Datenbank Spektrum (2021) 21:203–212
https://doi.org/10.1007/s13222-021-00386-8

SCHWERPUNKTBEITRAG

Continuous Training and Deployment of Deep Learning Models

Ioannis Prapas[1] · Behrouz Derakhshan[2] · Alireza Rezaei Mahdiraji[2] · Volker Markl[1]

Received: 31 May 2021 / Accepted: 31 August 2021 / Published online: 11 November 2021

Abstract
Deep Learning (DL) has consistently surpassed other Machine Learning methods and achieved state-of-the-art performance in multiple cases. Several modern applications like financial and recommender systems require models that are constantly updated with fresh data. The prominent approach for keeping a DL model fresh is to trigger full retraining from scratch when enough new data are available. However, retraining large and complex DL models is time-consuming and compute-intensive. This makes full retraining costly, wasteful, and slow. In this paper, we present an approach to continuously train and deploy DL models. First, we enable continuous training through proactive training that combines samples of historical data with new streaming data. Second, we enable continuous deployment through gradient sparsification that allows us to send a small percentage of the model updates per training iteration. Our experimental results with LeNet5 on MNIST and modern DL models on CIFAR-10 show that proactive training keeps models fresh with comparable—if not superior—performance to full retraining at a fraction of the time. Combined with gradient sparsification, sparse proactive training enables very fast updates of a deployed model with arbitrarily large sparsity, reducing communication per iteration up to four orders of magnitude, with minimal—if any—losses in model quality. Sparse training, however, comes at a price; it incurs overhead on the training that depends on the size of the model and increases the training time by factors ranging from 1.25 to 3 in our experiments. Arguably, a small price to pay for successfully enabling the continuous training and deployment of large DL models.

Keywords Deep learning · Model deployment · Continuous training · Continuous deployment

1 Introduction

Deep Learning (DL) is a subfield of Machine Learning (ML), involving Deep Neural Network (DNN) models, which has shown huge success in recent years. It has dramatically improved the state-of-the-art in many fields, like speech recognition [1], computer vision [2], and natural language understanding [3]. This success is explained by the fact that the quality of DL models improves with increasing dataset sizes, due to their ability to learn representations directly from data.

However, DL still faces several challenges. First, DNN results are not easily interpretable [4]. Second, their optimization is not theoretically well understood, relying on non-convex optimization [5]. Third, they are resource-intensive, taking days or weeks to train on expensive GPU clusters. Training DNN models requires extremely large datasets and compute resources that are exponentially rising [6]. Fourth, DL models can be massive in size; recently, the GPT-3 [3] architecture featured a staggering 175 billion parameters.

The two last problems of DL (compute-intensive, massive model sizes) are accentuated in the era of the Internet of Things (IoT) when we are surrounded by sensory devices (e.g. smartphones, cameras, sensors) that collect and generate data streams continuously [7]. Although there is a growing need for continuously updated DL models, current systems support periodic full retraining of DL models

✉ Ioannis Prapas
iprapas@protonmail.com

Behrouz Derakhshan
behrouz.derakhshan@dfki.de

Alireza Rezaei Mahdiraji
alireza.rm@gmail.com

Volker Markl
volker.markl@tu-berlin.de

[1] Technische Universität Berlin, Berlin, Germany

[2] Deutsches Forschungszentrum für Künstliche Intelligenz, Berlin, Germany

when enough new data are available or the performance of models degrades [8]. We find this process to be wasteful and resulting in stale models. *Wasteful*, because it discards the previously learned model that has consumed considerable amounts of compute-resources. Even if the previously learned model is not discarded but used for warm-starting, it still needs several epochs to converge and might not give state-of-the-art generalization [9]. *Model staleness* is a direct consequence of continuously arriving data; by the time a model is retrained, enough new data may be available to trigger a new retraining. Periodical retraining time grows as the dataset size increases. The fast arrival of new data together with an increasingly slow retraining process are the recipe for stale models in production.

To overcome the staleness of DL models, we propose to continuously train the previously learned model with proactive training [8]; a strategy for continuous training that performs Stochastic Gradient Descent (SGD) iterations with batches formed by a combination of new data and samples of historical data. We continuously update the deployed DL model residing on a remote machine using the gradient updates obtained by proactive training. For large DL models, the transmission of gradients over the network can easily become a communication bottleneck and endanger the privacy of training data. We opt to adapt our continuous training approach with an idea from the distributed DL training domain to reduce the communication cost of model updates among different workers [10–12]. More specifically, we sparsify the gradients calculated during each training iteration, keeping an accumulated memory of the unused gradients of previous iterations. The sparse gradient vectors are not only used for training but also communicated to the remote machine that handles the model deployment. The sparsification significantly reduces the communication needed to deploy model changes (allowing mini-batch level deployment).

Our contributions are as follows. First, we adapt proactive training to enable continuous training of DL models (Sect. 3). Second, we extend it with gradient sparsification to reduce the deployment cost and enable continuous deployment of DL models (Sect. 4). Finally, we perform an extensive experimental analysis on the impact of both proactive training and sparsification on training time, deployment cost, and model quality (Sect. 5).

2 Related Work

We divide the related work into two groups: model training when new data arrives, and deployment of DNNs models.

The standard approach for DL model training is to gather a big dataset and train the DL model over multiple epochs (passes over the dataset), while reshuffling the dataset before each epoch. Mini-batch SGD and its variants have several properties that are suitable for large scale datasets [13, 14] and are the de-facto optimization methods for DL training. After the initial training, when enough new data have become available or the model's performance has degraded the training process is restarted from scratch to prevent excessive model staleness. This is known as full retraining or batch learning. As opposed to batch learning, online learning keeps models fresh incrementally as new data arrives. Online learning for DL has received limited attention [15, 16], but should be highlighted more, as IoT applications gain more ground and produce streams of data [17]. In contrast to batch learning methods with expensive retraining cost whenever new training data arrive, online learning performs updates only based on the new training data. This makes online learning highly scalable and suitable for large-scale applications. However, in many cases, to converge to a good model, there is merit to continue training on historical data and not only take into account new data.

In some cases, the previously trained model still has some value [18] and this is why platforms like Tensorflow Extended (TFX) [19, 20] allow for the training to be started with the parameters of the previously trained model, a process called *warm-starting*. Recent work by Derakhshan et al. [8] shows that when it comes to training ML models, full retraining is not always required and online learning is not good enough. They propose to periodically trigger *proactive training*, a process that combines new data with samples of historical data into mini-batches for SGD-based optimization. This continuous training approach achieves superior quality to online training. When compared to full retraining, it achieves similar model quality, while reducing the total data processing and model training time by an order of magnitude. While Derakhshan et al. use proactive training to continuously train models with convex loss functions, such as SVMs and Logistic Regression models, it has never been tested with DL models until this study. We define and evaluate the proactive training for continuously training DL models. In a different setting than ours, this problem is similar to continual learning [21–24]. In that framework, a model is learning tasks one after another with the goal to learn the last task without *catastrophic forgetting* [25] that is losing the ability to perform on previous tasks.

After training, one must deploy the DL model and make it available for inference. Continuously triggering proactive training implies the continuous deployment of DL models. However, modern DL models can have a huge number of parameters and sizes ranging from several hundred MBs to several GBs. Naively deploying such big models continuously is not feasible because of the intractable communication cost. Work in gradient compression offers a promising direction for DL deployment. In this area, we find approaches that are meant to minimize the communication cost of parallel or distributed training. Instead of looking to compress the model, gradient compression is looking to minimize the model changes at each optimization step. Hogwild [26] allows workers to send gradients asynchronously without any sacrifice in convergence rates. Alistarh et al. [27] propose a quantized extension of SGD, called Quantized SGD (QSGD), which provides a trade-off between convergence rate and compression. Aji et al. [12] sparsify gradient updates by only considering the top-k components in every iteration and accumulating smaller gradients. Deep gradient compression [10] shows that extreme sparsification reduces communication costs by 99.9% and works well with modern deep learning architectures. Koloskova et al. [28] use it to perform decentralized training under arbitrarily large sparsification. Stich et al. [11] prove its good convergence guarantees provided that a memory accumulates not updated gradients. To the best of our knowledge, our work is the first to consider such gradient compression schemes for continuous deployment of DL models without any loss in model quality.

3 Continuous Training of DL Models

In this section, we describe proactive training [8] as a method that enables the continuous training for DL models. In proactive training, an ML model is updated using mini-batch SGD, where mini-batches are formed by combining new data with samples of historical data. After the training, new data become part of the historical dataset.

The proactive training initiates a mini-batch training of size b after a trigger condition on the data stream is met. Assuming the number of new elements satisfying the trigger condition is t (also called *trigger size*), the proactive training constructs a mini-batch of size b including the t new elements and $(b - t)$ elements sampled from the historical data. These mini-batches are used to calculate SGD gradient updates. At the end of a training iteration, the t new elements become part of the historical dataset and are subject to be sampled in future iterations. This procedure is summarized in lines 1, 2, and 9 of Algorithm 1 with a regular SGD update in between lines 2 and 9.

In this setting, a proactive training iteration is triggered after t new elements arrive from the data stream. By controlling the trigger size, the proactive training provides a balance between online gradient-based optimization and mini-batch SGD:

- A trigger size equal to zero ($t = 0$) is equivalent to mini-batch SGD.
- A trigger size equal to the batch size ($t = b$) is equivalent to mini-batch online gradient descent.

It is important to consider the following points when using proactive training. First, proactive training induces a bias towards historical data, as new data are expected to be selected fewer times for training. Second, the independent and identically distributed data assumption [29] can break when we use elements from the stream as they arrive. Third, while simple mini-batch SGD is done in epochs with contiguous access over the dataset, proactive training assumes sampling from large datasets, breaking the contiguous access. This does not pose a big challenge, as it can be solved through smart materialization [8], but still remains an element to consider.

4 Continuous Deployment of DL Models

After training, a DNN is typically deployed in an environment where it will serve prediction queries. Continuous training, as described in Sect. 3, implies continuous deployment, which in our case means transferring the gradient updates of the DNN's parameters across the network after every mini-batch update. This incurs a huge deployment cost as modern DNNs can have millions if not billions of parameters. The *deployment cost* is decomposed as the sum of the *communication* cost and the *loading cost*. The communication cost is the time it takes to send the model's parameters to the deployment server. The loading cost is the time it takes to update the model given the new parameters.

Typically, the communication cost is much larger than the loading cost. Thus, in order to reduce the deployment cost, our work focuses on reducing the communication cost of sending model changes at every proactive training update. This problem has successfully been addressed for reducing the communication cost of distributed training. Using sparse SGD with memory [11], at every iteration only k out of N ($k \ll N$) total gradients are selected for updating the model while the rest are kept accumulating in memory. Provided that the gradients are selected with appropriate operators, sparse SGD offers the same convergence rate as regular SGD [11]. Assume layer l of the DL model has N parameters and integer k is given ($k \ll N$). We apply the following two sparsification operators on the parameters of all layers of a given DL model:

- Random-k: Selects k out of N parameters uniformly at random.
- *Top-k*: Selects k out of N parameters with the largest absolute value.

To facilitate continuous deployment, we extend proactive training to perform sparse updates at every iteration. This means at every training iteration, we update only a small constant percentage of the total model parameters. Algorithm 1 shows how the extended proactive training works.

The algorithm takes as input a dataset *D*, the model's parameters θ, the learning rate α, and the mini-batch size *b*. Moreover, it receives the selection operator *selectork* (*randomk* or *topk*) and the memory of gradients which keeps accumulated unused gradients from past iterations. We compute the gradients *g* using mini-batch SGD (Line 3) and add residual gradients of the past iterations (*m*grad) to it (Line 4). Then, we apply the selection operator *selectork* and selects only *k* gradients to be used for updating the model (Line 5). We update the parameters of the local model in the machine that performs the training (Line 6). We send the *k* gradients over the network to update the deployed model (Line 7). We update the residual of the gradients (Line 8) and add the *t* data points to the historical dataset *D* (Line 9).

Algorithm 1: Continuous training and deployment iteration

```
Input: Selection Operator $selector_k$, trigger size $t$, Learning
       Rate $\alpha$, Mini Batch Size $b$
Data: Dataset D, Stream s, Params $\theta$, Memory $m_{grad}$
Result: Params $\theta$, Dataset D, Memory $m_{grad}$
1 Fetch $t$ new labeled data points from the stream
2 Sample $b - t$ data points from $D$
3 $g = \frac{1}{b} \sum_{i=1}^{b} \frac{\partial}{\partial \theta} L(f(x_i, \theta), y_i)$
4 $g = g + m_{grad}$
5 $g_{sparse} = selector_k(g)$
6 $\theta = \theta - \alpha * g_{sparse}$
7 Send $g_{sparse}$ to update deployed model
8 $m_{grad} = g - g_{sparse}$
9 Append $t$ new data points to $D$
```

This deployment method, however, is not constrained to run only in the case that we perform proactive training. With large data streams, one can imagine directly using online mini-batch SGD, as it has the nice property that it follows the gradient of the true generalization error. Sparse continuous deployment can effectively be used in that case to continuously deploy a model that is learned in an online fashion.

4.1 Hyper-parameters of Sparse Continuous Training

There are two important hyper-parameters of the Sparse Continuous Deployment. The first hyper-parameter is the *selector* (*topk* vs *randomk*). In theory, both the *randomk* and the *topk* can achieve the same convergence rate. However, in practice, *topk* has been shown to achieve better performance [11], as the relative gradient magnitude is thought to provide a simple heuristics for gradient importance [10]. The second hyper-parameter is the *sparsification ratio*, i.e., the percentage of the enforced sparsity. Previous work has shown that sparse SGD with memory can converge fast under arbitrarily large sparsification, with Deep Compression [10] achieving good convergence reducing 99.9% of the communicated gradients.

We show the impact of these hyper-parameters in the evaluation section.

5 Evaluation

We conduct experiments to compare the three training approaches, i.e., full, online, and proactive in terms of quality and training time. Furthermore, we investigate the impact of trigger size in proactive training. Lastly, we study the impact of sparsification on model quality and deployment overhead.

To meet this goal, we simulate a scenario in which an initial dataset is available for the initial training and the rest of the data become available in a streaming fashion. The initial training happens on the initial dataset for a constant number of epochs.

Then, we compare the different training approaches:

- For the *full retraining* approach, a full retraining is triggered periodically whenever *c* new elements are available. Each retraining is restarted from scratch and executed for *e* epochs, like the initial training.
- For the *online training* approach, the model is warm-started with the generated model from the initial training. An online training iteration is triggered once *b* new elements are available in order to perform a mini-batch SGD iteration.
- For the *proactive training* approach, similar to online training, the model is warm-started with the generated model from the initial training. As presented in Sect. 3, a proactive training iteration is triggered once *t* new elements are available. We investigate the impact of the trigger size *t* on training time and model accuracy. In addition, we use sparsity and examine how the sparsification ratio and the choice of the selection operator affect the overhead training and the model quality of the deployed model.

To evaluate and compare these approaches we use the **prequential evaluation** [30], which is a common method to evaluate ML algorithms on data streams. In prequential evaluation, every example in the stream is first used to test the model, and then to train it.

5.1 Setup

5.1.1 Hardware & Software.

We run all experiments on a server with an Intel Xeon E7 with 128 GB of main memory and an NVIDIA TESLA GPU K40 with CUDA 10.2. The code for training and using DL models leverages the GPU and has been written in Python 3.5, using the PyTorch [31] framework (version 1.2.0). The state-of-the-art DNN architectures that are used in the experiments have been imported from the torchvision [32] library (version 0.4.0).

5.1.2 Datasets.

We run our experiments on two common computer vision datasets for benchmarking DL models, namely MNIST [33] and CIFAR-10 [34]. We present only the main results of the MNIST experiments as they were initially used to validate our methods. The CIFAR-10 experiments are then given in detail.

5.1.3 Models.

For the MNIST experiments, we use the LeNet-5 [35] architecture, which represents a seminal CNN architecture. For the CIFAR-10 experiments, we use modern DL models with different sizes and features. Mobilenet_v2 [36] is chosen as a compressed DL architecture with around 2 million parameters that is meant to run on mobile devices. We are interested to see the effect of continuous training and its sparse variants for such a compact model. Resnet18 [37] is chosen as a medium-sized model with around 11 million parameters that includes many state-of-the-art (SOTA) features, such as batch normalization and skip connections. Resnet50 [37] is chosen as a deeper alternative of Resnet18 with slightly more than 23 million parameters. Densenet161 [2] is chosen as a very deep SOTA model with around 26.5 million parameters that is in a different family than the residual networks selected. We start the training for each of the models with pre-trained weights on Imagenet [38] in order to reduce the time needed to converge to good solutions. This underestimates the time needed for the full retraining in the general case, but allows to iterate over our experiments much faster.

5.1.4 Hyper-parameters.

This work proposes a general framework for continuously training and deploying DL models. To strengthen the generality of our work, we choose to refrain from rigorous hyper-parameter tuning. For all of our experiments, we use the same training configuration:

- We use ADAM optimizer [39] with its default parameters (learning rate 0.001, β_1 0.9 and β_2 0.999).
- For the proactive and online training approaches when we warm-start the model, we also warm-start the learned parameters of the ADAM optimizer.
- We set the batch size to 128, which has successfully been used to obtain SOTA performance on the CIFAR-10 dataset (which we extensively use in our experiments) for several DL models [40].

5.1.5 Evaluation metrics.

We use the prequential evaluation technique and report the total training time and the cumulative prequential accuracy for each one of the experiments. We also capture the number of parameters that must be transferred at each case as a proxy for the communication cost.

5.2 Early experiments on MNIST

To initially validate our approach, we started with some early experiments on a simple dataset (MNIST) and a simple DL model (LeNet-5). Compared to full retraining and online training, we found proactive training to achieve the best prequential evaluation with data arriving in a streaming fashion. Controlling the trigger size allows proactive training iterations to be triggered $\frac{\text{batch size}}{\text{trigger size}}$ times more often than online training. Even when using a trigger size that triggers proactive training 32× more than online training, proactive training still needs a fraction of the time of full retraining. Meanwhile, sparse training allows us to achieve comparable performance to the non-sparse variant under a large sparsification ratio, i.e., transferring only about 0.01% of the total parameters per iteration. We identify that sparse training using the *randomk* selector is not stable at the beginning of the training, confirming results from Lin et al. [10] that the *topk* selector offers a good proxy for the importance of gradient updates.

We refer the reader to the supplementary materials (Online Resource 1) for a detailed analysis of the MNIST experiments.

5.3 DL models on CIFAR-10

We conduct experiments on CIFAR-10 [34], which is a more complex dataset than MNIST, and train on it several SOTA DL models: mobilenet_v2 [36], resnet18 [37], resnet50 [37], and densenet161 [2].

The first 10,000 examples are used for the **initial training** which is done for 25 epochs. In order to achieve a good performance in a reasonable time, we warm-start all the models with weights of pre-trained models on Imagenet [38]. **Full retraining** is triggered every 10,000 new data points that become available and is done for 25 epochs starting from weights of pre-trained models on Imagenet.

Online training triggers one mini-batch SGD iteration once mini-batch size ($b = 128$) new elements are available.

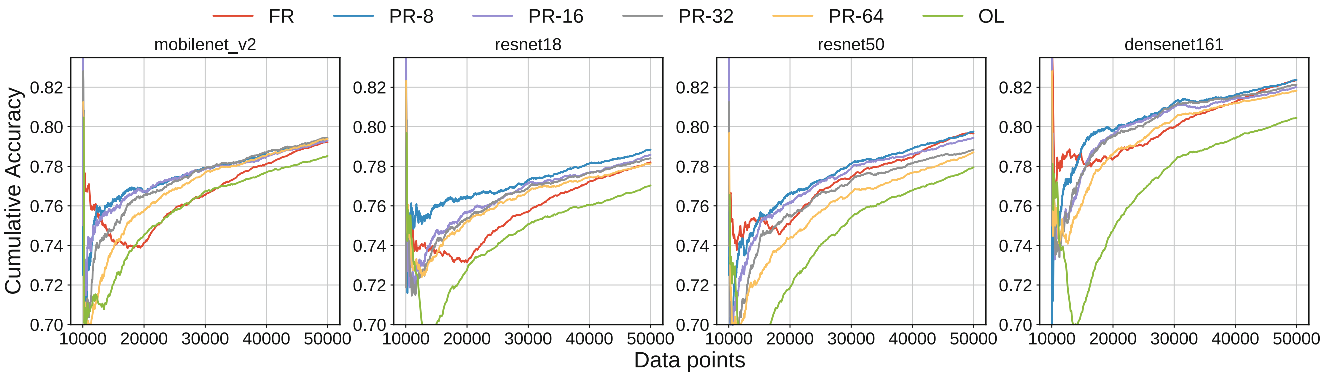

Fig. 1 Prequential evaluation for SOTA DL models on CIFAR. Comparison between full retraining (*FR*), online training (*OL*) and proactive training (*PR-t lines* for trigger size t = 64, 32, 16, 8)

According to the prequential evaluation, each mini-batch is first used for evaluation and then to train the model. No data point is revisited in online training.

Proactive training triggers one mini-batch SGD iteration once trigger size t new elements are available. According to the prequential evaluation, the new elements are first used for evaluation and then to train the model. After new elements are used for a first training iteration, they become part of the historical dataset and are subject to being sampled for future training iterations. For proactive training we experiment with i) different trigger sizes (t = 64, 32, 16, 8), ii) and different sparse selection ratios (1%, 0.1%, 0.01%) only using the sparse selector *topk* that proved more promising in the early MNIST experiments.

5.4 Full Retraining vs Online Training vs Proactive Training

Figure 1 shows the results of the prequential evaluation for each model and training approach (full retraining, online, proactive). Table 1 shows the corresponding total training times.

The online training is the fastest method and the proactive training finishes approximately $\frac{\text{batch size}}{\text{trigger size}}$ more slowly than online. Full retraining is the slowest method, needing around 3 times more training time. Concerning the prequential evaluation, we see that generally, online training is the worst method, followed by full retraining and proactive training which offers the best approach with increasing performance as more iterations are triggered (smaller trigger size).

These are the general trends, but there is more info hidden in the details. For all models, it is evident that at the very beginning of the online and proactive training they perform worse than the initial model. Online training takes the longest to recuperate. Proactive training recuperates faster when the trigger size is smaller. For mobilenet_v2, there is no clear winner among the proactive training approaches, with respect to the trigger size. Until the 30,000th point, lower trigger size is better, but then all methods seem to converge at a performance slightly better than full retraining. Except for resnet18, where we see that the proactive training outperforms the full retraining approach, we do not see large gaps in final cumulative accuracy between proactive training and full retraining. More specifically, for densenet161, resnet50, and mobilenet_v2, we see the proactive training performing better after stabilizing at the beginning of the training, but then converging to a comparable cumulative accuracy with full retraining.

5.5 Sparse Continuous Deployment

Figure 2 shows the results of the prequential evaluation for each model with sparsification. We use the *topk* sparse selector with selection ratios 1%, 0.1%, and 0.01% always using proactive training with trigger size equal to 8. The left part of Table 2 displays the average numbers of parameters changed per iteration for each one of the selection approaches and the corresponding non-sparse variant. The right part of Table 2 displays the corresponding total training times together with the full retraining and non-sparse proactive training for comparison.

Table 1 Total training time in seconds for full retraining (*FR* column), online training (*OL* column) and proactive training (*PRt* columns for trigger size t = 64, 32, 16, 8)

Model	*FR*	*OL*	*PR* 64	*PR* 32	*PR* 16	*PR* 8
mobilenet_v2	1,381.2	31.6	57.6	129.2	214.9	442.2
resnet18	493.8	13.1	21.1	42.1	94.2	169.8
resnet50	3,672.4	65.7	131.7	261.5	524.6	1,042.5
densenet161	6,864.4	122.1	242.4	486.4	952.1	1,923.3

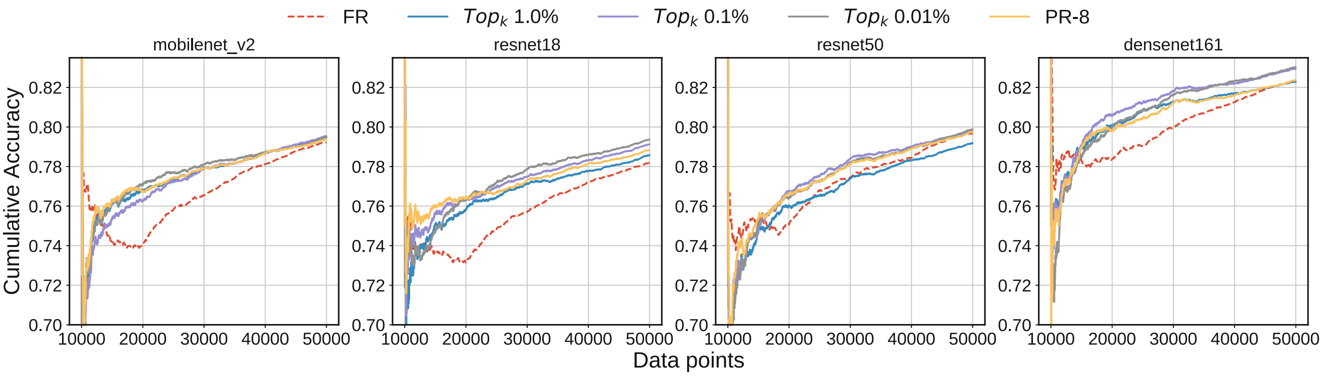

Fig. 2 Prequential evaluation for SOTA DL models on CIFAR. Comparison between full retraining (*FR*), and proactive training with a trigger size of 8 (*PR-8*) and various *topk* selection ratios (*Topk* for k = 1%, 0.1%, 0.01%)

Table 2 Left: Average number of parameters changed per iteration for the non-sparse variants (full retraining (*FR*) and non-sparse proactive training (*PR* 8)) and with varying *top*k selection ratios ("*topk*%" columns for k = 0.01, 0.1, 1). Right: Total training time in seconds for full retraining (*FR* column), non-sparse proactive training (*PR*8 column) and with various *topk* selection ratios (*topk*% columns for k = 0.01, 0.1, 1)

	Avg number of params changed per iteration				Total training time (sec)				
DL Model	*FR*/*PR*8	*top* 1.0%	*top* 0.1%	*top*0.01%	*FR*	*PR* 8	*top* 0.01%	*top* 0.1%	*top* 1.0%
mobilenet_v2	2,110,358	22,305	2,287	333	1,381.2	442.2	560.1	679	726.9
resnet18	4,068,411	111,800	11,201	1,146	493.8	169.8	288.3	279.2	305
resnet50	15,810,578	235,192	23,567	2,425	3,672.4	1,042.5	3,001.1	3,159.1	3,266.5
densenet161	15,991,847	264,641	26,498	2,886	6,864.4	1,923.3	3,870.3	4,032	4,154.2

With respect to the training time, it is evident that the *topk* selector induces a non-negligible overhead in the training process that depends on the size of the model. For mobilenet_v2 and resnet18, the sparse training times are on average larger by a factor of 1.48 and 1.7 respectively compared to their non-sparse variant. Sparse training on mobilenet_v2 has the largest variance among different sparse selection ratios with the overhead becoming slightly smaller when fewer parameters are selected for all the networks. For resnet50 and densenet161, the sparse training times are larger than the non-sparse variant by a factor of about 2 and 3, respectively. While the two networks have about the same number of parameters, the effect of running the *topk* selection is much bigger for resnet50. This is explained by the fact that resnet50 has more parameters per layer. Still, even with this overhead, the training time of sparse proactive training remains well below the total training time of full retraining. However, this overhead is deemed negligible as sparse training reduces the deployment cost by orders of magnitude. For exact numbers, the reader is referred to Table 2.

In general, sparse variants of proactive training follow closely the cumulative accuracy curve of the non-sparse variant. An exception to this is the sparse proactive training of resnet50 (Fig. 2) with a 1% selection ratio, which falls behind all the other approaches. In this set of experiments, we notice that greater sparsification values (fewer gradients used per iteration) lead to better prequential performance, even better than the non-sparse variant, forming a larger gap with the full retraining. We believe that this phenomenon is the result of the *topk* operator selecting the most "important" gradients per iteration, ignoring non-important ones. As a result, the sparsification provides some kind of *regularizing effect* in the training process.

As expected, Table 2 shows that with selection ratios of 1.0% 0.1%, 1.0% and 0.1% of the gradients per layer are changed per iteration accordingly. However, we see that for selection of 0.01%, the number of selected parameters is slightly larger than 0.01% of the total parameters. This difference is due to the fact that our implementation selects at least one parameter per layer, which accounts for this difference for layers that have less than 10,000 parameters. We discuss in detail the effect of the sparsification on the deployment cost in Sect. 5.7.

5.6 Final model quality

Having seen the superior performance of continuous training in a streaming setting, we would like to quantify the final model quality that each approach achieves. To this end, we measure the accuracy of each final model on the reserved test set of 10,000 images. We compare the models after the full retraining which uses the whole dataset (*FR*whole) and the last full retraining (*FR*last) which has not seen the last 10,000 elements, with the models after the end of online training and proactive training with a trigger

Table 3 Test accuracy of the final models for full retraining with the whole dataset (*FR*full), the last full retraining (*FR*last) that has been used for prequential evaluation, online training (OL) and proactive training with trigger size 8 without sparsity (*PR*8 column) and with different levels of sparsification using the *topk* sparse selector (*Topk%* columns with k = 1, 0.1, 0.01). Highest accuracy achieved for each model is shown in bold, *FR*whole ignored as it is considered the upper limit

DL model	*FR* whole	*FR* last	OL	*PR* 8	*top* 1.0%	*top* 0.1%	*top* 0.01%
mobilenet_v2	0.8275	0.8173	0.7995	0.8129	0.8149	**0.8196**	0.8087
resnet18	0.8232	0.8047	0.7999	0.8041	0.8003	0.8081	**0.8135**
resnet50	0.8405	0.8217	0.8146	0.8216	0.8211	0.8224	**0.8273**
densenet161	0.8581	**0.849**	0.8323	0.8428	0.8413	0.84	0.8464

size of 8 with and without sparsity for different *topk* selection ratios (1%, 0.1%, and 0.01%). Table 3 shows the results.

In all cases, full retraining that has access to the whole dataset (*FR*whole) achieves the best performance, and is considered an upper limit for our methods. We see that non-sparse proactive training consistently achieves comparable but slightly worse accuracy than the last full retraining, with online training providing the worst final model quality in all the cases. Sparse proactive training with sparse selection 1.0% is generally very close to its non-sparse variant, except for the case of resnet18 where it falls by about 0.04, achieving an accuracy closer to online training. Higher sparsification (*top*0.1% and *top*0.01%) training outperforms not only the non-sparse variant in most cases but also the full retraining with all models except for densenet161. This is probably due to the regularizing effect of the *topk* sparse selector as we described in the previous section. Among the sparse variants, the one with the most extreme sparsification (*top*0.01%) consistently outperforms the others, except for mobilenet_v2, where the sparse training with selection ratio 0.1% is the best. We suspect that this is due to the compactness of mobilenet_v2, which makes the sparsification above a certain threshold converge slower. It is out of the scope of this study to compare the differences in accuracy between the different models.

In short, these experiments confirm that the model qualities achieved by proactive training are comparable, if not superior to the ones we get from the last full retraining. When comparing with full retraining that has access to the whole dataset, we should consider it as the upper limit in terms of performance, but keep in mind that it will result in stale models in production most of the time, as in real streaming scenarios there is no time to do multiple passes on newly arrived data.

5.7 Findings

Our experiments find the sparse proactive training method to be suitable for continuous training and deployment. In all of our experiments, online training provides the worst prequential accuracy at the fastest training time. The proactive training approach consistently achieves comparable or superior performance than full retraining at a fraction of the time. The *topk* sparse selector can keep a model updated with sparse proactive training. Sparse proactive training follows closely the prequential curve of the non-sparse variant, reducing the number of changed parameters per iteration by 10,000×; thus, enabling the continuous communication of the gradients to a deployed model.

At the beginning of the prequential accuracy curves, both online and proactive training perform worse than the initially trained model. The problem can be attributed to the warm-started values of the ADAM optimizer. It can be fixed by allowing for a warmup period before deploying updated models in place of the initial model.

We now analyze how much the sparse selector *topk* reduces the deployment cost for densenet161, the largest model in our experiments. For the sake of simplicity, we assume that the total training time using the *topk* selector is 4000 s for all the selection ratios (the actual training times are 3870 s, 4032 s, and 4154 s for selection ratios 1.0%, 0.1%, and 0.01%, respectively). With a trigger size of 8, it performs $\frac{40{,}000}{8} = 5{,}000$ iterations. Therefore, a training iteration is performed roughly every $\frac{\text{training\_time}}{\#\text{ iterations}} \simeq \frac{4{,}000\,\text{s}}{5{,}000} = 800\,\text{ms}$. Assuming each parameter is a 32 bit float, we compare the bandwidth consumed when sending parameters equal to the number of the selected gradients by *topk* and its non-sparse variant:

- The non-sparse proactive training deploys a model (i.e., transferring all the parameters) about every 800ms, consuming a bandwidth of

$$\frac{26{,}494{,}090 * 32\ \text{bits}}{800\,\text{ms}} = 132.47\,\text{MBps}.$$

- The *topk* 1.0% transfers 618 parameters plus their indices, which we assume are 32 bit each, every 800 ms. This consumes a bandwidth of

$$\frac{2 * 264{,}641 * 32\ \text{bits}}{800\,\text{ms}} \simeq 2.65\,\text{MBps}.$$

- The *topk* 0.1% transfers 67 parameters plus the indices every 800ms, consuming a bandwidth of

$$\frac{2 * 26{,}498 * 32\ \text{bits}}{800\,\text{ms}} = 264.98\,\text{kBps}.$$

- The *topk* 0.01% transfers 13 parameters plus indices every 800ms, consuming a bandwidth of

$$\frac{2 * 2{,}886 * 32 \text{ bits}}{800\,\text{ms}} = 28.86\,\text{kBps}.$$

We see that for a DL model like densenet, the bandwidth needed for naive continuous deployment is 132.47 MBps, which would be hard to reliably maintain in real-world applications. It would also consume much of the bandwidth needed to receive and answer prediction queries. Sparsification can greatly reduce the communication cost of transferring model updates over the network to a mere 28.86 kBps without any loss in model quality. In reality, this cost can be much higher, because we use a slow GPU (Nvidia K40) by modern standards. Nvidia K40 has a theoretical FP32 performance of 5.046 TFLOPS while a more modern NVIDIA GeForce RTX 2080 Ti has a theoretical FP32 performance of 13.45 TFLOPS[1].

A counter-intuitive finding is that at times sparse training has achieved better prequential performance than its non-sparse variant. We believe that large models are highly redundant and the *topk* operator manages to select the less redundant changes per iteration, ignoring the non-important ones. In that way, sparsification provides a "regularizing effect" in the training process.

Sparse training is not all about advantages; there is a price to pay when using it. More specifically, the *topk* sparse selection incurs a non-negligible overhead that increases the proactive training time by 2× and 3× times for the larger models that we experimented with. However, sparsification reduces the deployment cost by orders of magnitude rendering its overhead on training cost negligible in end-to-end applications that comprise of training and deployment. Furthermore, we can improve the overhead of the sparsification by using fast ordered data structures (such as max-heaps or B-trees) that makes the *topk* operator more performant. Note that even with the extra overhead of the sparsification, the training times of sparse proactive training remain well below the full retraining. Although it does not alter the conclusions of our analysis, we should note that our reported full retraining times are wildly underestimated since we train the models for 25 epochs, instead of a few hundreds of epochs that the models require to achieve SOTA performance. For example, Densenets need 300 epochs to achieve SOTA performance on CIFAR-10 [2]. However, to achieve better accuracy in that many epochs one needs to rigorously tune the training hyper-parameters and learning rate schedules, which we refrained from doing in this work for all of the approaches for our simple choice of the hyperparameters.

6 Conclusion

Our work proposes to continuously train DL models with proactive training, as soon as new training data become available. Meanwhile, we enable the continuous deployment of very large DL models borrowing ideas from distributed DL training to sparsify weight updates per iteration in order to reduce the deployment cost. We reveal a regularizing effect of sparse training that at times allows achieving a better model quality than the non-sparse training variant. More importantly, it enables the continuous deployment of very large models and opens a new avenue for continuous DL model training and deployment in streaming settings.

In the future, we plan to investigate the following topics. First, we plan to study the impact of learning rate scheduling and more complex trigger conditions for the continuous training in order to compare against state-of-the-art performance. Second, we want to deeply explore the reasons for the instability at the very beginning of the continuous training, which in this study we attributed to the warm-up of the ADAM optimizer. Third, we plan to analyze real deployment scenarios and test the continuous deployment approach under unstable network connection with missing, delayed or corrupt model changes. Fourth, we want to test our methods with more datasets, and different flavors of DL models (e.g., Generative Adversarial Networks, Recurrent Networks), applied in a variety of domains(e.g., audio recognition, language modelling, time-series understanding). Fifth, we aim to test our methods on datasets that exhibit some concept drift, where we find that methods for continuous training and deployment are mostly needed.

Acknowledgements This work was funded by the German Ministry for Education and Research as BIFOLD—Berlin Institute for the Foundations of Learning and Data (ref. 01IS18025A and ref. 01IS18037A) and German Federal Ministry for Economic Affairs and Energy, Project "ExDRa" (01MD19002B).

Funding Open Access funding enabled and organized by Projekt DEAL.

[1] https://www.techpowerup.com/gpu-specs.

References

1. Hinton G, Deng L, Yu D, Dahl GE, Mohamed A, Jaitly N, Senior A, Vanhoucke V, Nguyen P, Sainath TN et al (2012) IEEE Signal Process Mag 29(6):82
2. Huang G, Liu Z, Weinberger KQ, van der Maaten L (2018) arXiv preprint. arXiv:1608.06993 1608
3. Brown TB, Mann B, Ryder N, Subbiah M, Kaplan J, Dhariwal P, Neelakantan A, Shyam P, Sastry G, Askell A et al (2020) arXiv preprint. arXiv:2005.14165
4. Samek W, Montavon G, Lapuschkin S, Anders CJ, Müller KR (2020) arXiv. preprint arXiv:2003.07631
5. LeCun Y (2007) In: 2007 NIPS workshop on Efficient Learning Vancouver, December. vol 7 (Citeseer)
6. Amodei D, Hernandez D (2018) https://blog.openai.com/ai-and-compute. Accessed: 1 Jun 2021
7. Zeuch S, Chaudhary A, Del Monte B, Gavriilidis H, Giouroukis D, Grulich PM, Breß S, Traub J, Markl V (2019) arXiv preprint. arXiv:1910.07867
8. Derakhshan B, Mahdiraji AR, Rabl T, Markl V (2019) In: EDBT, pp 397–408
9. Ash JT, Adams RP (2019) arXiv preprint. arXiv:1910.08475
10. Lin Y, Han S, Mao H, Wang Y, Dally WJ (2017) arXiv preprint. arXiv:1712.01887
11. Stich SU, Cordonnier JB, Jaggi M (2018) In: Advances in neural information processing systems, pp 4447–4458
12. Aji AF, Heafield K (2017) arXiv preprint. arXiv:1704.05021
13. Bottou L (2010) In: Proceedings of COMPSTAT'2010. Springer, Berlin, Heidelberg, pp 177–186
14. Bottou L (2012) In: Neural networks: tricks of the trade. Springer, Berlin, Heidelberg, pp 421–436
15. Jaini P, Rashwan A, Zhao H, Liu Y, Banijamali E, Chen Z, Poupart P (2016) In: Conference on Probabilistic Graphical Models, pp 228–239
16. Wu P, Hoi SC, Xia H, Zhao P, Wang D, Miao C (2013) In: Proceedings of the 21st ACM international conference on Multimedia, pp 153–162
17. Mohammadi M, Al-Fuqaha A, Sorour S, Guizani M (2018) IEEE Commun Surv Tutor 20(4):2923
18. Baylor D, Haas K, Katsiapis K, Leong S, Liu R, Menwald C, Miao H, Polyzotis N, Trott M, Zinkevich M (2019) In: 2019 USENIX Conference on Operational Machine Learning OpML 19, pp 51–53
19. Baylor D, Breck E, Cheng HT, Fiedel N, Foo CY, Haque Z, Haykal S, Ispir M, Jain V, Koc L et al (2017) In: Proceedings of the 23rd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, pp 1387–1395
20. Katsiapis K, Haas K (2019) In: Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining, pp 3182–3182
21. Lomonaco V (2019) Continual learning with deep architectures (Ph.D. thesis, alma)
22. Lomonaco V, Maltoni D, Pellegrini L (2019) arXiv preprint. arXiv:1907.03799, vol 1
23. Parisi GI, Lomonaco V (2020) In: Recent trends in learning from data. Springer, Cham, pp 197–221
24. Pomponi J, Scardapane S, Lomonaco V, Uncini A (2020) Neurocomputing
25. McCloskey M, Cohen NJ (1989) In: Psychology of learning and motivation, vol 24. Elsevier, Academic Press, pp 109–165. https://doi.org/10.1016/S0079-7421(08)60536-8
26. Recht B, Re C, Wright S, Niu F (2011) In: Advances in neural information processing systems, pp 693–701
27. Alistarh D, Grubic D, Li J, Tomioka R, Vojnovic M (2017) In: Advances in neural information processing systems, pp 1709–1720
28. Koloskova A, Stich SU, Jaggi M (2019) arXiv preprint. arXiv:1902.00340
29. Wikipedia contributors (2021) Independent and identically distributed random variables. https://en.wikipedia.org/w/index.php?title=Independent_and_identically_distributed_random_variables&oldid=1017206855. Accessed 9 Aug 2021
30. Dawid AP (1984) J Royal Stat Soc Ser A 147(2):278
31. Paszke A, Gross S, Massa F, Lerer A, Bradbury J, Chanan G, Killeen T, Lin Z, Gimelshein N, Antiga L et al (2019) In: Advances in neural information processing systems, pp 8026–8037
32. Marcel S, Rodriguez Y (2010) In: Proceedings of the 18th ACM international conference on Multimedia, pp 1485–1488
33. LeCun Y (1998) http://yann.lecun.com/exdb/mnist/. Accessed: 1 Jun 2021
34. Krizhevsky A (2009) MSc Thesis, University of Toronto
35. LeCun Y et al (2015) 20(5), 14. http://yann.lecun.com/exdb/lenet. Accessed 1: Jun 2021
36. Sandler M, Howard A, Zhu M, Zhmoginov A, Chen LC (2018) In: Proceedings of the IEEE conference on computer vision and pattern recognition, pp 4510–4520
37. He K, Zhang X, Ren S, Sun J (2016) In: Proceedings of the IEEE conference on computer vision and pattern recognition, pp 770–778
38. Deng J, Dong W, Socher R, Li LJ, Li K, Fei-Fei L (2009) In: 2009 IEEE conference on computer vision and pattern recognition Ieee, pp 248–255
39. Kingma DP, Ba J (2014) arXiv preprint. arXiv:1412.6980
40. kuangliu (2020) Train cifar10 with pytorch. https://github.com/kuangliu/pytorch-cifar. Accessed 1 May 2020