Data Life Aware Model Updating Strategy for Stream-based Online Deep Learning

Wei Rang, Donglin Yang, Dazhao Cheng*
Department of Computer Science
UNC Charlotte
Charlotte, USA
{wrang, dyang33, dazhao.cheng}@uncc.edu

Kun Suo
Department of Computer Science
Kennesaw State University
Kennesaw, USA
ksuo@kennesaw.edu

Wei Chen
Nvidia Corporation
Santa Clara, USA
weich@nvidia.com

***Abstract*—Many deep learning applications deployed in dynamic environments change over time, in which the training models are supposed to be continuously updated with streaming data in order to guarantee better descriptions on data trends. However, most of the state-of-the-art learning frameworks support well in *offline* training methods while omitting *online model updating* strategies. In this work, we propose and implement *iDlaLayer*, a thin middleware layer on top of existing training frameworks that streamlines the support and implementation of online deep learning applications. In pursuit of good model quality as well as fast data incorporation, we design a Data Life Aware model updating strategy (DLA), which builds training data samples according to contributions of data from different life stages, and considers the training cost consumed in model updating. We evaluate iDlaLayer's performance through both simulations and experiments based on TensorflowOnSpark with three representative online learning workloads. Our experimental results demonstrate that iDlaLayer reduces the overall elapsed time of MNIST, Criteo and PageRank by 11.3%, 28.2% and 15.2% compared to the periodic update strategy, respectively. It further achieves an average 20% decrease in training cost and brings about 5% improvement in model quality against the traditional continuous training method.**

***Index Terms*—Online Learning, Model Updating Strategy, Data Life Cycle**

I. Introduction

Deep Learning (DL) has played an essential role in many practical domains, such as pattern recognition, recommendation system and data mining. Most DL applications are running in the environments where input datasets are dynamic streaming and data changing patterns are unexpected. For example, both speed and even syntax of people's speaking tone keep changing in conversations, user interests always shift in watching movies, climate data are frequently changing in weather forecasting. When facing these phenomena, also known as concept drift [1], predicting models based on static data becomes inaccurate and obsolete very soon, causing failures in future predictions.

Recently, there are a couple of approaches to tackle concept drift problems, among which a typical one is online learning [2]. It realizes continuously model updating with dynamic data streams. Over the past decade, various online learning algorithms have been designed, e.g., Linear discriminant analysis [3] for pattern recognition, matrix factorization for recommendation system [4] and Bayesian inference for streaming data analytics [5]. With these algorithms, many issues on concept drift are solved and a range of applications are also developed in the industrial area. For example, Google [6] and Facebook [7] adopt online learning in predicting advertisement clicks. Netflix [8] uses a similar method in movie recommendation to its subscribers as well. However, the successful adoption of online learning is far less elegant since most online DL applications aim to improve the prediction accuracy with the sacrifice of code simplicity and easy-to-use interfaces. Furthermore, more hardware is expected to run such bloated systems, which significantly increase the system budget. Many popular DL frameworks, such as Coooolll [9] TensorflowOnSpark [10] never explicitly support for running online applications, and even less in model updating strategies. Currently, to achieve online learning purposes, users have to develop specific training loops to manually update models via a few basic strategies, such as continual and periodic updating approaches.

In light of the above issues, our prime goal is to develop a system that friendly supports online learning applications without much coding modifications and hardware resources. To be exact, we plan to introduce an easy-to-use middleware solution based on existing DL frameworks to streamline the support and implementation of stream-based workloads. Moreover, an efficient model updating strategy is necessary to determine how and when to perform model updating for DL applications considering data life stages and training costs. In this work, we tackle these challenges with iDlaLayer, a thin middleware layer prototype atop DL frameworks (e.g., TensorflowOnSpark) that facilitates stream-based online learning applications. Our prototype provides a few universal APIs to applications, with which two core functionalities of online learning, transmitting data and control information to the backend training framework, are realized. It eliminates cumbersome work on designing training loops and offers a novel model updating strategy. iDlaLayer determines how to build combined training samples based on data life cycle from data samples and when to perform model updating with a lower training cost. Then, iDlaLayer notifies the backend

*Dr. Dazhao Cheng is the corresponding author. This work is supported by the National Science Foundation under Grant CCF-1908843.

978-1-7281-6677-3/20/$31.00 ©2020 IEEE

training framework to retrain the model with these combined data samples. After model updating completion, a new model is sent to the model serving systems, e.g., Clipper [11], and is ready to serve inference requests. Specifically, we summarize the key contributions as follows:

- We empirically study the difference between offline and online learning in performance when running DL algorithms on an existing framework. We further investigate the reasons behind performance variations, which are mainly laid on data samples and updating policy.
- We introduce a concept named *data life cycle*, with which streaming data is divided into various life stages in terms of how much contributions it made to model updating.
- We propose a novel data life aware updating strategy (DLA), which relies on combined data sample and considers training cost when deciding whether to perform a model updating action.
- We implement iDlaLayer on top of TensorflowOnSpark and evaluate its performance with three representative applications. Our experimental results demonstrate that iDlaLayer reduces the overall elapsed time of MNIST, Criteo and PageRank by 11.3%, 28.2% and 15.2% compared to the periodic update strategy, respectively.

The rest of this paper is organized as follows. We describe background and motivations in Section II. Section III gives the detailed system design and implementation. We conduct extensive experiments and present the results in Section IV. We briefly review related work in Section V. Finally, we conclude our paper in Section VI.

II. Background and Motivation

In this section, we first give a brief introduction of online DL and the motivation of the continual model updating in dynamic environments. We then discuss the challenges and opportunities for online DL applications via case studies.

A. Offline vs. Online Deep Learning

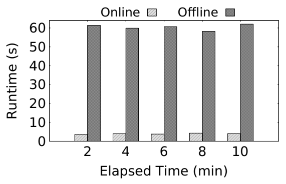
For many DL applications, the training stage is performed in an offline mode with batch data streams, which refers to an input dataset $S = \{s_1, s_2, s_3,...\}$ with sample s_i = {x, y} consisting of an instance x and a target label y. The purpose of the training stage is to find a proper parameter θ for a model to predict a label y for each instance x correctly. Deep learning algorithms iteratively update θ over training samples, in an offline or online manner, depending on whether the whole training dataset is available or not. In online DL, there are new training samples available over time, so the model is updated according to such samples. Specifically, an online learning algorithm updates its parameter θ by $\theta = f(\theta_i, s_i)$ when a new sample s_i is available, where f is an optimization function adopted in the algorithm. For offline learning, the parameter is updated with the entire batch S (containing a set of samples) of training dataset and the model is trained by iteratively applying an optimization function f' to all samples. That is, at the iteration of the training stage, the learning algorithm updates its parameter by $\theta = f'(\theta_k, S)$. Comparing these two learning strategies, it is evident that online learning is more lightweight and responses more promptly than offline learning as online mechanism only incorporates one data sample a time. This prominent feature makes online learning more efficient in dynamic environments where the data source keeps changing over time, in which the trained model has to be frequently updated to accurately describe the trend of input data. In the following subsection, we demonstrate this benefit and possible weakness with case studies.

B. Case Study

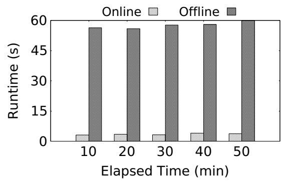
We study two popular online DL applications, i.e., pattern recognition and classification prediction, which are based on Convolutional Neural Network (CNN) [12] and Deep Neural Network (DNN) [13] respectively. For pattern recognition, we adopt an MNIST [14] database to evaluate its performance on recognizing people's handwriting while using a 1GB dataset of user clicking activities to check efficiency in classification prediction.

To run these two applications, we adopt a testbed consists of 15 nodes, each of which has Intel Xeon(R) CPU E5-2630v4@2.20GHz x 20 and 64GB DDR3 RAM, running Ubuntu 16.04 LTS operating system with kernel version 4.0, Scala 2.10.0, and Hadoop YARN 2.8.0 for cluster management. One node serves as the *master*, and all the other 14 nodes serve as *slaves*. These nodes are connected with Gigabit Ethernet. We run these two applications on TensorflowOnSpark [10]. We use the built-in example of TensorflowOnSpark to evaluate performance over MNIST database. Another DNN study goes to classification prediction. This system decides what type of ads should be displayed to different users by predicting the probability of user click. We choose real-world traffic logs provided by Criteo [15] as the input data to evaluate performance. In the case study, we first train a base model using a small portion of the database and periodically update the model with dynamic input data. Then the performance differences in training time is compared between offline and online learning approaches. We finally evaluate how online learning improves model quality with continually generated data.

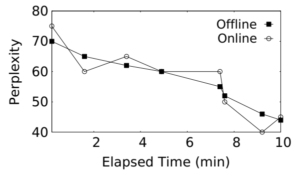
Training Time: Figure 1 (a) and (b) compare the training time achieved by online and offline training for CNN and DNN workloads respectively. In particular, we update the training model every 2 minutes in both online and offline manners. At each monitoring point, the offline method would retrain the model from scratch over all available data while the online mechanism only takes in the new arrival data. These processing logics result in the orders of magnitude difference between the two methods, making offline retraining a poor performance in dynamic environments. It is evident that the online training is significantly faster than the offline training. The reason behind is that offline training takes orders of magnitude longer time to retrain the model in an offline manner by incorporating all historical data. In contrast, online training only uses new arrival data and does not need to wait for the moment that all data are available.



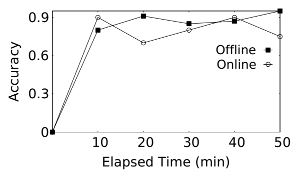
(a) CNN Training time



(b) DNN Training time



(c) CNN Model quality



(d) DNN Model quality

Fig. 1. The comparison of training time and model quality achieved by offline learning and online learning.

Model Quality: Figure 1(c) and (d) display how the incorporation of new arrival data improves the model quality. In Figure 1(c), we adopt a popular metric named *perplexity* [16] (lower is better) to measure the model quality of CNN application. We measure the training model quality every 2 minutes in offline and online manners respectively. With more data fed into the model, both perplexities decrease, which means a better model quality. Figure 1(c) shows the overall perplexities of offline and online trainings are very close while online manner is not as stable as offline training. We then compare the model performance of online and offline training in overall prediction accuracy. This generally used metric quantifies the overall prediction performance of a classification algorithm (larger is better). Figure 1(d) shows that the model quality by offline training increases with more data get incorporated, and its quality trend to be stable. The model quality by online training outperforms offline training at the time points 10 and 40 while it is worse than offline training during other time periods. The reason is that only using new arrival data in the online learning approach cannot guarantee stable model quality compared to the offline learning approach. Although such naive data update strategy contributes to speed up the data training process, it may discard many data samples with vital information to improve the model quality.

III. System Design

In this section, we present iDlaLayer, a thin middleware layer between applications and backend deep learning frameworks that realizes efficient online model updating.

A. Architecture Overview

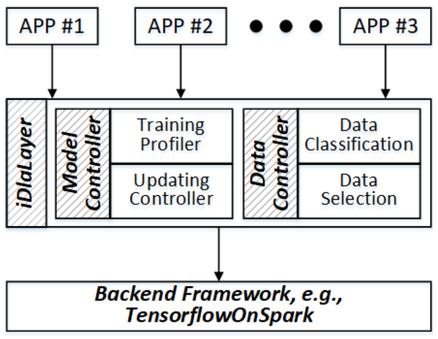


Fig. 2. Architecture of iDlaLayer.

Figure 2 provides an overview of iDlaLayer architecture. The critical contribution relies on a novel Data Life Aware model updating strategy (DLA) to online learning applications. DLA is capable of building better training data samples in terms of contribution made by data from different life stages and determining the right time to perform model updating with a lower training cost. More details on updating strategy and data life cycle would be discussed below. iDlaLayer mainly consists of two components as shown in Figure 2: **Model Controller** and **Data Controller**.

- **Model Controller** has two functional parts. The *training profiler* logs and profiles the training time for each application. We use linear regression approach [17] [18] to predict time consumed to train a model based on the size of sample data. The prediction result is then used as an estimation of future arrival data. The *updating controller* is responsible for deciding when to perform model updating and take the role of communicating with the data management component. Besides, it is the module where we implement DLA.
- **Data Controller** maintains and builds training data samples in terms of life stages. This component communicates with the updating controller to evaluate the contribution each data made to model updating and then use such information as an index determining data life stage among *Newborn*, *Mature*, *Preserve* and *Discard*. Moreover, the training data sample is built in this component based on the contributions to model updating made by data from various life stages.

B. Workflow and Implementation

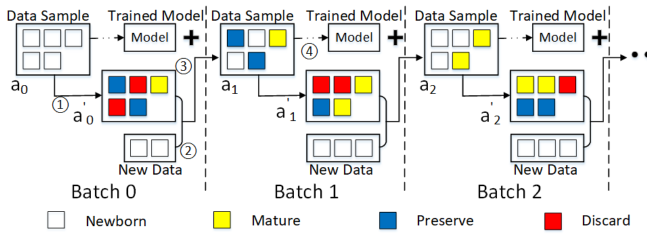


Fig. 3. Detailed Training Workflow of iDlaLayer.

Workflow: Combing the concepts described above, Figure 3 depicts the detailed workflow of our proposed architecture. Three training model stages are included: $Batch0$, $Batch1$ and $Batch2$. Specifically, we adopt white, yellow, blue and red to denote data block in *Newborn*, *Mature*, *Preserve* and *Discard* stages respectively. Data samples containing several data blocks are used to train the model. $Batch0$ is the first model training with the data sample a_0 that includes 5 *Newborn* data blocks (illustrated with white blocks). Since data blocks contribute variously to model quality, their data life stages are different when model training finished, which is shown by different colored blocks in $a_0^{'}$. Step ① demonstrates data life changes between a_0 and $a_0^{'}$. Comparing a_0 and $a_0^{'}$, red data blocks in *Discard* stage are replaced with new arrival data blocks and turns into $a_{i-1}^{'}$: 1 *Mature*, 2 *Preserve* and 2 *Discard* blocks. During the procedure of the model training with data sample a_0, some new data also arrived at the same time. If the current training finished and its training metrics are profiled, a new data sample a_1 (containing 2 *Newborn*, 1 *Mature* and 2 *Preserve* data blocks) for next training procedure $Batch1$ is supposed to be built (illustrated by Step ②). Reason behind this operation is that red blocks would contribute less for the coming model training, getting rid of those blocks brings more space and opportunities to new arrival data. Step ③ describes updating decision which is responsible for determining when to perform next training in terms of training cost. The next model training $Batch1$ is triggered with newly built a data sample and model from the previous training procedure. Furthermore, several performance metrics such as training cost, data contribution and "knob" parameter are profiled in Step ④. Similarly, $Batch1$ trains model and prepares data sample a_2 for next training stage $Batch2$. This training loop would continue until no more new data arrives or no improvement in model quality.

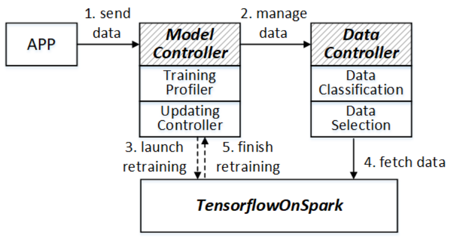


Fig. 4. The training workflow of iDlaLayer.

Implementation: We implement iDlaLayer with Java on top of TensorflowOnSpark. Figure 4 shows the workflow of iDlaLayer. (1) When a new data sample arrives, an application contacts its exclusive client process running in **Model Controller** to communicate with iDlaLayer and then data information is sent to **Data Controller** component. (2) *Data Classification* maintains the information of this dataset by setting and updating its life stage, and *Updating Controller* decides whether to make model updating. (3) Once the updating decision is made, iDlaLayer informs the backend framework–TensorflowOnSpark to perform model training. (4) Meanwhile, *Data Selection* is ready to send combined data samples to TensorflowOnSpark. (5) After the backend framework finishing retraining model with new data samples, it notifies **Model Controller**'s *Training Profiler* module to update the training time profile for that application. Then *Updating Controller* evaluates contributions the data samples have made as well

TABLE I
DATA LIFE STAGE DIVISIONS OF APPLICATIONS

Application	D	N	M	P
Pattern Recognition	0-5%	5-10%	10-36%	>36%
Classification Prediction	0-21%	21-35%	35-63%	>63%
Recommendation System	0-12%	12-21%	21-45%	>45%

D = Discard, N = Newborn, M = Mature, P = Preserve

as updates data's life stage. The updated model can finally be shipped to model serving systems for further inference requests.

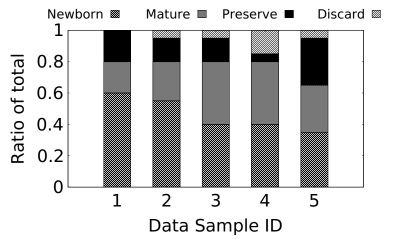
iDlaLayer uses a process *allocate_mem* to initialize memory space for maintaining data. A function named *serve_app* is launched in **Model Controller** component to handle training data samples for each application. When new data arrives, this process transfers data to **Data Controller** component by calling *reciv_data* function running in it. *devi_data* is responsible to decide and update data's life stages based on its contribution to model updating. Based on the proposed *Fuzzy Model*, we experimentally determine the data life stage ranges regarding to the distribution of each data's contribution to the model quality. Specifically, Table I displays classification standards of data life stages for the different applications we selected in our experiments, which is a trade-off between model quality and training time. We implement DLA in *Updating Controller* module determining when to perform model updating. Once an updating decision is made by *launch_update* in *Updating Controller*, building training data sample procedure starts to work by invoking *build_sample*. It is a lightweight process with quick response and prompt processing features to guarantee a solid service to backend systems. Simultaneously, the backend also gets updating notification and begins to fetch data sample from **Data Controller** component. After backend finishing model updating with new data, it recalls *log_profile* function in *Training Profiler* to update the training time cost by application and the contributions new data has made. Meanwhile, *devi_data* is informed by *log_profile* to update data life stages it maintains.

IV. EVALUATION

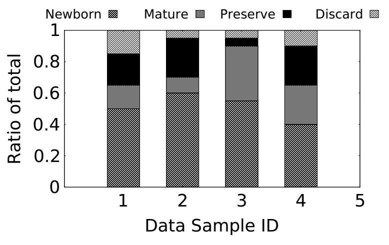
A. Experiment Setup

The evaluation testbed consists of 15 nodes, each of which has Intel Xeon(R) CPU E5-2630v4@2.20GHz x 20 and 64GB DDR3 RAM, running Ubuntu 16.04 LTS operating system with kernel version 4.0, Scala 2.10.0, and Hadoop YARN 2.8.0 for cluster management. One node serves as the *master*, and all the other 14 nodes serve as *slaves*. These nodes are connected with Gigabit Ethernet. We select three representative applications: the pattern recognition with MNIST dataset [14], the classification prediction based on Criteo [15] and the recommendation system with PageRank.

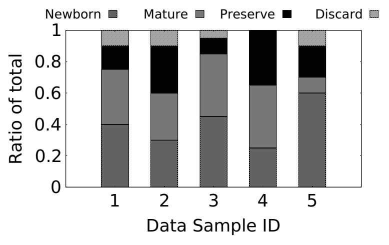
In our implementation-based experiments, we set up the scenario in which data are continuously generated and fed



(a) MNIST



(b) Criteo



(c) PageRank

Fig. 5. Ratio of data life stages on randomly chosen time points from MNIST, Criteo and PageRank.

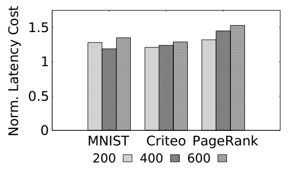
into the system. Each sample data has a timestamp in the trace so that they roughly follow the behavior of real-world data. To evaluate the effectiveness of iDlaLayer, we use an optimal offline training (Optimal) and a periodic update strategy (Periodic) as two baselines. Optimal is a simulation-based method that adopts a dynamic programming algorithm to update model. In this method, we assume that the backend notices future data arrival with which the best model updating strategy can be made. Periodic is a strategy that triggers model updating at intervals with a predefined period. The performance evaluation mainly focuses on (i) training cost, (ii) ratio of data life stages, (iii) model quality, and (iv) overhead.

B. *Effectiveness on Ratio of Data Life Cycle*

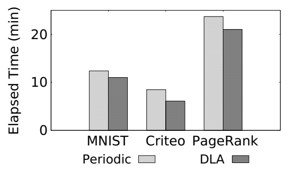
Figure 5 depicts data ratios of different life stages on five randomly time points from the three applications, i.e., MNIST, Criteo and PageRank. We use stacked histogram to denote the ratio of data life stages. Four patterned columns represent the four data life stages we defined above. At each time point, it is obvious that ratios of data life stages vary a lot since the solution to our objective function on finding better data combination ratios changes over time. There are a few conditions that a data life stage does not show up in the combinations e.g., the time point 1 of MNIST (Figure 5(a)) and the time point 4 in PageRank (Figure 5(c)). These cases illustrate the data in *Discard* stage makes no contributions to improving model performance so that they are not included in combined data samples. Figure 5 also displays that these combinations of data samples keep changing at runtime and each application shows unique behavior in terms of combined data samples. An effective ratio at a time point may not be valuable for the others. iDlaLayer is capable of building training data samples with data from different life stages in terms of how much contribution they could make to model updating.

C. *Effectiveness on Training Cost Reduction*

To illustrate performance differences in training cost, we compare our proposed solution with Optimal strategy. We conduct real trace-based simulations to demonstrate the performance difference in latency. Given the fact that the optimal training follows a linear pattern to the size of data samples, we run this experiment on 200, 400 and 600 data samples randomly selected from the datasets. Figure 6(a) depicts the



(a) Incorporation latency



(b) Incorporation elapsed time

Fig. 6. (a) displays data incorporation latency of DLA normalized by Optimal. (b) shows elapsed time from Periodic and iDlaLayer.

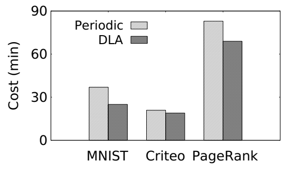


Fig. 7. Training cost.

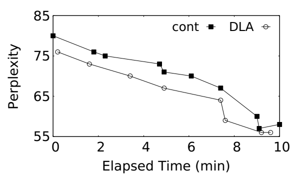
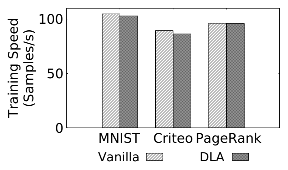


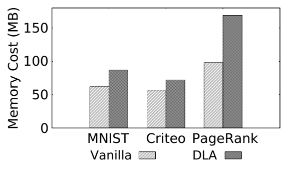
Fig. 8. Quality of MNIST.

data incorporation latency cost of our method, which is normalized by Optimal offline training strategy. By checking the results from those three applications, we observe that DLA also follows a linear relationship to the size of data sample. Moreover, our method performs very closely to Optimal offline training strategy in latency (which is about 1.32x on average).

We then compare Periodic updating and iDlaLayer deployed on a psychical cluster with 15 nodes. Figure 6(b) plots the performance (elapsed time in incorporating data) of Periodic and iDlaLayer. The improvements incurred by iDlaLayer are 11.3%, 28.2% and 15.2% for MNIST, Criteo and PageRank respectively. Criteo benefits more than the other two applications. The reason is iDlaLayer could maintain data based on its life stages with more useful data combined into data samples and Criteo demands more duplicated data in model training. The figure shows that PageRank requires more time in training model while our method still decreases its total cost. Figure 7 compares the training cost of Periodic update strategy and iDlaLayer. The training costs of all applications show a similar change trend at runtime. Our architecture achieves 10% (Criteo) less training cost on average and the maximum improvement reaches 32.5% (MNIST).



(a) Training speed



(b) Memory consumption

Fig. 9. The training speed and the memory consumption of Vanilla TensorflowOnSpark and iDlaLayer.

D. Effectiveness on Model Quality

Model quality ultimately decides how well a prediction could be achieved. Specifically, we choose MNIST application and compare its quality between continuous (*cont*) and iDlaLayer architectures. Model quality is measured by *perplexity* as we discussed in Section II. Figure 8 presents the result of a 10-minutes time window, where each dot represents the end of one model updating procedure. Overall, the perplexity of *cont* and iDlaLayer drops with the incorporation of new data. iDlaLayer's perplexity drops more than *cont* as it uses more beneficial data samples in model updating. Besides, it has fewer data points in the dotline comparing with *cont* because some retraining instances are abandoned and merged into others. Consequently, every model updating procedure in iDlaLayer covers more data and brings a better quality, which is about 5% improvement, as shown in Figure 8.

E. Overhead

To analyze the overhead of iDlaLayer, we use the vanilla TensorflowOnSpark as a baseline and conduct experiments to measure (i) training speed, which measures how many samples the system could process per second and (ii) memory consumption, which scales how much memory resource it takes to realize the retraining strategy. Figure 9(a) depicts the training speed comparison between the vanilla TensorflowOnSpark and iDlaLayer. iDlaLayer results in 1.8%, 3.5% and 0.7% slowdown on MNIST, Criteo and PageRank respectively. We can explain those slowdowns with DLA's strategies on combined data samples and model updating decisions, which take a short time before the training procedure begins. Overall, slowdown in training speed, which is 2% on average, does not significantly affect applications' performances. Instead, iDlaLayer decreases more time spent in finishing the whole applications. Figure 9(b) shows memory consumption of the vanilla TensorflowOnSpark and iDlaLayer. For all three applications, iDlaLayer consumes 28.7%, 20.8% and 42.1% more memory resource in running MNIST, Criteo and PageRank respectively. The average memory overhead introduced is about 30.5% (23 MB), which is negligible to the whole system-level memory resource. Considering the performance improvement achieved by DLA, such memory overhead is an acceptable trade-off between space and time consumption.

V. Related Works

Various deep learning frameworks are available to provide proper support for popular neural network algorithms, such as TensorFlow [19], Coooolll [9]. Recently, a new trend in deep learning is to realize distributed learning with a parameter server model. BigDL [20] allows end-users to build deep learning applications using a single unified data pipeline and the whole pipeline directly run on top of some existing deep learning systems in a distributed fashion. However, most of them perform well in offline training models while never explicitly supporting online updating strategies. Instead, users have to design customized training loops to manually realize online learning via continual or periodic approach, which is inefficient and cumbersome. iDlaLayer is orthogonal to these systems and complemental to them with supporting and implementing online DL applications.

Data streaming systems have been widely studied and increasingly used in real-world commercial solutions rather than pure academics. Differential Dataflow [21] is designed to process large volumes of streaming data efficiently. Naiad [22] automatically incrementalizes dataflow computations and is capable of building low-latency data-parallel iterative computation. However, Differential Dataflow [21] and Naiad [22] mainly focus on quickly react on larges input data and respond to any changes inside the data, and cannot respond to online learning cases that processing the same computation with different data sample in each iteration. iDlaLayer implements online learning features for stream-based workloads in dynamic environments. Similar to studies [23] [24] on latency and throughput in processing streaming data, our algorithms can help such systems achieve a balance between processing cost and latency.

Recently, many works focus on exploring deep learning model updating strategies. For example, Incremental&Decremental SVM [25] and Weighted-SVM [26] resort to re-engineer existing learning algorithms by developing incremental versions of the basic SVM and adjusting the training data sample weights in an SVM-specific manner, respectively. Velox [27] proposes a model updating strategy with combing online learning and statistical techniques. DLA is complementary to these works, and could potentially be applied in a system like Velox to support online learning.

VI. Conclusion

In this paper, we propose *iDlaLayer*, a middleware layer built on top of existing deep learning frameworks that streamlines the support and implementation of online learning applications. To achieve a better training model of describing data tendency in dynamic environments, we introduce DLA, a novel model updating strategy, which builds training data samples in terms of contributions from different data life stages in model training, and considers the training cost consumed in model updating. Our experimental results demonstrate that iDlaLayer reduces the overall elapsed time of MNIST, Criteo and PageRank by 11.3%, 28.2% and 15.2% compared to the popular periodic update strategy, respectively. It further achieves an average 20% decrease in training cost and brings about a 5% improvement in model quality against the traditional continuous training method.

References

[1] J. Gama, I. Žliobaitė, A. Bifet, M. Pechenizkiy, and A. Bouchachia, "A survey on concept drift adaptation," *ACM computing surveys (CSUR)*, vol. 46, no. 4, p. 44, 2014.

[2] L. Xiao, "Dual averaging methods for regularized stochastic learning and online optimization," *Journal of Machine Learning Research*, vol. 11, no. Oct, pp. 2543–2596, 2010.

[3] S. J. Prince and J. H. Elder, "Probabilistic linear discriminant analysis for inferences about identity," in *2007 IEEE 11th International Conference on Computer Vision*. IEEE, 2007, pp. 1–8.

[4] J. Mairal, F. Bach, J. Ponce, and G. Sapiro, "Online learning for matrix factorization and sparse coding," *Journal of Machine Learning Research*, vol. 11, no. Jan, pp. 19–60, 2010.

[5] B. Tamara, B. Nicholas, W. Andre, and W. Ashia, "Streaming variational bayes." NIPS, 2013.

[6] H. B. McMahan, G. Holt, D. Sculley, M. Young, D. Ebner, J. Grady, L. Nie, T. Phillips, E. Davydov, D. Golovin *et al.*, "Ad click prediction: a view from the trenches," in *Proceedings of the 19th ACM SIGKDD international conference on Knowledge discovery and data mining*. ACM, 2013, pp. 1222–1230.

[7] X. He, J. Pan, O. Jin, T. Xu, B. Liu, T. Xu, Y. Shi, A. Atallah, R. Herbrich, S. Bowers *et al.*, "Practical lessons from predicting clicks on ads at facebook," in *Proceedings of the Eighth International Workshop on Data Mining for Online Advertising*. ACM, 2014, pp. 1–9.

[8] C. A. Gomez-Uribe and N. Hunt, "The netflix recommender system: Algorithms, business value, and innovation," *ACM Transactions on Management Information Systems (TMIS)*, vol. 6, no. 4, p. 13, 2016.

[9] D. Tang, F. Wei, B. Qin, T. Liu, and M. Zhou, "Coooolll: A deep learning system for twitter sentiment classification," in *Proceedings of the 8th international workshop on semantic evaluation (SemEval 2014)*, 2014, pp. 208–212.

[10] L. Yang, J. Shi, B. Chern, and A. Feng, "Open sourcing tensorflowonspark: Distributed deep learning on big-data clusters," 2017.

[11] D. Crankshaw, X. Wang, G. Zhou, M. J. Franklin, J. E. Gonzalez, and I. Stoica, "Clipper: A low-latency online prediction serving system," in *14th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 17)*, 2017, pp. 613–627.

[12] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," in *Advances in neural information processing systems*, 2012, pp. 1097–1105.

[13] G. Hinton, L. Deng, D. Yu, G. Dahl, A.-r. Mohamed, N. Jaitly, A. Senior, V. Vanhoucke, P. Nguyen, B. Kingsbury *et al.*, "Deep neural networks for acoustic modeling in speech recognition," *IEEE Signal processing magazine*, vol. 29, 2012.

[14] L. Deng, "The mnist database of handwritten digit images for machine learning research [best of the web]," *IEEE Signal Processing Magazine*, vol. 29, no. 6, pp. 141–142, 2012.

[21] F. McSherry, D. G. Murray, R. Isaacs, and M. Isard, "Differential dataflow." in *CIDR*, 2013.

[15] "Google experiments with tensorflow on criteo dataset," https://labs.criteo.com/2017/02/google-experiments-tensorflow-criteo-dataset/, accessed: 2018-6-28.

[16] H. M. Wallach, I. Murray, R. Salakhutdinov, and D. Mimno, "Evaluation methods for topic models," in *Proceedings of the 26th annual international conference on machine learning*, 2009, pp. 1105–1112.

[17] P. Lama, S. Wang, X. Zhou, and D. Cheng, "Performance isolation of data-intensive scale-out applications in a multi-tenant cloud," in *2018 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, 2018, pp. 85–94.

[18] D. Yang, W. Rang, and D. Cheng, "Joint optimization of mapreduce scheduling and network policy in hierarchical clouds," in *Proceedings of the 47th International Conference on Parallel Processing*. ACM, 2018, p. 66.

[19] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard *et al.*, "Tensorflow: A system for large-scale machine learning," in *12th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 16)*, 2016, pp. 265–283.

[20] J. J. Dai, Y. Wang, X. Qiu, D. Ding, Y. Zhang, Y. Wang, X. Jia, C. L. Zhang, Y. Wan, Z. Li *et al.*, "Bigdl: A distributed deep learning framework for big data," in *Proceedings of the ACM Symposium on Cloud Computing*, 2019, pp. 50–60.

[22] D. G. Murray, F. McSherry, R. Isaacs, M. Isard, P. Barham, and M. Abadi, "Naiad: a timely dataflow system," in *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, 2013, pp. 439–455.

[23] M. H. Javed, X. Lu, and D. K. Panda, "Cutting the tail: designing high performance message brokers to reduce tail latencies in stream processing," in *2018 IEEE International Conference on Cluster Computing (CLUSTER)*. IEEE, 2018, pp. 223–233.

[24] I. Botan, R. Derakhshan, N. Dindar, L. Haas, R. J. Miller, and N. Tatbul, "Secret: a model for analysis of the execution semantics of stream processing systems," *Proceedings of the VLDB Endowment*, vol. 3, no. 1-2, pp. 232–243, 2010.

[25] G. Cauwenberghs and T. Poggio, "Incremental and decremental support vector machine learning," in *Advances in neural information processing systems*, 2001, pp. 409–415.

[26] R. Klinkenberg, "Learning drifting concepts: Example selection vs. example weighting," *Intelligent data analysis*, vol. 8, no. 3, pp. 281–300, 2004.

[27] D. Crankshaw, P. Bailis, J. E. Gonzalez, H. Li, Z. Zhang, M. J. Franklin, A. Ghodsi, and M. I. Jordan, "The missing piece in complex analytics: Low latency, scalable model management and serving with velox," *arXiv preprint arXiv:1409.3809*, 2014.