Batch Methods for Incremental Learning

Noah Golmant
University of California, Berkeley
Department of Computer Science
noah.golmant@berkeley.edu

Evan R. Sparks
University of California, Berkeley
Department of Computer Science
sparks@cs.berkeley.edu

Joseph Gonzalez
University of California, Berkeley
Department of Computer Science
jegonzal@cs.berkeley.edu

ABSTRACT

Data is increasingly collected and processed in an online fashion, but existing large-scale machine learning systems are batch-oriented. Existing stream-oriented ML algorithms are often limited in application scope or present significantly stronger assumptions to the learning model. We present two techniques to automatically and generically adapt batch-oriented algorithms to the incremental setting, extend an existing large-scale system to support the algorithms, and demonstrate the effectiveness of these techniques on several classification workloads.

1 INTRODUCTION

The speed at which data is collected and expected to be acted upon is rapidly increasing. Traditional methods for training and updating machine learning models utilize batch processing techniques, but these approaches become more and more expensive as data arrives more quickly.

Incremental and online learning techniques offer the opportunity to improve model performance by integrating new data in a continuous manner [5]. This avoids the need to fully retrain on every new batch of data. However, most machine learning algorithms are designed for batch environments [1] and deploying these algorithms in the online setting can be challenging due to the sensitivity of algorithm parameters to changing workloads and in many cases may degrade accuracy significantly.

In this work, we investigate two generic techniques to efficiently and accurately incorporate new data into an existing model in order to significantly reduce training time while minimizing degradation in prediction accuracy. In both methods, we introduce light-weight model update procedures that leverage existing batch learning algorithms by constructing fixed datasets from the stream. In order to achieve these goals, we make the assumption that data arrives into the system in mini-batches of a fixed size, as in large-scale streaming systems like Spark's Structured Streaming [9].

The key insight in this work is that we can leverage a duality between spatial and temporal partitioning of data to transform advances in communication-efficient statistical optimization designed to enable learning across space to derive new streaming algorithms that effectively *communicate efficiently across time*. That is, we take algorithms that are designed for physically partitioned datasets and apply them to physically *and* temporally partitioned datasets, enabling systems support for incremental learning with modest additional overhead and implementation effort, The two techniques we propose here leverage existing distributed machine learning frameworks to derive stream oriented machine learning algorithms. We again emphasize the generic nature of these techniques: this allows us to implement a broader class of optimization algorithms that are traditionally less well-suited for streaming environments.

Finally, we demonstrate how such these extensions can be effectively incorporated into a batch-oriented system. We extend KeystoneML [10], a framework to construct distributed, end-to-end machine learning pipelines, to use these incremental techniques with minimal changes. We experimentally validate the expected trade-offs between statistical accuracy and training run-time in a large-scale incremental learning environment.

2 BACKGROUND

The majority of work in distributed machine learning has been aimed towards batch learning algorithms and optimization techniques. In the batch setting, we receive large batches of training points from which we produce a machine learning model that one then deploys to the field. To produce a model, these methods require as input a large number of examples, processed all at once. However, many such methods do not scale well enough to handle larger data workloads and parameter spaces[2]. Additionally, if the underlying input data distribution changes over time in a phenomenon known as temporal or concept drift [13], this training process becomes impractical and actually degrades statistical accuracy as the model becomes ”stale”.

In the incremental learning setting, we receive new data and update an existing machine learning model to extend model knowledge [5]. The model stores only some of the statistics of the training data, and the learning algorithm requires strategies to update the model while preventing it from ”forgetting” the statistics of older data. Such algorithms are typically difficult to transfer directly into an incremental setting, and require significant modifications to do so. There are no ”one-size-fits-all” incremental learning algorithms.

KeystoneML [10] is a framework to construct end-to-end machine learning pipelines by composing a series of logical operators on data. In this work, we extend KeystoneML on Apache Spark to introduce new operators to perform incremental learning in a primarily batch-oriented system. The techniques and operators developed here also easily extend to frameworks like Spark's Structured Streaming.

3 BATCH METHODS FOR INCREMENTAL LEARNING

Consider a learning algorithm on a live data stream at some time t. We have an existing parameter vector θ_t that represents the estimated parameters of the model at time t. Our goal is to extend this model to incorporate n new data points from the stream to produce the next parameter vector, θ_{t+1}.

In the presence of infinite memory and computing power, we could store all data and retrain on everything we have seen so far. This guarantees high performance, since we are using the best offline methods available, but requires a model updates that run quadratic in the number of samples. Given memory and computing constraints, we must instead learn primarily from this new data.

To do so, we consider three methods. One of these demonstrates the issues surrounding incremental learning. The other methods are able to produce good models using batch-oriented algorithms that operate on fixed subsets of the stream. Their generic nature allows us to optimize training objectives using techniques other than streaming stochastic gradient descent, like L-BFGS [2].

3.1 One-shot SGD

Perhaps the best known incremental learning algorithm for convex optimization is Stochastic Gradient Descent (SGD). Given an input dataset, SGD iteratively computes a stochastic approximation of the true gradient of the objective function, using only a small batch of training samples.

Unfortunately, building a model with SGD requires multiple passes, or *epochs*, over the training data. In the incremental learning setting, it is impossible to make multiple passes over the training data because it is potentially infinite, and we wish to look at the new data exactly once– we instead can only perform *one-shot SGD*. As we will show in Section 5, this lack of access to data history during training can result in models that significantly under-perform models trained using the conventional method.

To address this issue, we now propose two incremental update algorithms which take advantage of established theoretical results to learn models that perform closer to those trained in the batch setting than those trained using one-shot SGD with similar requirements in terms of computation and memory.

3.2 Reservoir Sampling

We consider learning on a dataset X of size N, which is divided into m partitions, each of size n. We denote partition t of X as $X_t = \{x_1^t, x_2^t, \ldots, x_n^t\}, \forall t \in \{1, \ldots, m\}$. Each entry x_i^t of the partition is some real-valued, d-dimensional vector. To perform reservoir sampling, we construct a reservoir R, which is an array consisting of k elements. We store the first k points we see immediately in R. As we receive a new batch X_t containing n new elements, we consider each element $x_i^t \in X_t$ such that with probability $1/t$, we replace some existing element in R with x_i^t, and with probability $1 - 1/t$ we keep the old item. This provides an unbiased sample of the dataset at time t, and so running SGD on this sampled subset provides an unbiased estimate of the gradient of the loss function [11].

To train the model for time $t + 1$, we initialize the learning algorithm to the parameter vector θ_t obtained in the previous step. We mix in the n samples according the procedure above, and we apply our learning algorithm to train a new parameter vector, θ_{t+1}, on the examples contained in R. We can use any existing batch-oriented algorithm to learn a model from a fixed-size dataset constructed from the stream.

3.3 Model Averaging over Time

We now consider learning a parameter vector θ_t that attempts to minimize an objective function $f(X_t)$ on a particular partition. We can create a new model with parameter vector θ by simply averaging all the previously learned parameters:

$$\theta = \frac{1}{m}\sum_{t=1}^{m}\theta_t$$

Assuming the objective f follows certain convexity constraints commonly satisfied by machine learning techniques that fall in the class of convex empirical risk minimization (e.g., ridge regression and logistic regression), then the authors of [12] prove that this model achieves mean squared error that decays as $O(N^{-1} + (N/m)^{-2})$. Moreover, recent work suggests that this simple method is still useful in optimizing over the highly non-convex loss surfaces of neural networks [7].

Now we can consider the situation in which we sequentially process the partitions in an unbounded stream, and we do not know the number of partitions, m, or the total number of points, N. Then we can instead compute the new parameter vector θ_{new}, minimizing the objective $f(X_t)$ on the current partition, and we incorporate it into the model for time $t + 1$ using a weighted average:

$$\theta_{t+1} := \alpha_t\theta_{new} + (1 - \alpha)\theta_t$$

For $0 \leq \alpha_t \leq 1$. This update step is easy to perform, and it does not make any restrictions on the types of algorithms used to produce θ_{new}. This enables, for example, the use of second-order methods like L-BFGS in a streaming context. In our experiments, we will use $\alpha_t = 1/(t+1)$ to recover a simple cumulative average. However, one may use this technique to incorporate additional knowledge about distributional drift or model confidence in a simple manner.

4 INCREMENTAL LEARNING FOR KEYSTONEML

The KeystoneML framework allows the user to compose batch-oriented machine learning pipelines using a set of logical operators. We attempt to make a minimal set of changes to the underlying framework to allow for incremental learning. In KeystoneML, an `Estimator` is a logical operator that is applied to a distributed collection of data items to produce a `Transformer`, which is itself an operator that can be applied to data items to produce new data items (e.g. features or predictions).

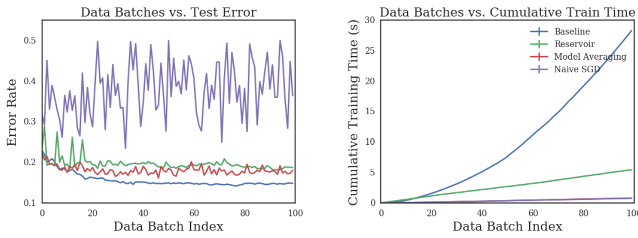
4.1 Core addition: Incremental Estimators

Our core addition to KeystoneML is a new logical operator to facilitate incremental learning. This operator, the `IncrementalEstimator`, does not change the underlying methods to compose operators into a pipeline. With it we introduce the `Model` interface. a `Model` can represent any learned parameters for an estimator, e.g. the weights of a logistic regression model.

4.2 Incremental updates

The first `IncrementalEstimator` takes an initial `Model` object and a collection of data items to produce a new `Model`. A new `Transformer` uses a `Model` to make predictions on data. When new data is available to train on, we can use the `Model` produced at the previous step as the starting point for the new `IncrementalEstimator`. We detail the interface in figure 5. This approach is similar to the `partial_fit` API of the popular `sklearn` package in Python, but operates at scale and within an overarching pipelined structure. It also fits into the existing computation graph structure of KeystoneML which enables significant efficiency gains.

Figure 1: Microbenchmark statistics for trigger classification on the ACE2005 dataset.



(a) Statistical performance on a test set. **(b) Cumulative training runtime.**

As an example, consider a `LogisticRegression` class that implements the `IncrementalEstimator` interface. At time t we learn the weights vector θ_t given the new data X_t. A `Transformer` object uses θ_t to make predictions on data. At the next timestep, we supply both θ_t and the new data, X_{t+1}, as arguments to the `LogisticRegression` class. This produces a new weight vector, θ_{t+1}, and a `Transformer` that uses these weights to make new predictions. An example of this procedure is shown in figures 3 and 4.

This is flexible enough to accommodate batch-oriented algorithms, as well as the incremental adaptations that use the methods outlined in section 3.

5 EVALUATION

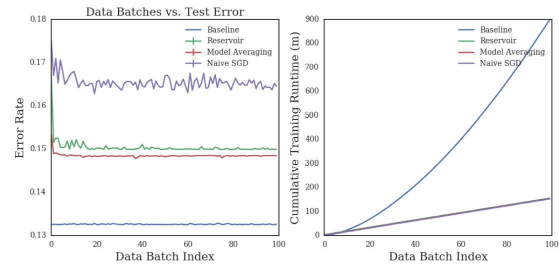
We compare the reservoir sampling and model averaging methods against a baseline and a naive incremental adaptation. The baseline represents the degenerate case of training a new model using all data we have ever seen. In the naive incremental adaptation, we use an online algorithm like stochastic gradient descent with an annealed learning rate.

We first test the results on a microbenchmark, using an NLP binary classiication task derived from the ACE2005 event extraction challenge. The goal of this task is to identify whether a word token is a "trigger" word, i.e. if it indicates the presence of an event in a sentence. Word tokens are represented as vectors extracted from a Word2Vec model [8]. We implement the incremental and baseline methods using logistic regression in the scikit-learn package in Python. Finally, we test the model averaging technique using a new incremental API in KeystoneML. For both of the model averaging and reservoir sampling methods, we optimize the logistic regression objective using and L-BFGS solver.

5.1 Incremental Loss Comparisons

We compare the different approaches on test error as a function of the number of data batches seen. The results are shown in figure 2a. Note that the final error of the reservoir model is within 8% of the baseline, and model averaging results in a model with error within 10% of the baseline. The naive incremental method, however, fails to achieve comparable results. We emphasize that the poor performance of the One-Shot SGD case is due to the fact that each piece of data is presented exactly once.

Figure 2: Statistics for the Amazon reviews dataset on KeystoneML.



(a) Statistical performance on the test set. **(b) Cumulative training runtime.**

```
val featurizer = Trim andThen
    LowerCase andThen
    Tokenizer andThen
    NGramsFeaturizer(1 to 2) andThen
    TermFrequency(x => 1) andThen
    (CommonSparseFeatures(1e5), data)
val textClassifier = featurizer andThen
    (LogisticRegression(lambda=1e-3), data, labels)
val predictions = textClassifier(testData)
```

Figure 3: A batch KeystoneML text classification pipeline gains support for incremental updates with only minor changes to the highlighted core learning operators.

```
val prevModel = textClassifier.model
val newTextClassifier = featurizer andThen
    (LogisticRegression(lambda=1e-3), newData, newLabels, prevModel)
val newPredictions = newTextClassifier(testData)
```

Figure 4: We incrementally update the existing model with new training data.

```
abstract class IncrementalEstimator[A, B, M] {
  def withData(data: RDD[A], oldModel: M): (Pipeline[A, B], M)
  def fit(data: RDD[A], oldModel: M): M
  def transformer(model: M): Transformer[A, B]
}
```

Figure 5: An API for the IncrementalEstimator operator.

5.2 Training runtime

In figure 2b, we plot the cumulative training time of the various algorithms. We observe a linear speedup in total training time between the baseline and incremental methods with respect to the number of data batches processed.

5.3 Implementation in KeystoneML

We implement the IncrementalEstimator operator according to the API specified in figure 5. We test it using the model averaging technique with a text classification pipeline based on logistic regression over TF-IDF [6] n-gram features of a corpus of Amazon product reviews, as illustrated in Figure 3. The dataset consists of 65m product reviews with 100k sparse features. We simulate an incremental

environment by dividing the dataset into 100 equally sized partitions. The online components of this pipeline are highlighted in the figure. Importantly, the application developer-facing changes required to make this pipeline support incrementalism were only modest changes to the operator definitions for the `Estimator` operators, `CommonSparseFeatures` and `LogisticRegression`. Statistical performance and training runtime for this pipeline is shown in figures 3a and 3b. The training time improvement and the learning curves are similar to those obtained in the microbenchmark implementation.

6 RELATED WORK

Related to this work is the HAZY [4] system, which encompasses training and inference inside an RDBMS, and provides cheap incremental model updates to incorporate new data. However, model performance iteratively degrades using the cheap updates, and HAZY uses heuristic strategies to determine when to update the model to use all the data available.

We refer to [12] to validate the model averaging technique in a physically distributed setting. This work differs in that it distributes the workload across a stream instead of across independent machines.

Reservoir sampling is a technique to choose a random sample of k elements from a list containing N items, where N is very large or unknown. The authors of [3] demonstrate an efficient way to perform reservoir sampling in a memory-constrained environment.

KeystoneML [10] is a system to construct end-to-end large-scale machine learning pipelines in a distributed environment. KeystoneML uses a high-level API to specify pipelines by composing logical operators. We directly extend it to implement incremental learning in a batch-oriented system.

7 CONCLUSIONS AND FUTURE WORK

In this work we have demonstrated the effectiveness of two novel incremental learning techniques that work with traditionally distributed, batch-oriented algorithms and are simple to implement. Both empirically achieve a small degradation in predictive performance for a large speedup in training time.

The model averaging technique has been analyzed in the case of some convex objective functions, but recent work has demonstrated its potential in the case of the non-convex objectives of neural networks in a federated learning context [7]. It remains to be seen how effective these physically distributed deep learning techniques are in a streaming context.

Several questions remain open for future work. For example, future work may investigate how incremental learning performance varies as the size of a data batch varies over time. Also, it remains to be seen that these techniques are effective in the presence of targeted concept drift.

REFERENCES

[1] R. Ade and P. Deshmukh. Methods for incremental learning: a survey. *International Journal of Data Mining & Knowledge Management Process*, 3(1):119, 2013.

[2] L. Bottou, F. E. Curtis, and J. Nocedal. Optimization Methods for Large-Scale Machine Learning. *ArXiv e-prints*, June 2016.

[3] P. S. Efraimidis and P. G. Spirakis. Weighted random sampling with a reservoir. *Information Processing Letters*, 97(5):181–185, 2006.

[4] X. Feng, A. Kumar, B. Recht, and C. Ré. Towards a unified architecture for in-rdbms analytics. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*, pages 325–336. ACM, 2012.

[5] P. Joshi and P. Kulkarni. Incremental learning: Areas and methods-a survey. *International Journal of Data Mining & Knowledge Management Process*, 2(5):43, 2012.

[6] C. D. Manning, P. Raghavan, H. Schütze, et al. *Introduction to information retrieval*, volume 1. Cambridge university press Cambridge, 2008.

[7] H. B. McMahan, E. Moore, D. Ramage, and B. A. y Arcas. Federated learning of deep networks using model averaging. *CoRR*, abs/1602.05629, 2016.

[8] T. Mikolov, I. Sutskever, K. Chen, G. S. Corrado, and J. Dean. Distributed representations of words and phrases and their compositionality. In *Advances in neural information processing systems*, pages 3111–3119, 2013.

[9] M. P. Singh, M. A. Hoque, and S. Tarkoma. Analysis of systems to process massive data stream. *CoRR*, abs/1605.09021, 2016.

[10] E. R. Sparks, S. Venkataraman, T. Kaftan, B. Recht, and M. J. Franklin. KeystoneML: Optimizing pipelines for large-scale advanced analytics. In *ICDE*. IEEE, 2017.

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

[12] Y. Zhang, M. J. Wainwright, and J. C. Duchi. Communication-efficient algorithms for statistical optimization. In F. Pereira, C. J. C. Burges, L. Bottou, and K. Q. Weinberger, editors, *Advances in Neural Information Processing Systems 25*, pages 1502–1510. Curran Associates, Inc., 2012.

[13] I. Zliobaite. Learning under concept drift: an overview. *CoRR*, abs/1010.4784, 2010.