods for sampling of historical data in order to investigate their effect on the performance of the system. Moreover, we plan to investigate the trade off between prediction latency and prediction accuracy.

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] João Gama, Raquel Sebastião, and Pedro Pereira Rodrigues. Issues in evaluation of stream learning algorithms. In *Proceedings of the 15th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 329–338. ACM, 2009.

[16] 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.

[17] 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.

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

[19] 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.

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

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

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

[23] 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.

[24] 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.

[25] 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.

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

[27] 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.

[28] 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.

[29] 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.

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

[31] John C Platt. 12 fast training of support vector machines using sequential minimal optimization. *Advances in kernel methods*, pages 185–208, 1999.

[32] 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.

[33] 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.

[34] 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.

[35] 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.

[36] 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.

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

[38] W Nick Street and YongSeog Kim. A streaming ensemble algorithm (sea) for large-scale classification. In *Proceedings of the seventh ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 377–382. ACM, 2001.

[39] Jeffrey S Vitter. Random sampling with a reservoir. *ACM Transactions on Mathematical Software (TOMS)*, 11(1):37–57, 1985.

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

[41] 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.