Erasmus Mundus Joint MSc in
Big Data Management and Analytics

Technische Universität Berlin

Master Thesis

Continuous Training and Deployment of Deep Learning Models

Ioannis Prapas

Matriculation Number: 415376

15.08.2020

Supervised by Prof. Dr. Volker Markl

Advised by Behrouz Derakhshan and Dr. Alireza Rezaei Mahdiraji

This dissertation originated in cooperation with the Deutsches Forschungszentrum für Künstliche Intelligenz (DFKI).

There is no doubt that conducting a Master thesis requires time, devotion and mental stamina. However, working on it becomes much easier when you are surrounded by people that support you and care enough to provide their guidance.

In that respect, I am lucky to have had the support of two excellent advisors throughout this project, Behrouz Derakhshan and Dr. Alireza Rezaei Mahdiraji. Both of them are excellent researchers, very friendly and fun to work with. We regularly met once every two weeks, when they heard about my progress and provided advice for the next steps. They were also available for ad-hoc discussions whenever I needed them. Their experience allowed them to be open to new ideas with potential and at the same time recognize ideas with limited or no potential. Behrouz always shared his clear vision of the next steps and did not even stop devoting time to this work during his paternity leave. Alireza made sure that I would not diverge too much from the scientific methods and the end goal. I am grateful to have them as my supervisors and I am sure to miss our regular meetings.

An acknowledgement section should always mention my parents Nikos and Maria for their continuous support since the beginning of my life. I also want to thank my fellow classmate Sokratis Papadopoulos for always being one step ahead in his Master thesis' progress and therefore inspiring me to move forward as well. And of course, I want to thank Danai who has been on my side throughout this thesis and to whom I promise to travel together more when the work on this thesis is over.

Last but not least, I am thankful for the full scholarship from the Erasmus Mundus Joint Master Program in Big Data Management and Analytics (BDMA) which allowed me to stay focused on my work for this thesis.

Hereby I declare that I wrote this thesis myself with the help of no more than the mentioned literature and auxiliary means.

Berlin, 15.08.2020

. . . .

(Signature, Ioannis Prapas)

Abstract

In recent years, Deep Learning has consistently surpassed other Machine Learning methods and achieved state-of-the-art performance in multiple fields, such as speech recognition, language understanding, and computer vision. Deep Learning model training is usually done in epochs, i.e. multiple passes over the dataset. With modern Deep Learning model sizes that can have billions of parameters and dataset sizes of the Big Data era, training is a very compute-intensive process. After enough new training data become available, the model has to be refreshed, commonly achieved by restarting the training from scratch, a process called full retraining. Full retraining is wasteful as it discards the previously learned models, despite the huge amounts of compute needed to generate them. Moreover, it results into stale models which need to be updated again when enough new data are available. Instead, we propose to continuously train Deep Learning models via proactive training. Proactive training uses a combination of new data together with a sample of historical data to form mini-batches for Stochastic Gradient Descent based optimization. Continuous training implies continuous deployment of the model to a server where it is going to serve prediction queries. Naively deploying Deep Learning models, i.e. transferring millions - if not billions - of parameters at every training iteration heavily bottlenecks the process and consumes high amounts of network bandwidth. We borrow ideas from distributed model training to sparsify model changes in order to facilitate continuous deployment. Our experimental results with LeNet5 on MNIST and modern Deep Learning models on CIFAR10 show that proactive training keeps models fresh with comparable - if not superior - performance to full retraining at a fraction of the time. Meanwhile, combined with sparsity, sparse proactive training enables very fast updates of a deployed model with arbitrarily large sparsity reducing communication per iteration up to 10,000x times, with minimal - if any - losses in model quality. Sparse training, however, comes at a price; it incurs an overhead on the training that depends on the size of the model and increases the training time ranging 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 Deep Learning models.

Contents

List of Figures

List of Tables

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 [35], computer vision [39] and natural language understanding [14]. This success is explained by two important advantages of DL in comparison to traditional ML:

- **DL performs automatic feature engineering**. When compared to conventional ML approaches that rely on hand-designed features, DNN models can extract useful features directly from raw data via representation learning [56], which is inherent to their training. In an abstract way, the parameters of a DNN model can be thought as learnable feature extractors. Deeper layers of a DNN combine the features of earlier layers to learn more higher level features.

- **DL model quality scales better with large datasets**. As a consequence of the previous point, large datasets allow DNN models to learn increasingly better representations and continue to improve their performance in a data-informed manner.

The success of DL has been therefore enabled with the coming of the Big Data era that made large datasets available and advances in GPU technology that allowed for their computation. Meanwhile, dedicated frameworks like Caffe [41], PyTorch [68] and Tensorflow [1] make the implementation of DL models easier and have thus further contributed to their success. These frameworks offer high-level abstractions for defining, training and deploying DNNs and remove the Data Scientists' burden of calculating derivatives, by offering automatic differentiation.

Paired with its advantages, DL also possesses some negative properties:

- Firstly, despite works to visualize and understand what DL is actually learning [71], DNNs remain widely *black-box models* and their results are not directly interpretable.

- Secondly, DL relies on minimizing *non-convex loss functions* which are not theoretically well understood. This means that results on convex ML models like the Support Vector Machine (SVM) or Linear Regression do not necessarily translate to DL. Yann Le Cun [54] argues that non-convexity is not a problem as DL simply works better than shallow convex learners; the problem is our limited theory behind it.

- Thirdly, DL models are *resource-hungry* needing to train for multiple epochs (passes over the dataset) on expensive hardware in order to be useful. According to Krizhevsky et al. [51], Alexnet "takes between five and six days to train on two GTX 580 3GB GPU". Neural machine translation [83] took 9 days to train on 96 K80 GPUs. To achieve superhuman level performance, AlphaZero [75] trained for about 8 hours using more than 5000 TPUs while its total training took 100 times that, i.e. about a month. In many applications, such as financial models and recommender systems, new training data is constantly collected and models are updated by retraining from scratch on the whole dataset.

- Fourthly, DL models can be massive, which can make the model deployment a very slow process. It is not uncommon for modern DL architectures to have a few hundred million or even several billions of parameters. A recent extreme is the GPT-3 [14] language model that features about 175 billion of parameters.

Our work deals with these two last problems that arise after the initial training of a DL model, when new data are available, and the model's staleness increases. The current prominent approach is to trigger an expensive and slow full retraining on the updated full dataset. Instead, we propose to use the learned parameters and a strategy for continuous training on new and historical data. We match this continuous training approach with sparsified model changes in order to reduce the communication and continuously deploy model changes as soon as they happen at every training iteration.

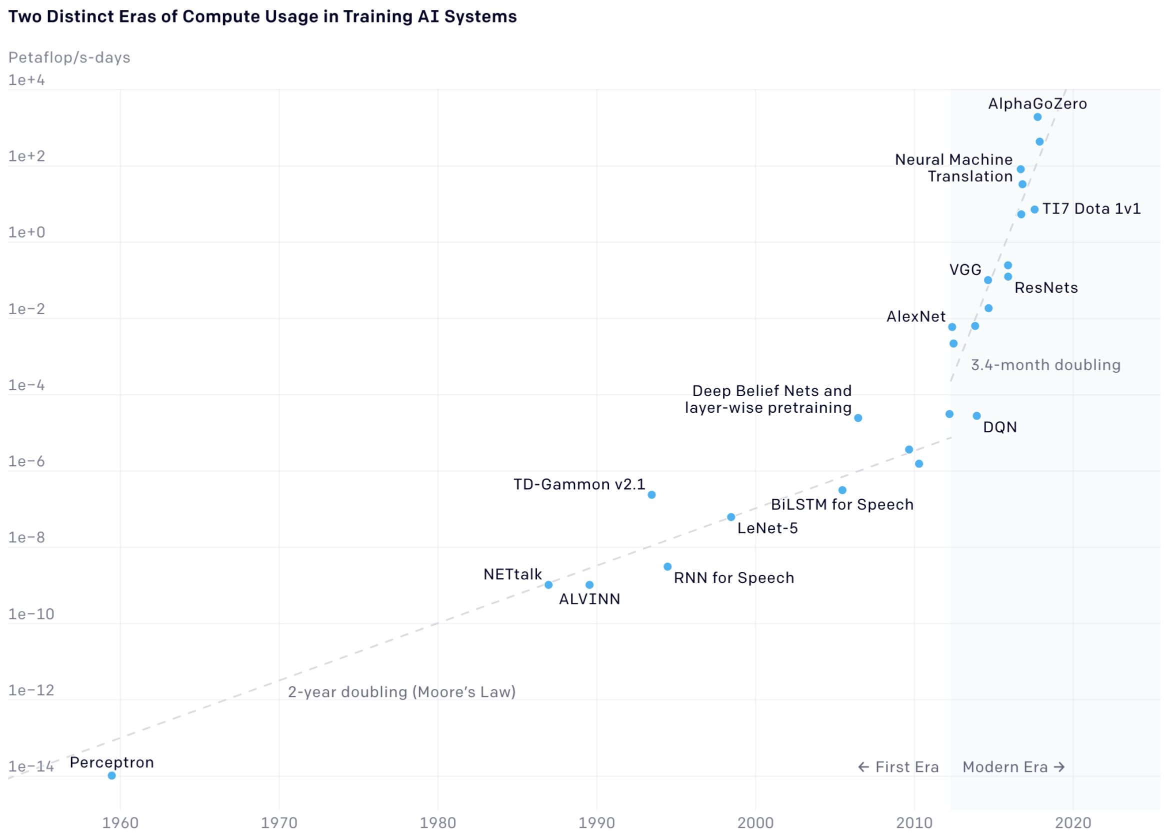


Figure 1.1: The total amount of compute, in petaflop/s-days, used to train relatively well known model architectures [5]

1.1 Motivation

In the era of the Internet of Things (IoT), we are abounded by sensory devices (e.g. smartphones, cameras, sensors) that collect and generate data streams continuously. Meanwhile, systems that rely on DL use ways to continuously generate labeled data through the way their users interact with the systems. For example, Netflix users in 2012 were generating 4 million ratings per day and adding 2 million movies and TV shows to the queue [4].

Modern applications require DL models to learn from these streams as soon as possible, in order to provide predictions that are as relevant as possible. Current systems support DL model updating, by performing a periodical full retraining when enough new data are available or model performance degrades [23]. We find this process to be wasteful and resulting to stale models:

- Wasteful, because it discards the previously learned model that has consumed considerable amounts of compute. 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 [6].

- 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. Consequently, the periodical retraining method suffers in terms of scalability, because the needed time to retrain a model keeps growing as the dataset keeps growing.

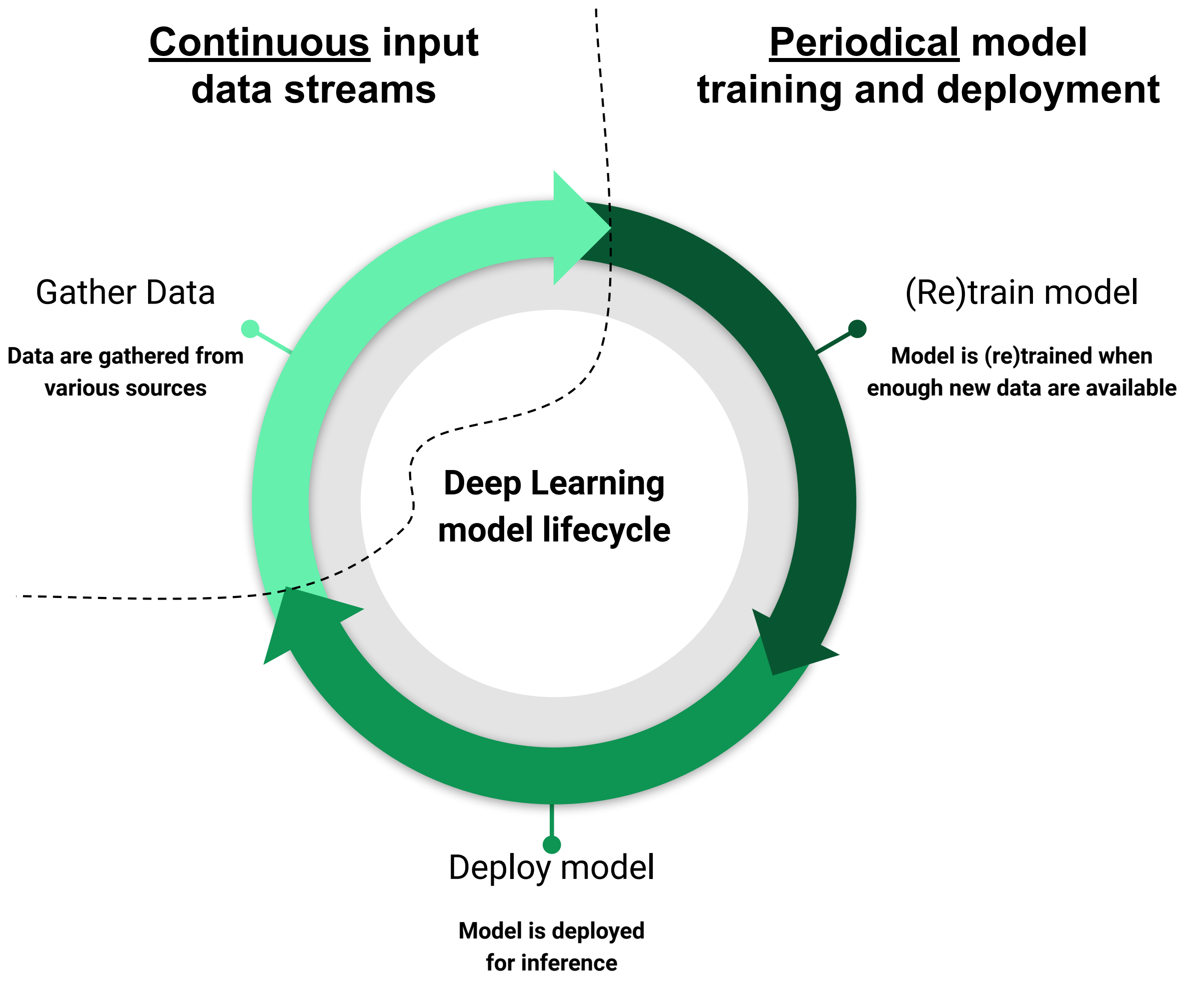

Continuous input data streams
Periodical model training and deployment
Gather Data
Data are gathered from various sources
(Re)train model
Model is (re)trained when enough new data are available
Deep Learning model lifecycle
Deploy model
Model is deployed for inference

Figure 1.2: The DL model lifecycle and the mismatch between continuous input data streams and periodical training and deployment.

DL is a class of algorithms which has been shown to achieve state-of-the-art performance in multiple ML problems. In order to achieve that performance, they need excessively high amounts of data and compute that is exponentially rising (Figure 1.1). With the emergence of data streams as the input to DL models, we identify a mismatch in the DL model lifecycle between continuously arriving data and periodical training and deployment (Figure 1.2). training DL models only on batch datasets is not sufficient for applications that require learning as soon as data become available. In the meantime, for most applications, online learning either needs rigorous tuning or does not provide the same model quality as batch learning approaches [23], since there is merit to continue training on historical data and not only performing one pass. In this thesis, we propose a balanced solution between online learning and full retraining to continuously train DL models.

A trained DL model is required to perform inference, i.e. respond to prediction queries. Since the requirements for inference are different from the requirements for training, a DL model is typically deployed in a different machine than the one used for training. Therefore, continuous training implies continuous deployment to a remote machine. However, modern DL models can have from multiple millions to mutliple billions of parameters, a fact which makes frequent deployment quite challenging and is a major obstacle to continuous training. To successfully overcome this obstacle, this thesis borrows ideas from the distributed DL training domain and proposes a sparse continuous training and deployment.

1.2 Objective

This work aims to provide a direction to solve the mismatch shown in Figure 1.2 by making all the processes of the lifecycle continuous. We propose an approach to continuously train and deploy DL models while new data arrive.

For the continuous training, we leverage proactive training as proposed by Derakhshan et al. [23]. Proactive training uses a combination of new data with samples of historical data to form mini-batches for mini-batch SGD-based training. Although it has been used to successfully 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 implement proactive training for DL models and experiment with LeNet-5 on MNIST and state-of-the-art DL models on CIFAR [50]. In order to facilitate the frequent deployment that is implied at every proactive training iteration, we identify that we have to reduce the communication cost of model changes. We borrow concepts from distributed DL training to perform compressed model changes that are sufficiently small to be transferred across the network at every training iteration. We experiment with different sparsification strategies reducing the communication cost up to 99.99% per iteration. Our work enables continuous training and deployment of DL models in environments where training data arrive continuously.

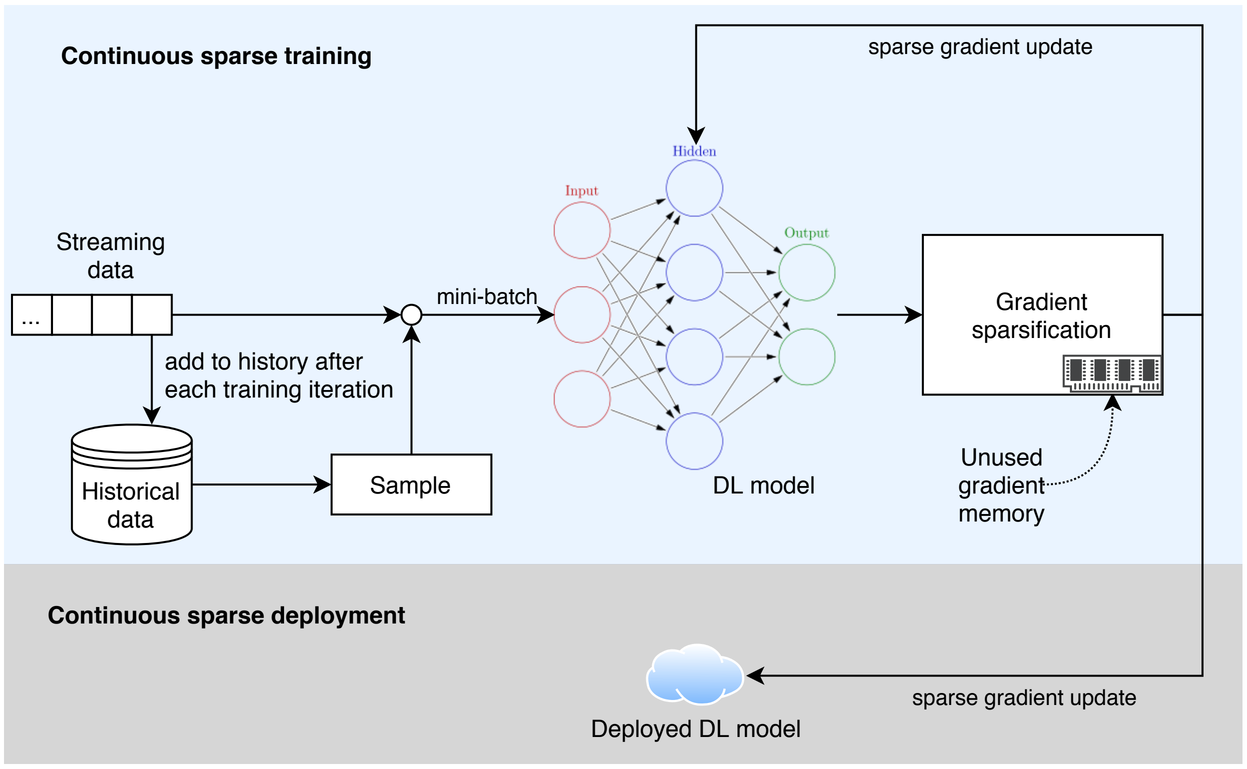


Figure 1.3: Proposed solution for continuously training and deploying DL models

Our solution is presented in Figure 1.3. New streaming data are combined with historical data to form mini-batches that are fed to a DL model, a process we describe in Section 4.1 as proactive training. The model is updated with sparse SGD with memory (Section 4.2), which allows to only use a small percentage of model changes per iteration with no or minimal compromises in model quality. The sparse changes reduce the communication cost for sending the changes to the deployed model and the unused gradients are accumulated in memory to contribute to later updates. After being used for training, new data become part of the historical database and are therefore subject to being sampled in the future.

1.3 Limitations of this study

This work does not study the learning rate scheduling or any starting and stopping criteria for the continuous training that can allow to achieve state-of-the-art performance. Each of these topics deserves in-depth future work. In our experiments, we quantify the network cost of continuous DL model deployment but do not actually send any model changes. Unstable network connection can cause many undesirable effects such as missing, delayed or corrupt model changes, which a future study should address. Moreover, our approach is proposed and implemented for centralized DL model training, but is important to be extended in later works to parallel and distributed setups. Finally, although relevant for the streaming setting, this work does not study adaptive neural network architectures, but focuses on static DNNs and therefore assumes inputs that have a static schema.

1.4 Contributions

This works enables the continuous training and deployment of DL models. We validate our approach by experimenting (Chapter 5) with LeNet-5 on MNIST and state-of-the-art DL models on CIFAR [50]. The contributions of this work can be summarized by the following points:

- **Enabled the continuous training of DL models**. We repurpose the proactive training approach for the continuous training of DL models (Chapter 4) and validate experimentally (Chapter 5) that it can achieve comparable model quality to full retraining resulting into models that are constantly fresh.
- **Complexity analysis of (re)training approaches**. We conduct a complexity analysis (Section 4.1.1) that compares full retraining, online training and proactive training as methods for keeping models updated after the initial training. We find that under certain assumptions, the cost of the full retraining approach grows quadratically as new data become available and depends on the size of the historical dataset. The same cost grows linearly for online and proactive training and is independent of the size of the historical dataset.
- **Enabled the continuous deployment of DL models**. We bring the idea of sparse training for the continuous deployment of DL models. Although it carries an overhead in

training time, it can provide arbitrarily large reductions in the communication costs and can therefore enable frequent DL model deployment. To the best of our knowledge, this is the first study that proposes sparse training for continuous deployment. It has applications beyond the combined use with proactive training as it can enable continuous deployment in the online training setting as well.

- **Empirical analysis of the hyper-parameters of sparse training**. For sparse training, we perform a rigorous empirical study (Chapter 5) measuring the overhead and model quality trade-offs when using different sparse selectors (top_k and $random_k$) and various sparsification ratios. We see the effect of these settings in a variety modern DL models.

- **Regularizing effect of top_k sparse training**. The top_k sparse selector allows to consider only the gradients with the higher absolute magnitudes at every training iteration. Our experiments reveal that top_k sparse training can at times achieve better model quality that its non sparse variant, suggesting that it brings a regularizing effect. As such, we confirm that the magnitude of the gradients is a good proxy of the importance of model changes.

1.5 Outline

The following presents the structure of the document and an overview of the chapters that follow:

- In chapter 2 we dive into related literature. We first discuss about work that has focused on continuous training of DL architectures, and then present related work for deployment compression and gradient compression.

- In chapter 3, we give a brief overview of DL and dive deeper into DL training and deployment.

- In chapter 4, we present our methods for continuous training and deployment. We also conduct a complexity analysis that compares full retraining, online training and proactive training as (re)training approaches.

- In chapter 5 we evaluate our methods with LeNet-5 DL model on the MNIST dataset and various modern DL models on the CIFAR dataset.

- Lastly, in chapter 6 we conclude the thesis with a summary of its contributions, findings and directions for future work.

2 Related Work

Current DL models and systems are not suited to deal with newly arrived data as the most prominent approach is to discard the previously trained model and start training from scratch. The goal of this Master thesis is to seamlessly continue the training and deployment of DL models when new training data become available. Our work has two major directions:

- Efficient continuous training of DL models reusing pretrained model and historical data.
- Efficient continuous deployment of continuously trained DL models.

The related work is presented separately for these directions in the following sections of this chapter. In Section 2.1 we see work on training DL models when new data become available. In section 2.2, we present ideas around compressed model deployment.

2.1 Retraining of DL models

When it comes to DL model training, the standard approach 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 has good properties for training in large scale datasets [12, 13] and practically has proved its worth in training DL models as the de-facto optimization method. It is generally combined with momentum or made adaptive (Adagrad [25], Adam [47]) in order to converge faster. However, some studies criticize the widely used adaptive variants, showing that there are cases where better generalization can be achieved with vanilla SGD [81, 46].

After initial training, when enough new data become available or the model's performance degrades the training process is restarted from scratch in an effort to prevent model staleness. However, the previously trained model still has some value [9] and this is why platforms like Tensorflow

Extended (TFX) [8, 44] allow for the training to be started with the parameters of the previously trained model, a process called *warm-starting*. Properties of the warm-starting have not yet been studied in detail, but it is generally accepted that it will lead to good model quality faster than from-scratch training. Nevertheless, warm-starting still falls under the area of batch learning, where the whole data set needs to be available during training. Moreover, there is no evidence that state-of-the-art generalization can be achieved with warm-starting, while there is evidence of the contrary [6]. In addition, if you combine the fast arrival of new data with an increasingly slow retraining process, you end up with a great recipe for stale models in production.

As opposed to batch learning, online learning keeps models fresh incrementally as new data arrives. Online learning for DL has received limited attention [40, 82], but should be highlighted more as IoT applications gain more ground and produce streams of data [64]. In contrast to batch learning methods that bring an expensive retraining cost whenever new training data arrive, online learning performs updates based only on the new training data. This makes online learning highly scalable and suitable large-scale applications. However, in many cases in order to converge to a good model, there is merit to continue training on historical data and not only take into account new data. Our work proposes a method for continuous training of DL models that provides a balance between online Learning and full retraining.

Recent work by Derakhshan et al. [23] shows that when it comes to training ML models, full retraining is not always required and Online Learning is not good enough. Full retraining is wasteful, as old model is discarded and Online Learning is not performant enough, because continuous training on past data has some merit. They propose to periodically trigger *proactive training* which 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 [59, 60, 67, 69]. In that framework, a model is learning tasks one after another with the goal to learn the last task without *catastrophic forgetting* [62] that is losing the ability to perform on previous tasks. In a more limited scope, we would like models to learn from new data, without forgetting how to deal with historical data. A characteristic example of work in this domain is employing regularization to not let the parameters of a network diverge too much from past embeddings of the trained model [59, 60]. More similar to our setting, we find rehearsal-based continual learning [33, 62] in which characteristic examples are kept in buffers for continuing training together with new data as they arrive.

2.2 Deployment of DL models

Training a DL pipeline is one step of the real-world DL applications. After training, one must deploy the pipeline and model and make them available for inference. Continuously triggering proactive training implies continuous deployment of DL models. However, modern DL models can have a huge number of parameters and size ranging from several hundred MBs to several GBs. Naively deploying such big models continuously is not feasible because of the intractable communication cost. We identify two relevant research directions to make this communication cost tractable: i) *model compression* ii) model change compression, also known as *gradient compression.*

Model compression has generally been examined as a solution to deploy models on edge devices with limited capabilities. The most prominent solutions in this domain propose quantization, pruning or sharing of a model's parameters. *Network quantization* compresses a model by reducing the number of bits required to store its parameters, with the extreme case of binarization (1-bit quantization). Han et al. [31] propose deep compression which clusters model weights, performs quantization-aware training and huffman coding to produce models with a compression rate of up to 49x. Mobilenets [38] combine these ideas together with an efficient convolutional neural network architecture to achieve competitive performance on Imagenet at a fraction of parameters and with much less computation needed for inference. One interesting idea is *knowledge distillation* [15, 36], that transfers the knowledge of a large model to a smaller one, which is to taught to replicate the predictions of the larger model. Many modern DL platforms [68, 1] offer quantization and pruning modules which produce models that offer a balance between model size and model quality aiming to optimize the inference and deployment time on devices with limited computing capabilities.

Despite their usefulness for deploying on edge devices, these approaches require sacrifices either in accuracy or training time to achieve compression. Therefore, we find them not to be suited for the general deployment case that needs to stay as fast and as accurate as possible.

Work in gradient compression offers a more promising direction for our deployment scenario. In this area, we find approaches that are meant to minimize 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 [70] lets workers send gradients in an asynchronous fashion without any sacrifice in convergence rates. Alistarh et al. [3] propose a stochastic quantizated extension of SGD, called Quantized SGD (QSGD), which provides a trade-off between convergence rate and compression. Seide et al. [73] empirically show that even 1-bit quantization converges at the same rate as normal training, provided that quantization errors are propagated to the next iterations. Aji et al. [2] sparsify gradient updates by only considering the top-k components in every iteration and accumulating smaller gradients. Deep gradient compression [58] shows that extreme sparsification reduces communication costs by 99.9% and works well with modern deep learning architectures. Koloskova et al. [49] use it to perform decentralized training under arbitrarily large sparsification. Stich et al. [76] prove its good convergence guarantees provided that a memory accumulates not updated gradients. To the best of our knowledge, this work is the first to consider such gradient compression schemes for continuous deployment of DL models without any loss in model quality.

3 Background

In this chapter, we present background concepts around DNNs. In Section 3.1, we start with a brief high-level overview of DL, its workings and its modern history. After, in Section 3.3 we show how they are trained, before continuing with a general idea of how they are deployed (Section 3.4).

3.1 What is DL?

3.1.1 The Artificial Neuron

To understand DL, we have to first zoom into its **basic component**: The *artificial neuron.* It is loosely inspired by a biological neuron (see Figure 3.1), which receives stimuli from other neurons and ”fires” an electrical pulse, provided that the aggregated stimulus reaches a certain threshold. Similarly, an artificial neuron receives input x, combines it with its learnable parameters (weights W), makes a sum of the result (adding a bias term) and activates or not as dictated by its non-linear activation function f. As such, the output of the neuron is shown in Formula 3.1 in a vectorized form.

$$y = f(W^T x + bias) \tag{3.1}$$

DNN models use non-linear activation functions, like the hyperbolic tangent (tanh), the logistic sigmoid and the Rectified Linear Unit (ReLU). This study does not discuss the details of each one and their modern variants. The reader is referred to a DL book like the one by Goodfellow et al. [27] for an in-depth analysis.

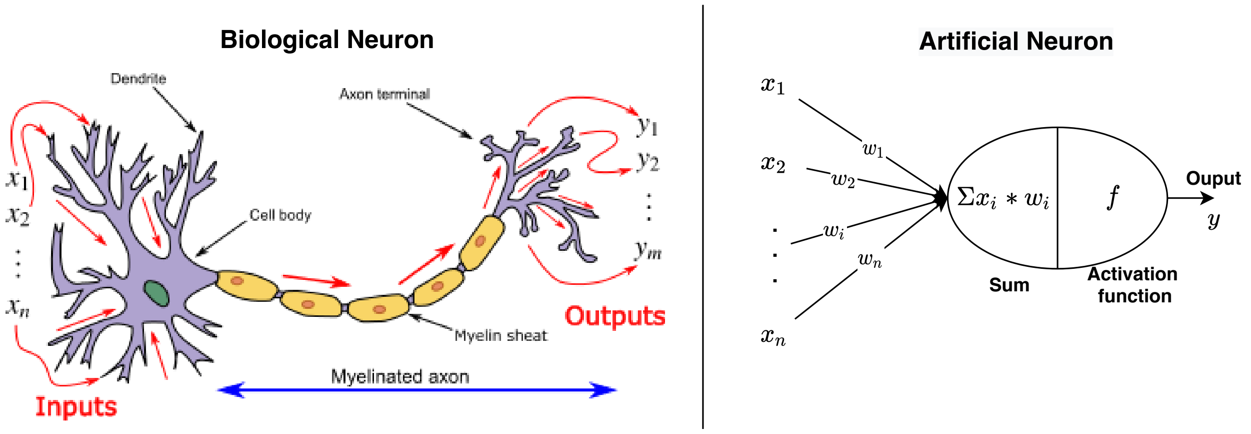


Figure 3.1: Similarities between a Biological Neuron (left) and an Artificial Neuron (right). Biological Neuron image taken from Wikipedia [79].

3.1.2 Artificial Neural Networks

An Artificial Neural Network (ANN) is comprised of layers of Artificial Neurons. An initial *input layer* feeds data to its subsequent ANN layer. A layer's output can serve as input to the next ANN layer, with the last layer being the *output layer*. Any layers between the input and output are called *hidden layers*. **DNNs are simply ANNs with more than one hidden layers**. The more hidden layers an ANN has, the deeper it is. Deeper layers learn more complex, abstract features combining the features learned from earlier layers. Experimental results have consistently shown that greater depth does achieve better generalization for a great variety of tasks [27].

Currently, there is a big variety of DNN models in the wild. Convolutional Neural Networks (CNNs) [55] are comprised of convolutional layers that work well on image data. Recurrent Neural Networks RNNs consist of neurons whose output is fed back to their input and they have been shown to work well with sequence data such as text or time-series. Major variants of RNNs include the Long Short Term Memory (LSTM) [37] and Gated Recurrent Unit (GRU) networks [17]. Generative Adversarial Netwroks (GANs) [28], originally proposed for unsupervised learning, are comprised of a generator and discriminator network that compete with each other. The discriminator learns to tell is some data are real or not, and the generator learns to fool the discriminator by generating data that looks real, mimicking the data generation process. Another type of ANN for unsupervised learning is the Autoencoder, which learns a non-linear compressed representation of the input data.

3.2 Brief History of DL

The main ideas behind Neural Networks have been around since the 1960s. The field did not receive much attention until the success of DL on ImageNet [22], with Alexnet [51] surpassing the state-of-the-art by a large margin. At the time, in 2012, it was the only Neural Network solution. After that, the Imagenet leaderboard has been dominated by DNN solutions which have continued to improve their error rate and have even surpassed human performance [43] (Figure 3.2).

Since then, DL has consistently achieved great results in multiple domains, such as computer vision, speech recognition [32], language understanding [24], and machine translation [83]. DL combined with reinforcement learning has allowed agents to learn by interacting with their environment and achieving extraordinary tasks. More specifically Deep Reinforcement Learning has achieved superhuman performance at playing the game of Go [75] and video games (Atari [63], Starcraft [78], Dota 2 [11]). It has also brought advances in robotics and autonomous driving [30].

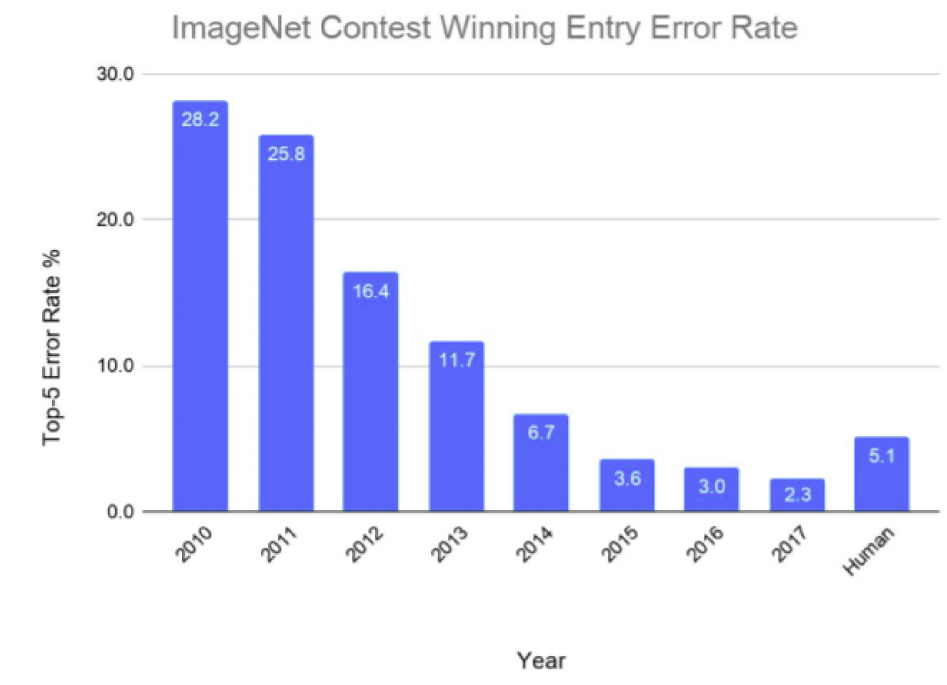


Figure 3.2: ImageNet classification contest winner Top-5 error rate over time [21]

In the last decade, DL has been applied in multiple sciences and has been considered by Klinger et al. [48] as a General Purpose Technology, which places it as an invention next to the steam engine, the electric motor and the microprocessor. Among the examples of the DL diffusion into other sciences, Jeff Dean [21] points out "earthquake prediction, flood forecasting, protein folding, high energy physics, medical diagnosis and genomics".

3.3 DL model training

Training a DL model falls under the domain of *empirical risk* minimization. Minimizing the prediction error on observed (or training) data is used as a proxy to minimize the generalization error, that is the expected error on test data, also called the *expected risk*. In the typical supervised setting training data x are fed into a DNN model with parameters θ to produce the output $f(x, \theta)$. The output of the DNN is then compared with the desired output(labels of training data) by using the value of a loss function $L(f(x, \theta), y)$. The goal of training a DL model is to modify its parameters θ in a way that minimizes the expected value of the loss function L. The expected value of the loss function can also be found in the literature as the cost function, the objective function or the error function and is usually denoted as $J(\theta)$. As such, the problem of learning for DNNs boils down to minimizing Formula 3.2.

$$J(\theta) = \mathbb{E}_{x,y \in observed\_data} L(f(x, \theta), y) \tag{3.2}$$

Algorithm 1: Mini-Batch Stochastic Gradient Descent.

```
  Input: Model Parameters $\theta$, Dataset D, Learning rate $\alpha$, Mini Batch Size $b$
  Result: Updated Model Parameters $\theta$
1 while stopping criteria not met do
2     sample b elements from dataset D $\{x^{(1)}, ..., x^{(b)}\}$, with labels $\{y^{(1)}, ..., y^{(b)}\}$
3     $g = \frac{1}{b} \sum_{i=1}^{b} \frac{\partial}{\partial \theta} L(f(x_i, \theta), y_i)$        // calculate gradients
4     $\theta = \theta - \alpha * g$        // update model's parameters
5 end
```

As stated by Goodfellow et al. [27], "the non-linearity of a neural network causes most interesting loss functions to become non-convex". This does not allow a DL model's parameters to be optimized with global convergence guarantees. Instead, the update of a DNN's parameters is typically done via optimization algorithms based on mini-batch Stochastic Gradient Descent (SGD). The mini-batch SGD (Algorithm 1), takes as input a model's parameters θ, a dataset D, learning rate α and a mini-batch size b, and outputs the updated optimized parameters θ. It runs until some stopping criteria are met (Line 1). These stopping criteria can be something similar to a predefined number of iterations or convergence of the loss function to a certain threshold. At each iteration, a mini-batch is formed by b labelled elements from dataset D (Line 2). The approximate gradient of the loss function is calculated based on that mini-batch sample (Line 3). The parameter's θ are updated by following the opposite direction of the gradients with a step size equal to the learning rate α (Line 4). This makes the learning rate a very important parameter that needs careful tuning, depending on the model and dataset that it is applied to. It is common practice for the learning rate be decayed after a number of iterations.

Following the opposite direction of the gradients, the parameters can be updated in order to minimize the cost function towards critical points of zero curvature. The ideal regions of zero curvature are the global minima, but SGD can also get stuck on points other of zero curvature like local minima or saddle points (Figure 3.3) and is sensitive to the initial values of the parameters. However, using good parameter initialization [26] and noisy approximate gradients has empirically been shown to effectively to minimize the generalization error. Moreover, variants of SGD like Momentum, Adagrad [25] and ADAM [47] have been developed to improve its convergence time.

The gradients of the cost function with respect to each one of the parameters of a DNN is efficiently calculated with the backpropagation algorithm [45]. As such the DL model training is done in two passes:

- In the **forward pass** the input flows through the network to generate the model's output.
- In the **backward pass** the model's output is compared with the desired output using a loss function. Each individual neuron's contribution to the loss is calculated with backpropagation algorithm and the parameters of the network are updated with the SGD algorithm.

An interesting property of SGD is that it follows the gradient of the true generalization error provided that no data are repeated. With a large enough dataset, convergence within some predefined tolerance can occur performing an incomplete pass over the data. In practice, though, good performance can be achieved by performing multiple passes over the data (epochs). Rather than using the SGD algorithm that continuously samples a dataset as shown in Algorithm 1, most DL applications do the training in epochs. Before each epoch the dataset is shuffled and split into batches. This allows for simpler implementations and contiguous memory access.

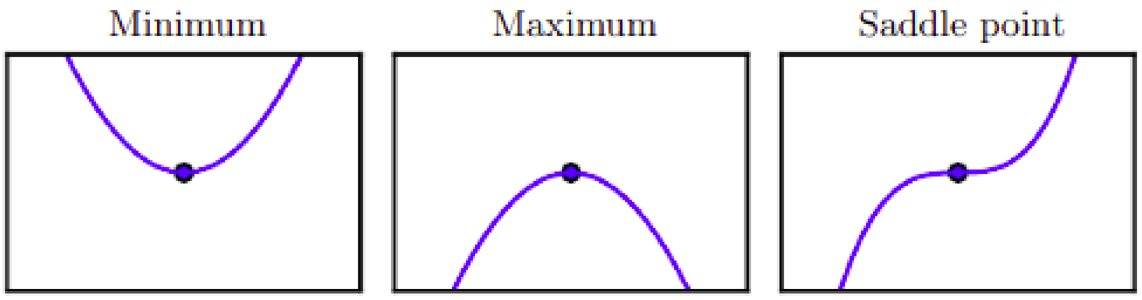


Figure 3.3: Types of critical points for SGD-based optimization

Arguably, one of the most important properties of SGD-based optimization is that the computation time per update is independent of the number of training examples. This allows it to be one of the best algorithms for learning from large-scale datasets and the de-facto method for trainining DNNs, despite the fact that it is not the best optimization algorithm [13].

3.3.1 Hardware for DL training

DL model training involves a relatively small number of types of operations like matrix multiplications, vector operations, application of convolutional kernels and other linear algebra calculations [21]. Thus, many mechanisms that are employed by Central Processing Units (CPUs) to efficiently run general-purpose programs (e.g. branch predictors, hyperthreaded-execution, cache memory hierarchies) are unnecessary in DL. Instead, the operations involved in DL model training are similar to the ones needed for video rendering, a domain where Graphics Processing Units (GPUs) excel. The massive parallelism offered by GPUs combined with their high memory bandwidth allows them to achieve significant speedups in DL model training over their CPU counterparts [74].

DL can also profit from more dedicated hardware such the Tensor Processing Units (TPUs) [42], implemented by Google. They take advantage of the fact that DL models are very tolerant to reduced-precision arithmetics and introduce an optimized floating point format (bfloat16). This allows them to optimize memory usage and improve the training efficiency [21].

3.3.2 Software for DL training

Most prominent software frameworks for DL training are open-source and backed by industry giants and academia. Modern software efficiently leverages of the power of one or multiple GPUs to enable fast training execution. The landscape of DL software is changing quite fast with new players entering the game. Theano [10] and Caffe [41] are two of the early DL frameworks that supported GPU execution and come from the academia. In recent times, some of the most prominent frameworks are Pytorch [68] which is primarily backed by Facebook, Tensorflow [1] which is mainly backed by Google and MXNet [16] which is currently an Apache project backed by various industry giants like Amazon and Microsoft.

We find that modern DL software deals with training for multiple epochs on batch datasets rather than continuously on data streams. However, not many changes are needed to adapt to continuous training algorithms since SGD - the de facto training algorithm of DL - is inherently an incremental optimization algorithm.

3.4 DL model deployment

After a DL model is trained, it needs to be deployed in an environment where it will serve prediction requests on production, unseen data. Serving prediction requests requests is also known as inference. The requirements for inference are different than the ones for training and therefore it is common practice to deploy on different machines than the ones used for training.

During training there is a major need for efficiently executing both the forward pass which generates a model's output but also the generally slower backward pass which updates the model's parameters based on the calculation of the gradients. Moreover, training is done in batches and there is need for throughput and no concern for latency. Inference cares only about the forward pass which is less costly than a training iteration that involves the costly backward pass. Latency is very relevant at inference time; typically user requests cannot wait more than 1 second for receiving results for their prediction queries. The model serving for inference has to be scaled according to the demand, while training has to be scaled in order to finish in a reasonable amount time. While typically DL model training is done in GPUs, inference can be fast enough in CPUs or even edge devices with limited computing capabilities.

There are quite a few ways one might want to deploy a DL model:

- Web deployment is the typical use case where a model is deployed on a web server.
- Mobile deployment is about deploying ML models in mobile phones.
- Embedded deployment is about deploying ML models on edge devices.

In this study, we assume the more general web deployment. In that setting, there are quite a few options regarding the deployment architecture which is tightly coupled with the software that will be used for deployment:

- Embed the model code in a web server and expose a prediction interface. This is a custom solution that views model serving as another web application.
- Load the model in a containerized application and handle serving via orchestration with a software like Kubernetes. This is a more modern solution of serving web applications that can easily scale to user demand.
- Load the model with a dedicated ML serving framework like TF Serving [66] or Clipper [18]. These frameworks handle quite well model deployment and support commonly needed model deployment features, such as A/B model testing, security and model versioning. They also provide optimizations, such as batching for performing inference on GPUs. TF Serving uses NVIDIA's TensorRT [77] that provides further optimizations for achieving low latency and high-throughput for deep learning inference applications on NVIDIA's GPUs. Deployment frameworks generally expect a model in a serialized format like ONNX [7] and do not provide in-place model changes.

We identify that there exist a lot of frameworks dedicated to serving DL models after training on batch data, but not one that handles training on continuous data streams. ML models trained on streams are generally required to train and infer in the same environment. The main reason that the continuous deployment of modern DL models is widely unexplored is because modern DL models generally feature several millions to several billions of parameters, which makes it hard to imagine updating a remotely deployed model in a continuous fashion.

4 Continuous Training and Deployment of DL models

In this chapter, we first describe proactive training as a method that enables the continuous training for DL models in Section 4.1. In the same section, We perform a complexity analysis comparing full retraining, online training and proactive training. We conclude the section with important considerations that should be when proactive training is used. Later, we identify that the continuous deployment is bottlenecked by the communication cost and propose to use a sparse training method for DL in order to reduce the deployment cost. (Section 4.2).

4.1 Continuous Training of DL Models

Our method for continuously training DNNs is inspired by the proactive training approach of Derakhshan et al. [23]. In proactive training, an ML model is updated using mini-batch SGD with mini-batches formed by combining new data combined with samples of historical data. After used for training, new data become part of the historical dataset.

To apply the proactive training (Algorithm 2) for DNNs, we define trigger size t as the count of new elements that will trigger the execution of a training iteration. The input data for the proactive training algorithm are the model's parameters θ, the historical training dataset D and a data stream s, which provides a continuous stream of new training data. The hyper-parameters of proactive training are the learning rate a and mini-batch size b similar to the mini-batch SGD algorithm (Algorithm 1), and also the trigger size t. A mini-batch of size $b > t$ will be constructed by combining t new elements fetched from the stream (Line 1) with a sample of $b - t$ historical elements (Line 3). Having this mini-batch, a normal mini-batch SGD iteration is triggered (Lines 3 and 4), similarly to Algorithm 1. After the t new elements are used for the first training iteration, they subsequently become part of the historical dataset (Line 5) and are subject to be sampled in future iterations.

Algorithm 2: Proactive SGD iteration

```
Input: Learning rate $\alpha$, Mini Batch Size $b$, Trigger size $t$
Data: Model Parameters $\theta$, Historical Dataset D, Data Stream $s$
Result: Updated Model Parameters $\theta$, Updated Historical Dataset $D$
1 Fetch t new examples from the stream $\{x^{(1)}, ..., x^{(t)}\}$, with labels $\{y^{(1)}, ..., y^{(t)}\}$
2 Sample b-t examples from D $\{x^{(t+1)}, ..., x^{(b)}\}$, with labels $\{y^{(t+1)}, ..., y^{(b)}\}$
3 $g = \frac{1}{b}\sum_{i=1}^{b} \frac{\partial}{\partial\theta} L(f(x_i, \theta), y_i)$        // calculate gradients
4 $\theta = \theta - \alpha * g$        // update model's parameters
5 $D = D \cup \{(x^{(1)}, y^{(1)}), ..., (x^{(t)}, y^{(t)})\}$        // add data from stream to the dataset
```

This training method as described above 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.

This balance can be seen as a way to control parts of the approximation-estimation-optimization trade-off as described by [12]. This trade-off is guided by the decomposition of the *excess error* ϵ which measures how closely a learning model approximates the optimal model of the data generation process as shown in Equation 4.1.

$$\epsilon = \epsilon_{app} + \epsilon_{est} + \epsilon_{opt}, \tag{4.1}$$

where ϵ_{app}, ϵ_{est}, ϵ_{opt} are defined as follows:

- The **approximation error** ϵ_{app} measures how closely the function that is learnable by a model can approximate the true function. This error can be reduced by choosing more complex models that represent larger families of learnable functions.
- The **estimation error** ϵ_{est} measures the effect of minimizing the empirical risk (training error) instead of the expected risk (test error). This error can be reduced with more training data or simpler models.

- The **optimization error** ϵ_{opt} measures the impact of the approximate optimization on the expected risk. This error can be reduced by running the optimization for more time. It becomes significant when training on a time budget on large-scale datasets.

In that setting, the proactive training approach does not affect the approximation error, but provides a way to balance between the estimation error and the optimization error by controlling the trigger size:

- A small trigger size (close to zero) will do more optimization iterations and therefore reduce the optimization error, but may potentially increase the estimation error as it can enlarge the gap between the training error and the test error.
- A large trigger size (close to the batch size b) will minimize the estimation error as the online gradient descent minimizes the true generalization error, that is the expected risk. At the same time, it will keep the optimization error high because the optimizer will be triggered less frequently.

From another perspective, the proactive training can help a model learn from new data as they arrive, while not forgetting the training on past data. It can be seen as a way to prevent catastrophic forgetting, also called catastrophic interference [62]. In the domain of continual learning, we find rehearsal-based learning [33, 62] to have some similarity with proactive training. In rehearsal-based approaches, characteristic examples are kept in buffers for continuing training together with new data as they arrive.

4.1.1 Complexity Analysis: Full Retraining Vs Online Training Vs Proactive Training

In this subsection, we compare the complexity of the different training methodologies for continuing to update DL models after the initial training. We assume that the initial training has been executed for e epochs on a dataset D (size $|D|$) with a mini-batch SGD optimizer with a batch size of b. Therefore, the initial training has performed $e * \frac{|D|}{b}$ iterations.

Our analysis includes the following (re)training approaches:

- Full Retraining: Once c new elements are available, a new training from scratch for e epochs is triggered.

- Online Training: Once b new elements are available, an online training iteration is triggered.
- Proactive Training: Once t new elements are available a proactive training iteration is triggered.

Our analysis is concerned with the number of iterations as a function of epochs e, number of initial training elements $|D|$, the batch size b, the c number of new elements that trigger a full retraining and the t number of new elements that trigger a proactive training iteration.

Full Retraining

We assume that a full retraining is triggered as soon as c new elements are available.

Then at each retraining, we have the following number of iterations:

- the 1^{st} retraining does $e * \frac{|D|+c}{b}$ iterations
- the 2^{nd} retraining does $e * \frac{(|D|+2*c)}{b}$ iterations
- the n^{th} retraining does $e * \frac{(|D|+n*c)}{b}$ iterations

In total, after $n*c$ new elements are available, full retraining does $\frac{n*|D|*e*c}{b}+\frac{n*(n+1)*e*c}{2*b}$ iterations, which is $\mathcal{O}(\frac{n*|D|*e*c}{b}+\frac{n^2*e*c}{b})$ iterations.

Online training

With online mini-batch SGD, an iteration of size b is triggered every time b new elements are available ($b \ll c$). For every c elements that enter the system, online training performs a constant number of $\frac{c}{b}$ iterations.

In total, after $n * c$ new elements are available, $\frac{n*c}{b}$ iterations. Online mini-batch SGD will perform $\mathcal{O}(\frac{n*c}{b})$ iterations.

Proactive training

With proactive training, a mini-batch iteration of size b is triggered every time t new elements are available ($t < b \ll c$). For every c elements that enter the system, proactive training performs a constant number of $\frac{c}{t}$ iterations, i.e. $\frac{b}{t}$ times more iterations than online mini-batch SGD.

In total, after $n * c$ elements are available, proactive training performs $\frac{n*c}{t}$ iterations, $\frac{b}{t}$ times more than online training. Proactive training will perform $\mathcal{O}(\frac{n*c}{t})$ iterations.

Summary

This analysis demonstrates two main advantages of online and proactive training over full retraining:

- The number of iterations for online and proactive training does not depend on the size of the historical database $|D|$.
- The number of iterations for online and proactive training grows linearly as with the number of newly available elements, while for full retraining it grows quadratically.

All of the discussed approaches can stop the training when convergence criteria have been met. The complexity analysis given in this section does not account for the fact that as more data are available for training, it is possible to reach convergence in less epochs than in the initial training. Although this is not the typical case in the real-world where new data exhibit shifts in the distribution. Given the non-convexity of DL, the analysis cannot include convergence and therefore it should be considered with a certain degree of uncertainty. As it is often the case in DL, the final verdict is given by the empirical results (Chapter 5), which back up this analysis.

4.1.2 Considerations regarding the use of Proactive Training

In this subsection we identify some limitations of our continuous training approach that should be taken into account when using it.

Estimator bias towards old samples

The estimator becomes biased towards old examples, as they are expected to be chosen for training more times than new data. This bias becomes asymptotically insignificant.

Correlation of new data

It is crucial for SGD that the training samples are selected randomly. Some datasets are arranged in such a way that successive examples are highly correlated. Training on new data as soon as they arrive and not selecting them randomly may break the assumption for uncorrelated examples in the mini-batches.

Sampling overhead

A good property of SGD is that it allows for contiguous memory access and simple implementations. Sampling from large datasets can be costly, but required for proactive training. Derakhshan et al. [23] propose to preprocess and proactively materialize training features in order to avoid part of this cost. Periodic shuffling before selecting mini-batches in order has been shown to not hurt the convergence of SGD. This becomes more challenging when new data constantly enter the system, but we can imagine extensions of the approach that use it instead of direct sampling. Nevertheless, sampling optimization is out of the scope of this study, but needs to be addressed in a future work.

4.2 Sparse Continuous Deployment of DNNs

After training, a DNN is typically deployed in an environment where it will serve prediction queries. Continuous training as described in Section 4.1 implies continuous deployment, which in our case means transferring 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 (c_{depl}) is decomposed as the sum of the communication cost(c_{comm}) and the loading cost (c_{load}). 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 ($c_{comm} \gg c_{load}$) and we can therefore end up in Equation 4.2, which says that the deployment cost is almost equal to the communication cost.

$$c_{depl} = c_{comm} + c_{load} \overset{c_{comm} \gg c_{load}}{\simeq} c_{comm} \tag{4.2}$$

Thus, 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 [76], at every iteration only k out of N ($k < N$) total gradients are selected for updating the model while the rest are kept accumulating in memory. Provided that the selection of the gradients is done via appropriate operators, sparse SGD offers the same convergence rate as regular SGD. Among the appropriate operators, the ones that have been shown to be useful and are used in this study are the following:

- Random-k selector: Selects k_l out of N_l parameters from each DNN layer l uniformly at random.
- Top-k selector: Selects the k_l parameters out of N_l parameters from each DNN layer l with the greatest absolute value.

As such, to facilitate continuous deployment, we propose to modify the continuous training method of 4.1 to perform sparse updates at every iteration. This way, at every training iteration only a small constant percentage of the total model parameters is changed, which allows them to be sent over the network for immediate deployment. Algorithm 3 presents this training procedure. Similarly to the mini-batch SGD, this algorithm takes as input a dataset D, the model's parameters θ, the learning rate α and the mini-batch size b. Additionally, it receives the sparse selection operator $selector_k$ ($random_k$ or top_k) and the memory of gradients which keeps accumulated unused gradients from past iterations. The gradients g are calculated normally as in the simple mini-batch SGD Algorithm 1 (Lines 2, 3) and then summed with the memory of past unused gradients m_{grad} (Line 4). Then, a selection operator $selector_k$ is applied to select k number of gradients out of the total N to be applied for the model update (Line 5). The model gradients are applied locally in the machine that performs the training (Line 6), but also sent over the network to update the model deployed for answering inference queries (Line 7). A memory of the same size as the model's parameters is kept to accumulate unused gradients ($g - g_{sparse}$) at each iteration (Line 8).

The extension of this Algorithm 3 to perform proactive training happens by changing the mini-batch to contain t new elements from the input stream and $b - t$ sampled elements from the historical database. 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 to directly use 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.

Algorithm 3: Sparse SGD with memory for training and deployment

```
Input: Selection operator $selector_k$ , Learning rate $\alpha$, Mini Batch Size $b$
Data: Dataset D, Model Parameters $\theta$, Memory of Gradients $m_{grad}$
Result: Updated Model Parameters $\theta$, Updated Memory of Gradients $m_{grad}$
1 while stopping criteria not met do
2     sample a b examples from dataset D $\{x^{(1)}, ..., x^{(b)}\}$, with labels $\{y^{(1)}, ..., y^{(b)}\}$
3     $g = \frac{1}{b} \sum_{i=1}^{b} \frac{\partial}{\partial \theta} L(f(x_i, \theta), y_i)$          // calculate gradients
4     $g = g + m_{grad}$          // add memory of old unused gradients
5     $g_{sparse} = selector_k(g)$          // select k gradients
6     $\theta = \theta - \alpha * g_{sparse}$          // update model's parameters
7     send $g_{sparse}$ to update deployed model
8     $m_{grad} = m_{grad} + \underbrace{(g - g_{sparse})}_{\text{unused gradients}}$          // update memory of unused gradients
9 end
```

4.2.1 Hyper-parameters of Sparse Continuous Deployment

In this subsection, we discuss about the important hyper-parameters of Sparse Continuous Deployment:

- **The selector operator** (top_k vs $random_k$): In theory, both the $random_k$ and the top_k can achieve the same convergence rate. However, in practice top_k has been shown to achieve better performance [76], as the relative gradient magnitude is thought to provide a simple heuristics for gradient importance [58].

  Another important consideration when choosing a selection operator is that of the **overhead** it incurs in the training process. To the best of our knowledge, this has not been studied and we will therefore examine it with an empirical study in Chapter 5.

- **The sparsification ratio**: Previous work has shown that sparse SGD with memory can converge fast under arbitrarily large sparsification, with Deep Compression [58] achieving good convergence reducing 99.9% of the communicated gradients. Later, in chapter 5 we include an experimental study of how the sparsification ratio affects model quality.

5 Evaluation

In this chapter, we present the evaluation of the continuous training and deployment methods discussed in Chapter 4. Continuing in Section 5.1, we talk about the questions that we address in our experiments and discuss the methodology we use to achieve this goal. Then, we present our experiments' configuration, talking about the hardware and software used and our choice of hyperparameters (Section 5.2). In Section 5.3, we present our experiments with a simple model (LeNet-5 [57]) and a simple dataset (MNIST [53]). Later, in Section 5.4 we present the results of training state-of-the-art DL models (Mobilenet [72], Resnet [34], Densenet [39]) in a more complex dataset (CIFAR10 [50]). Both sections with experiments (Sections 5.3 and 5.4) are structured in the same way: i) firstly we introduce the dataset and models used in a subsection called Dataset and Model(s), ii) secondly, we give an overview of the experiment setup in a subsection called Setup iii) thirdly, we present the results in a subsection called Results iv) and finally we discuss about implications of our results in a subsection called Discussion. We conclude this chapter with a discussion (Section 5.5) which summarises our experimental results, our related findings and the shortcomings of our experiments that should be addressed in future studies.

5.1 Experimental Methodology

With our experiments we would like to answer the following questions:

- How do full retraining, online training and proactive training compare in terms of a) model quality, b) training time?
- How does the trigger size t affect the proactive training approach in terms of a) model quality, b) training time?

- For sparse continuous deployment, how far can we sparsify a model's changes per iteration without sacrificing model quality?

- What is the overhead of sparse continuous deployment for a) different selectors, b) varying sparsification ratios?

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 (re)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.

- Similarly to online training, for the proactive training approach, the model is also warm-started with the generated model from the initial training. As presented in Section 4.1, a proactive training iteration is triggered once t new elements are available. We examine the effects of changing the trigger size t, in terms of training time and accuracy.

For the proactive training approach, we leverage sparse SGD with memory as presented in 4.2 to compress model changes enough to facilitate Sparse Continuous Deployment. We examine how the sparsification ratio and the choice of the selection operator affect the overhead of the Sparse Continuous Deployment and the model quality of the deployed model.

For evaluating each one of the approaches we use the **prequential evaluation** [20], which is a common method to evaluate ML algorithms on streams. Its idea is quite simple - every example in the stream is first used to test the model, and then to train it.

5.2 Configuration

This section presents the software and hardware we used to run our experiments (Section 5.2.1) and our choice of the hyperparameters that the experiments used (Section 5.2.2). The code implements our solution for continuous training and deployment as shown in Figure 1.3.

5.2.1 Software and Hardware

We run all experiments in 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 [68] framework (version 1.2.0). The state-of-the-art DNN architectures that are used in Section 5.4, have been imported from the torchvision [61] library (version 0.4.0).

5.2.2 Hyperparameter Choice

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 our experiments, we use the same training configuration:

- As an optimizer, we use ADAM [47] 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 dataset (which we extensively use in our experiments) for several DL models [52].

5.3 Experiments on MNIST with LeNet

To validate our approach, we start our experiments with a simple DL model (LeNet-5 [57]) and a simple dataset (MNIST [53]). We start by introducing the MNIST dataset and LeNet model in

Section 5.3.1, then present the experimental setup (Section 5.3.2), before diving into the results (Section 5.3.3). We end this section with a discussion on the implications of our experiments (Section 5.3.4).

5.3.1 Dataset and Model

In the next paragraphs we give an overview of the MNIST dataset and the LeNet-5 model.

The MNIST Dataset

The MNIST dataset contains 28x28 grayscale images of handwritten digits as shown in Figure 5.1. It has 60,000 training examples and 10,000 testing examples. According to the website of the dataset [53], "it is a good database for people who want to try learning techniques and pattern recognition methods on real-world data while spending minimal efforts on preprocessing and formatting."

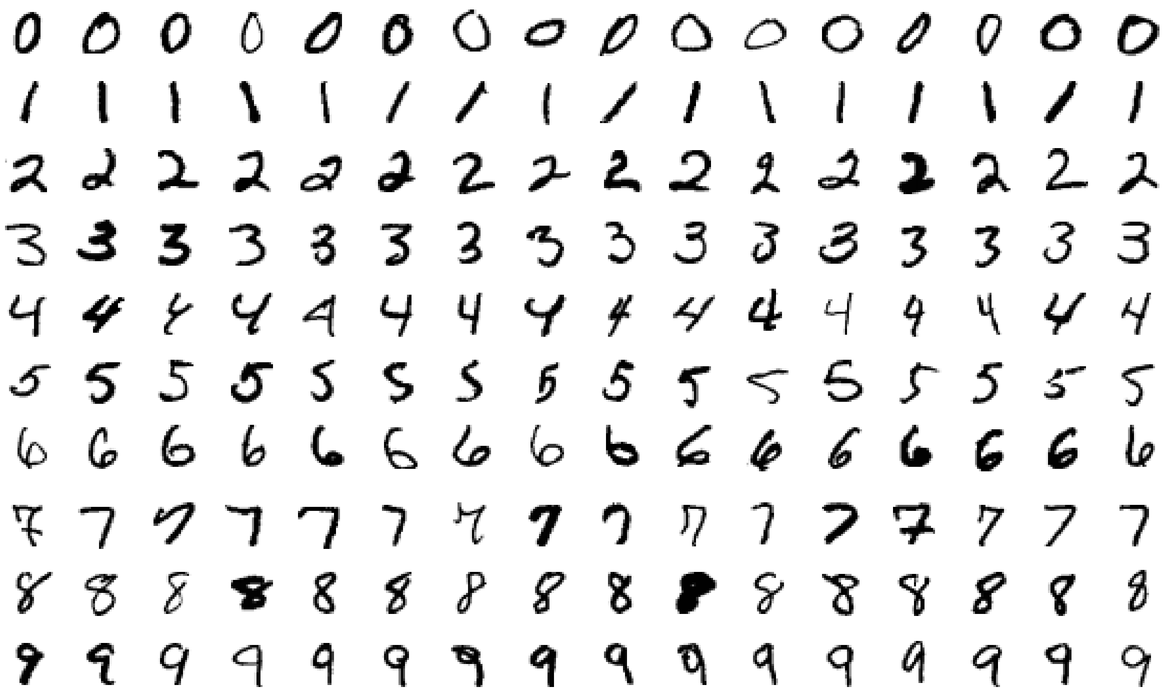


Figure 5.1: MNIST dataset examples [80]

As per the experimental methodology discussed in Section 5.1, we use the first 10,000 training examples for the initial training. Then we make the rest of the training dataset available in a streaming fashion and evaluate with prequential evaluation.

The LeNet-5 Model

For the MNIST experiments we use the LeNet-5 [57] architecture (Figure 5.2), which represents a seminal CNN architecture. It features two sets of convolutional and pooling layers, followed by a flattening layer, then two fully-connected layers and finally a classifier layer. As noted in Andrew Ng's Deep Learning Specialization [65] course on CNNs this early CNN architecture introduced two patterns that are quite common even for modern CNNs: i) as you go deeper in a network, from left to right in Figure 5.2 the height and width of the features is reduced and their number of channels is increased, and ii) a CNN consists of one or more blocks of convolutional layers followed by a pooling layer and at the end some fully connected layers before the outputs. For the pooling layers, we use max pooling instead of average pooling and as an activation we use the ReLU instead of the sigmoid function, since these choices are more common these days. Moreover, the original LeNet-5 model uses an output layer that is not useful by today's standards [65] and thus we replace it with a softmax layer. After these changes, our implementation follows closely the implementation of a DL2AI tutorial [19].

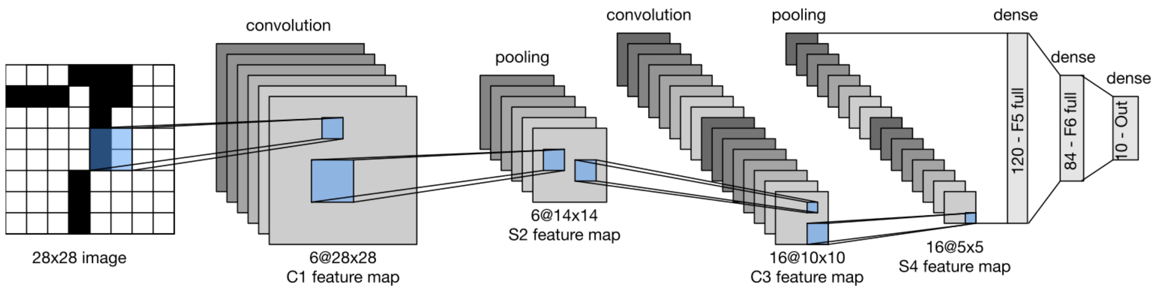


Figure 5.2: LeNet-5 Architecture [19]

5.3.2 Setup

The first 10,000 examples are used for the **initial training** which is done for 10 epochs. The initially trained model is used to warm-start both online and proactive training. Also, the parameters of the ADAM optimizer are warm-started.

Full retraining is triggered every new 10000 data points that become available and is done from scratch for 10 epochs. More specifically, after the initial training a full retraining is triggered when 20,000, 30,000, 40,000 and 50,000 new training examples become available. A trained model on points with indexes $[0, x)$ - where $x = 10,000, 20,000, 30,000, 40000, 50000$ - is responsible for inference on the next 10000 points, i.e. the points with indexes $[x, x + 10,000)$.

Online training triggers one mini-batch SGD iteration once a mini-batch size of $b = 128$ new elements are available. According to 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 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) different sparse selectors (top_k and $random_k$), iii) and different sparsification ratios ($1\%, 0.1\%, 0.01\%$).

5.3.3 Results

In this subsection, we present the results of our experiments. We split the presentation as follows:

- We first compare the full retraining to online training and proactive training in terms of prequential accuracy and total training time.
- Then, we compare the overhead, reduction in communication parameters and prequential accuracy using different sparsification strategies (top_k, $random_k$) and sparsification ratios.

Full Retraining Vs Online Training Vs Proactive Training

This first set of experiments compares the (re)training approaches in terms of model quality (Figure 5.3) and total training time (Figure 5.4).

In terms of model quality, the full retraining approach improves only at the points where the retraining is triggered (every $10,000$ data points annotated with RT), while proactive and online training improve continuously with every mini-batch SGD update. This allows the online training to start better than the full retraining, but full retraining quickly catches up after the 2^{nd} retraining is triggered. After that, full retraining continues to grow the gap with online training. Proactive training's prequential accuracy becomes increasingly better with smaller trigger sizes, that enable for an increasing number of mini-batch SGD iterations. With a trigger size of 64 that performs $\frac{128}{64} = 2$ times more iterations than online training, proactive training significantly surpasses the online training, achieving better model quality than full retraining early on in the training process, and is only overtaken by the full retraining around the $40,000^{th}$ data point where the 3^{rd} full retraining is triggered. With trigger sizes lower than 64, namely 32, 16 and 8, proactive training consistently achieves better prequential accuracy than all the other approaches.

In terms of total training time (Figure 5.4) the full retraining approach is the slowest at 69.30 seconds and online training is the fastest at 2.63 seconds. Proactive training performs exactly $\frac{batch\ size}{trigger\ size}$ times more iterations than online training and approximately this much more time (see Figure 5.5). Even with a trigger size of 8 - the lowest we tried - proactive training's total training time is about half of that of the full retraining. In short, proactive training is in between the two other approaches, achieving training times linearly increasing with the number of iterations, which depends on the chosen trigger size.

In short, online training is the fastest method but falls short in model quality. The full retraining approach results in good models at the points that it is triggered, but due to its periodical nature these models quickly become stale and therefore fall behind the proactive training which improves continuously combining both historical and new data in its training process. The proactive training is able to achieve the best prequential accuracy at a fraction of the time of full retraining. The total training time of proactive training increases by a factor that approximates $\frac{batch\ size}{trigger\ size}$ compared to the online total training time.

Sparse Deployment

For the following set of experiments, we use the proactive training method that achieves the best prequential accuracy (proactive training with trigger size 8). We extend it to do sparse training iterations as explained in Chapter 4 in order to facilitate sparse continuous deployment, and examine the effect of different deployment settings.

More specifically we compare the different sparse selectors, namely top_k and $random_k$, with

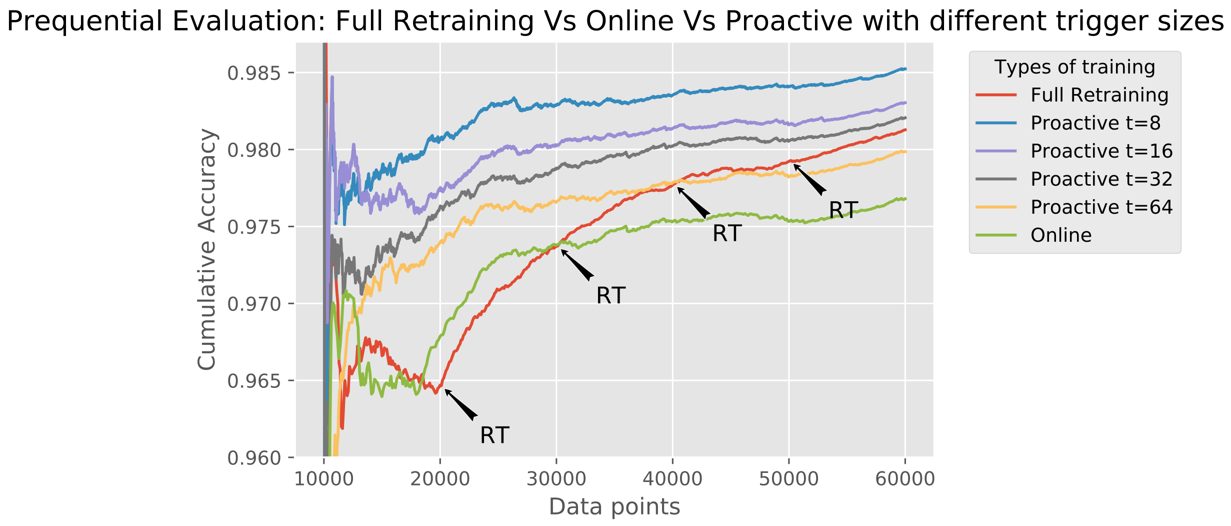


Figure 5.3: Cumulative accuracy of prequential evaluation of full retraining, online training and proactive training with different trigger sizes (8, 16, 32, 64). Retraining points are marked with the "RT" arrows.

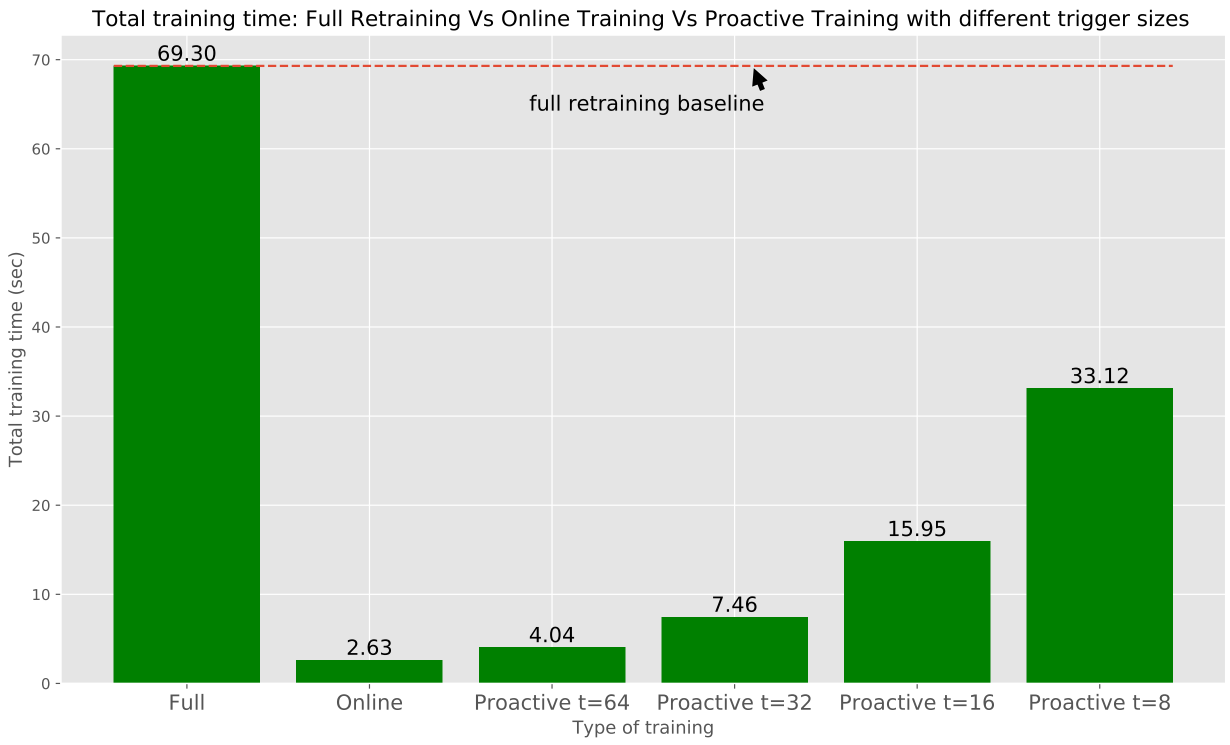


Figure 5.4: Total training time of the different (re)training approaches

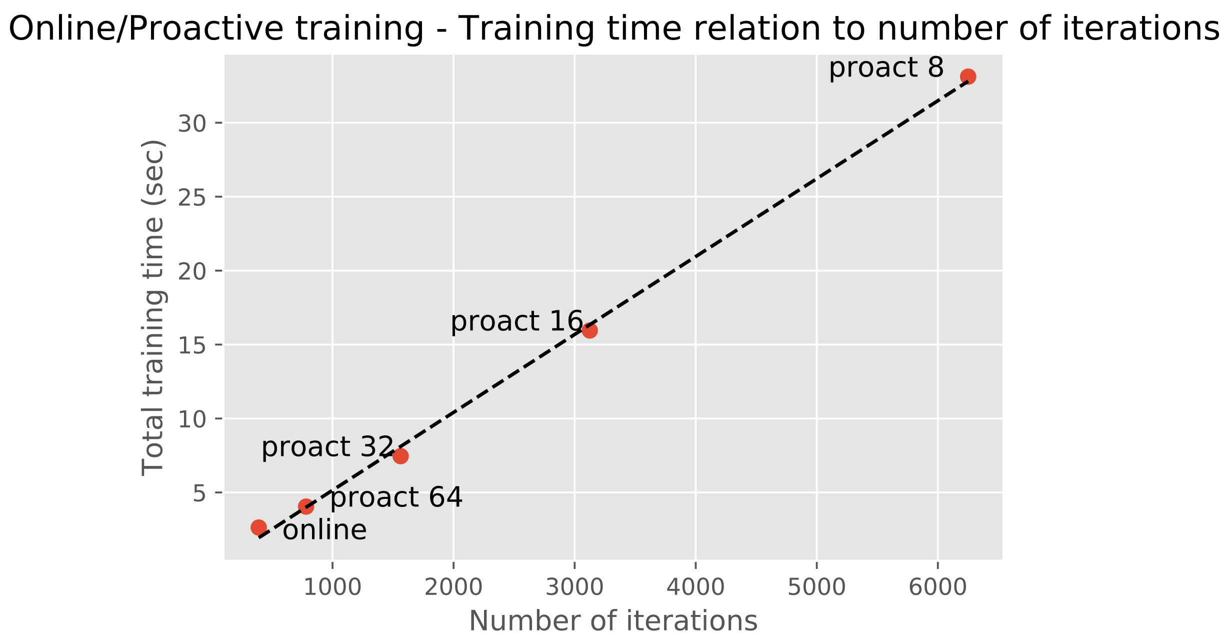


Figure 5.5: Total training time as a function of mini-batch SGD iterations for online and proactive training. Online training is annotated with the label online and proactive training with the label proact t (for trigger size t = 8, 16, 32, 64).

varying sparsification ratios. For the sparsification ratios we choose 0.01, 0.001 and 0.0001, which means that only 1% (0.01), 0.1% (0.001) and 0.01% (0.0001) of the gradients per layer are used for updating the model at each iteration. The least amount of gradients selected by the sparse selectors is 1 even the ratio dictates less than 1 selections. Our experiments report i) the cumulative prequential accuracy as a proxy of the achieved model quality (Figures 5.6 and 5.7), ii) the total training time, which shows the overhead of the chosen settings when compared to the training without sparsity (Figure 5.8) and iii) the number of model parameters changed (non-zero) per iteration (Figure 5.9).

Figure 5.6 shows the effect of varying sparsification ratios (1%, 0.1%, 0.01%) for the sparse selector top_k. We see that all approaches perform slightly worse than proactive training without sparsity and better than full retraining. Between top_k 1.0% and top_k 0.1% there is no visible difference at the final achieved prequential accuracy while top_k 0.01% performs slightly worse than the other two. It seems as though the sparsification ratio below a certain threshold degrades the performance, as it can delay important parameter changes.

Figure 5.7 shows the effect of varying sparsification ratios (1%, 0.1%, 0.01%) for the sparse selector $random_k$. All $random_k$ settings perform worse than full retraining with the greater sparsification (smaller ratio) performing increasingly worse by small margins. This performance difference compared to the full retraining is caused due to a big drop in prequential accuracy during the training on the first 10,000 data points. In retrospect, this may be due to the warm-started parameters of the ADAM. ADAM was used to train with dense gradients in the initial training and now it is called to continue the process with sparse gradients. This is probably not very obvious with the top_k sparse operator which manages to select the most "important" parameters and recuperates faster. The drop is bigger when the enforced sparsity is smaller, but at the same time the curves of smaller sparsity reach slightly higher cumulative accuracy at the end. In the beginning the warmup of ADAM has a higher effect when the sparsity is smaller. Later in the prequential accuracy curves, as more changes happen per iteration less sparse variants are able to recuperate faster.

Figure 5.9 shows the average number of parameters changed per iteration for each one of the sparsification strategies. We report the number of non-zero gradients. This is why top_k chooses slightly more parameters than $random_k$. Also why the average number of parameters changed per iteration is lower than the total number of the parameters. After clarifying this, the results are as expected, meaning that with sparsification ratio of 1%, 0.1% and 0.01%, the parameters changed at each iteration are reduced by one, two and three orders of magnitude, respectively. The deployment cost is also reduced by the same corresponding factors. Later, in the discussion (Section 5.3.4) we discuss about how the effect of the sparse training on the communication cost.

An important difference between top_k and $random_k$ is that $random_k$ worsens the initial trained model at the beginning of the training, before making it better. In addition to its worse performance in terms of quality, $random_k$ is also slower than top_k. Both strategies for sparse proactive training incur an overhead in training time (Figure 5.8) compared to their non-sparse variant. The top_k sparse proactive training increases the total training time by a factor of around 1.5, while the $random_k$ variant is increases it by a factor of around 2.35 over the non-sparse variant. A significant part of the big overhead of $random_k$ has to do with our implementation: PyTorch 1.2.0 does not support native random sampling without replacement and therefore we ended up using a numpy implementation before converting to a PyTorch tensor. The reason that we do not continue using the $random_k$ for the rest of our experiments has more to with each performance in prequential evaluation, rather than its speed which could be improved with a better implementation.

In a streaming environment, we are interested to maintain good model quality at all times and induce minimal overhead; as such the $random_k$ selector does not look like it fits the deployment use case. This early experiment shows that the top_k sparse selector is more robust for continuous deployment. Regarding the sparsification ratio, the best ratio for continuous deployment is the lowest one that does not sacrifice model quality. Our experiments showed we can reduce the communicated gradients by 99.99% with minimal compromises in prequential accuracy.

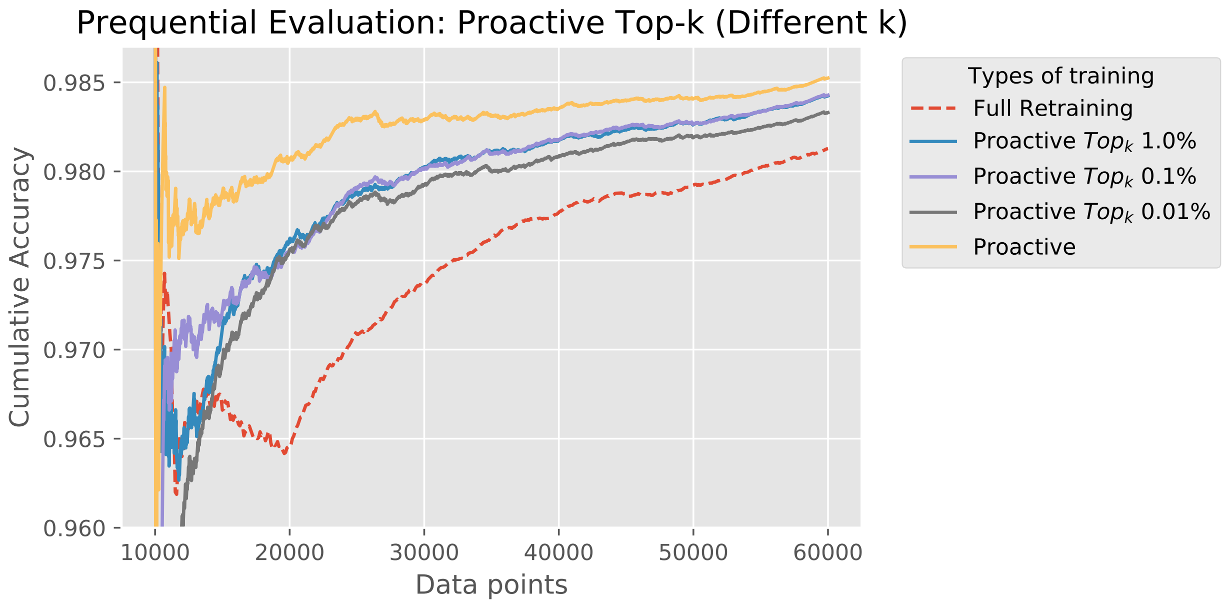


Figure 5.6: Cumulative accuracy of prequential evaluation of full retraining and proactive training without sparsification and with top_k sparsification with varying ratios. All types of (sparse) proactive training have trigger size 8.

5.3.4 Discussion

In this section, we have presented our experiments and their results with LeNet DL model on MNIST dataset. We show that proactive training can offer a balance in training time between online training and full retraining while consistently offering the best prequential accuracy with trigger sizes less than 64. As discussed in Section 4.1, controlling the trigger size is a way to control this balance between online training and full retraining. Trigger sizes close to the batch size give a training time and performance similar to online training. Lower trigger sizes give a higher training time, approximately $\frac{batch\ size}{trigger\ size}$ larger than online training, accompanied with a better accuracy.

Meanwhile, even with extreme sparsification, transferring only 13 out of 61,706 parameters per

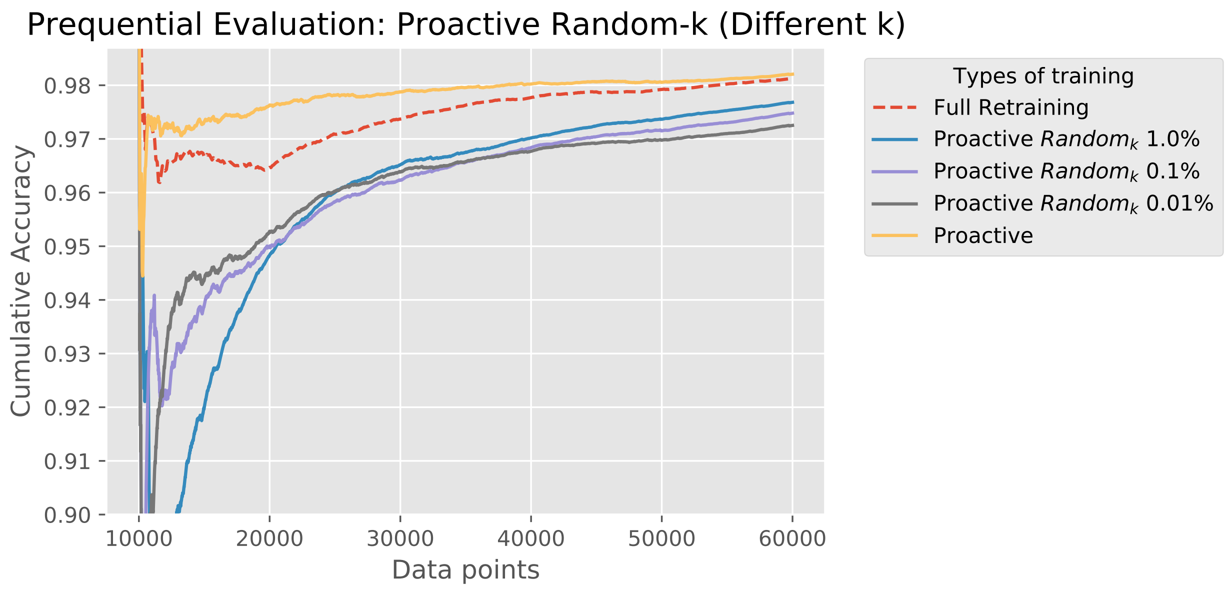


Figure 5.7: Cumulative accuracy of prequential evaluation of full retraining and proactive training without sparsification and with $random_k$ sparsification with varying ratios. All types of (sparse) proactive training have trigger size 8.

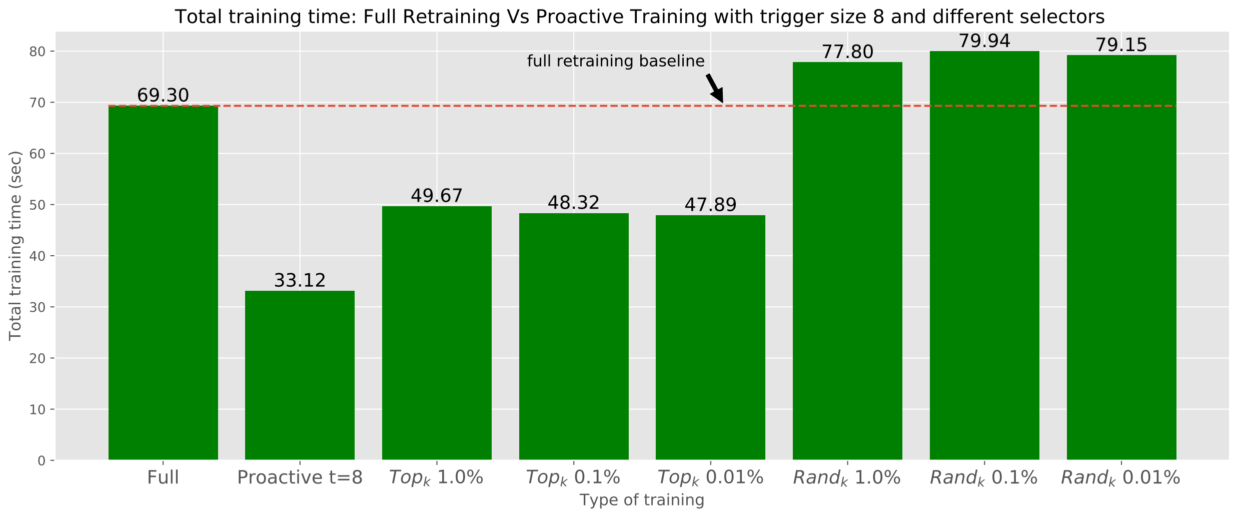


Figure 5.8: Total training time for full retraining and proactive training without sparsity and with the different sparsification strategies. All types of proactive training have trigger size 8.

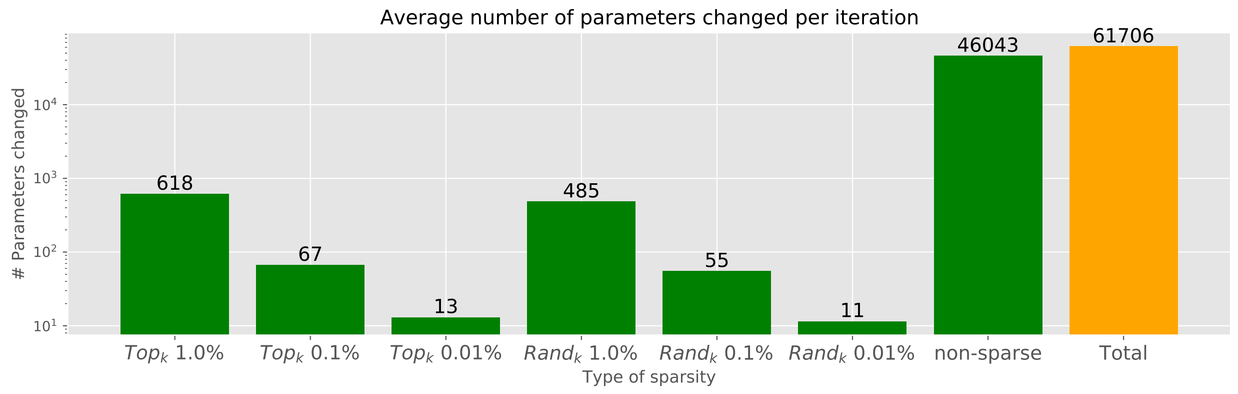


Figure 5.9: Average number of parameters changed per SGD iteration proactive training without sparsity and with the different sparsification strategies

iteration and accepting an overhead compared to non-sparse training, we can keep the model up-to-date without any losses in model quality or convergence rate. That is true when using the top_k sparse selector which selects the largest gradients by magnitude at each iteration. Our experiments confirm that the magnitude of the gradients does seem to offer a good simple proxy for the importance of the model changes. The $random_k$ sparse selector does eventually lead in good models, but can lead to worse models at the beginning of the training and is slower than top_k. Our $random_k$ sparse selector is slow in part because the PyTorch library does not offer an optimized function for choosing a random sample without replacement. Our implementation uses the sample function of numpy and then converts to a torch tensor. We suspect that the models are initially worse due to the instability of choosing randomly a small number of parameters to update at every iteration. Continuing this random sampling process eventually converges to a good solution as the non-used gradients are accumulating and are subject for selection in future samples. However, there is neither a guarantee nor an expectation for early convergence. Because it can worsen the model at the beginning of the training, sparse proactive training with the $random_k$ selector falls behind full retraining in the prequential evaluation.

Let us analyse how much using the sparse selector top_k reduces the deployment cost. For the sake of simplicity let us assume that the total training time using any the top_k selector is 50s for all the sparsification ratios (it is actually 49.67s, 48.32s and 47.89s for sparsification ratios 1.0%, 0.1% and 0.01% respectively). With a trigger size of 8, it performs $\frac{50{,}000}{8} = 6,250$ iterations. Therefore, a training iteration is performed roughly every $\frac{training\_time}{\#\ iterations} \simeq \frac{50s}{6{,}250} = 8ms$. Let's compare the bandwidth consumed when sending parameters equal to number selected by top_k and its non-sparse variant, considering each parameter is a 32bit float:

- The non-sparse proactive training needs to send all the parameters every 8ms, consuming a bandwidth of $\frac{61{,}706*32\text{bits}}{8ms} = 30.85$ MBps. The non-sparse proactive training actually needs to do this in less time as it is faster than the top_k variant.

- The top_k 1.0% needs to send 618 parameters every 8ms, and also the indices of the parameters that we will assume are stored in 32bit integers. This consumes a bandwidth of $\frac{2*618*32\text{bits}}{8ms} =$ 618 kBps.

- The top_k 0.1% needs to send 67 parameters plus the indices of the changes every 8ms ,consuming a bandwidth of $\frac{2*67*32\text{bits}}{8ms} = 67$ kBps.

- The top_k 0.01% needs to send 13 parameters plus the indices of the changes every 8ms , consuming a bandwidth of $\frac{2*13*32\text{bits}}{8ms} = 13$ kBps.

We see that even for a small network like LeNet-5, the sparsification can greatly reduce the communication cost of transferring model updates over the network. Imagining that the model is deployed in an environment that receives prediction queries, we want the deployment to consume the least bandwidth possible.

This first series of experiments validates our approach for continuous training and deployment. Regarding the continuous deployment, the top_k sparse selector is more promising. Therefore, we choose to use it for the next series of experiments.

5.4 Experiments on CIFAR with Modern DL Models

We continue the evaluation, experimenting with State-Of-The-Art (SOTA) DL models (Mobilenet_v2 [72], Resnet18 [34], Resnet50 [34], and Densenet161 [39]) on a more complex dataset (CIFAR [50]). We start by introducing the CIFAR dataset and the SOTA models in subsection 5.3.1, then present the experimental setup (subsection 5.3.2), before diving into the results (subsection 5.3.3). We end this section with a discussion on the implications of our experiments (subsection 5.3.4).

5.4.1 Dataset and Models

In the next paragraphs we give an overview of the CIFAR dataset, and the SOTA DL models (Mobilenet_v2, Resnet18, Resnet50, Densenet161) used in this set of experiments.

The CIFAR Dataset

The CIFAR-10 dataset [50] consists of 60,000 32x32 colour images in 10 classes as shown in Figure 5.10, with 6000 images per class. There are 50,000 training images and 10,000 test images.

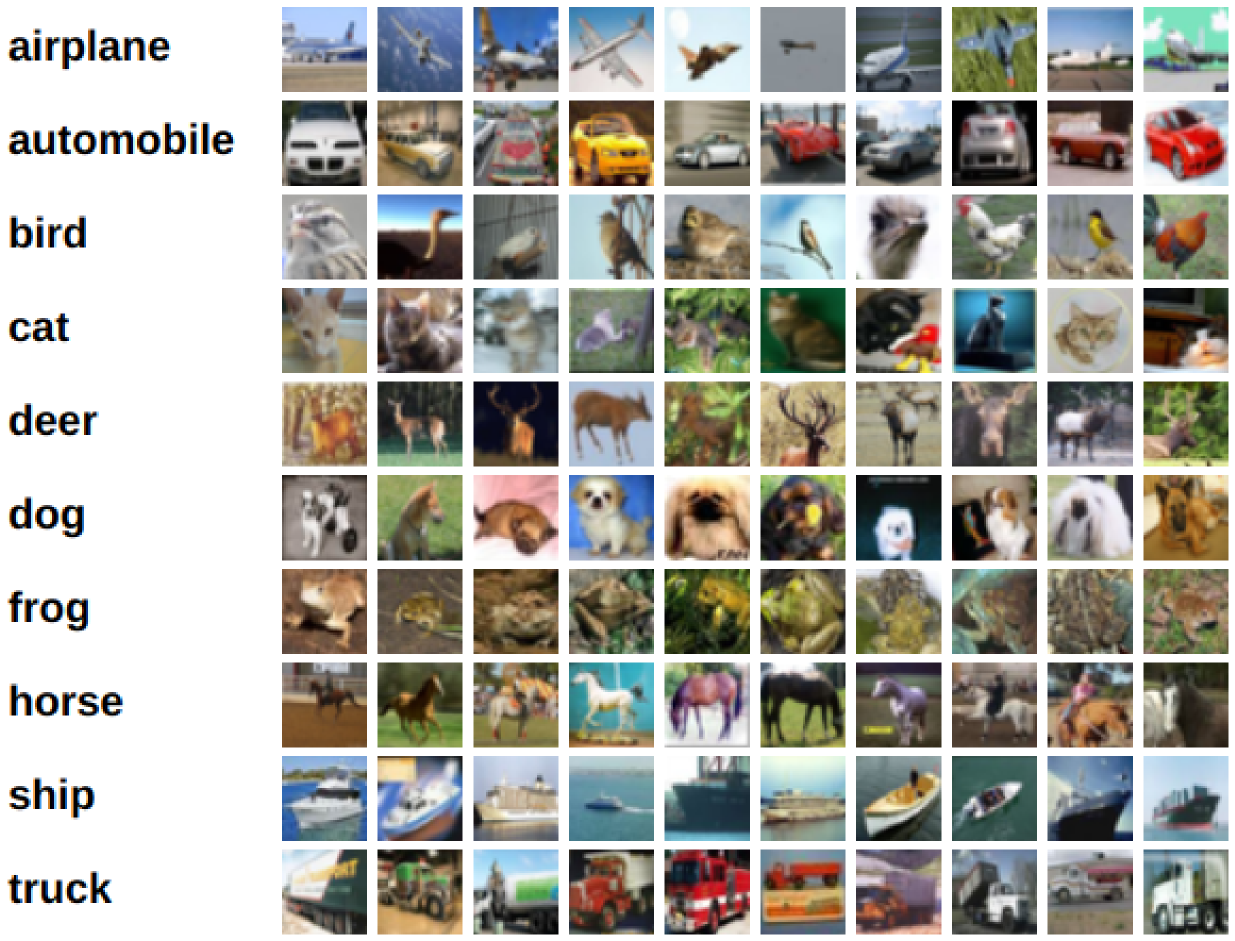


Figure 5.10: Illustration of the classes of CIFAR10 [50]

SOTA DL models

For this set of experiments we use modern DL models with different sizes (Table 5.1) with the following features:

- Mobilenet_v2 [72] is chosen as a compressed DL architecture 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 [34] is chosen as a medium-sized model that includes many SOTA features, such as batch normalization and skip connections.

- Resnet50 [34] is chosen as a very deep alternative of Resnet18.

- Densenet161 [39] is chosen as a very deep SOTA model that is in a different family than the residual networks selected.

We start the training for each one of the models with pre-trained weights on Imagenet [22] 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.

DNN Architecture	**Trainable parameters**	**Depth**
mobilenet_v2 [72]	2,236,682	53
resnet18 [34]	11,181,642	18
resnet50 [34]	23,528,522	50
densenet161 [39]	26,494,090	161

Table 5.1: DNN Architectures used in our experiments and the number of trainable parameters of each one

5.4.2 Setup

The first 10000 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 [22].

Full retraining is triggered every new 10,000 data points that become available and is done for 25 epochs starting from weights of pre-trained models on Imagenet. More specifically, after the initial training a full retraining is triggered when 20,000, 30,000, and 40,000 new training examples become available. A trained model on points with indexes $[0, x)$ (where $x = 10,000, 20,000, 30,000, and 40,000$)

is responsible for inference on the next 10000 points, i.e. the points with indexes $[x, x + 10,000)$. Each full-retraining is warm-started with weights pre-trained on Imagenet [22].

Online training triggers one mini-batch SGD iteration once mini-batch size ($b = 128$) new elements are available. According to 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 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 sparsification ratios ($1\%, 0.1\%, 0.01\%$) only using the sparse selector top_k that proved more promising in the MNIST experiments (Section 5.3).

5.4.3 Results

In this subsection, we present the results of our experiments. We split the presentation as follows:

- We first compare the full retraining to online training and proactive training in terms of prequential accuracy and total training time.
- Then, we compare the overhead, reduction in parameters changed per iteration and prequential accuracy using different sparsification ratios (1%, 0.1%, 0.01%) for the sparse selector top_k.

Full Retraining vs Online Training vs Proactive Training

Figure 5.11 shows the results on prequential evaluation for each model and (re)training approach (full retraining, online, proactive). Table 5.2 shows the corresponding total training times.

Results on CIFAR with modern DL models are quite consistent with the results we got on MNIST with the Lenet-5 simple model. For the total training times there are no surprises. The online training is the fastest method and the proactive training is finished $\frac{batch\ size}{trigger\ size}$ slower than online. Full retraining is by far the slowest, and its gap over the other approaches is bigger in the CIFAR experiments that every retraining does 25 epochs instead of 10 in MNIST. In the prequential evaluation, we see that generally online training is the worst method, followed by full retraining and proactive training offers the best approach with increasing performance as more iterations are triggered (smaller trigger size).

DL Model	**Full**	**Online**	$\boldsymbol{Proact_{64}}$	$\boldsymbol{Proact_{32}}$	$\boldsymbol{Proact_{16}}$	$\boldsymbol{Proact_{8}}$
mobilenet_v2	1381.2	31.6	57.6	129.2	214.9	442.2
resnet18	493.8	13.1	21.1	42.1	94.2	169.8
resnet50	3672.4	65.7	131.7	261.5	524.6	1042.5
densenet161	6864.4	122.1	242.4	486.4	952.1	1923.3

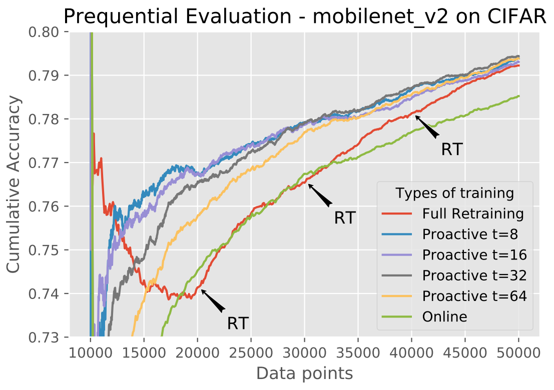
Table 5.2: Total training time in seconds for full retraining (Full column), online training (Online column) and proactive training($Proact_t$ columns for trigger size $t = 64, 32, 16, 8$).

These are the general trends that are consistent with the MNIST experiments of section 5.3, but there are also some major differences with regards to the prequential evaluation:

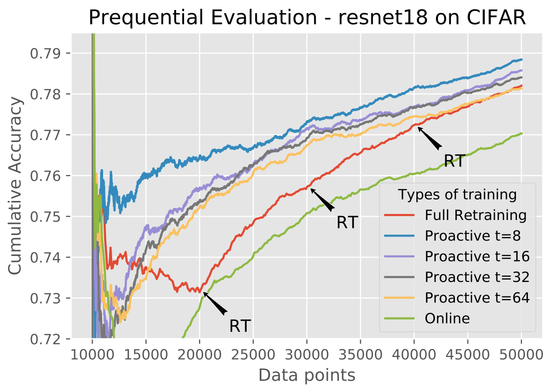
- For all models, it is evident that in the very beginning of the online and proactive training they perform worse than the initial model. Online training takes the longest to recuperate. Proactive training recuperates faster when the trigger size is smaller.
- For the Mobilenet_v2 (Figure 5.11a) there is no clear winner among the proactive training approaches, with respect to the trigger size. Until the 30000^{th} point lower trigger size is better, but then all methods seem to converge at a performance slightly better than full retraining.
- With the exception of Resnet18 (Figure 5.11b), where we see a clear win of the proactive training against 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 (Figures 5.11d 5.11c, 5.11a), we see the proactive training doing clearly better after stabilising at the begin of the training, but then converging to a comparable cumulative accuracy with full retraining.

Sparse Continuous Deployment

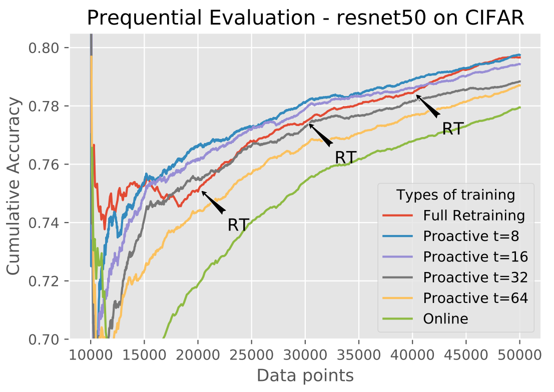
Figure 5.12 shows the results on prequential evaluation for each model and sparsification approach. We use the top_k sparse selector with sparsification ratios 1%, 0.1% and 0.01%, always using proactive training with trigger size equal to 8. Table 5.4 displays the average numbers of parameters



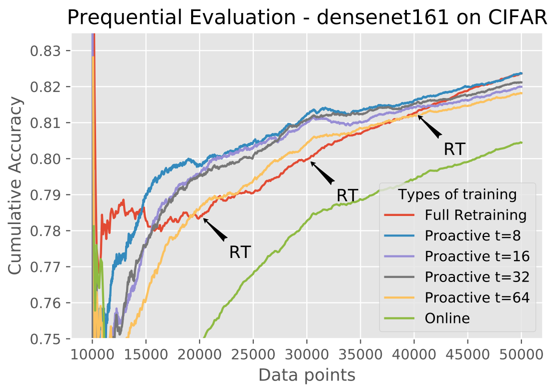
(a) Mobilenet_v2 cumulative accuracy on CIFAR10



(b) Resnet18 cumulative accuracy on CIFAR10



(c) Resnet50 cumulative accuracy on CIFAR10



(d) Densenet161 cumulative accuracy on CIFAR10

Figure 5.11: Prequential evaluation for SOTA DL models on CIFAR. Comparison between full retraining, online training and proactive training with varying trigger sizes.

changed per iteration for each one of the sparsification approaches and the corresponding non-sparse variant. Table 5.3 displays the corresponding total training times together with the full retraining and non-sparse proactive training for comparison.

With respect to the training time, it is evident that the top_k 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 time is on average larger by a factor of 1.48 and 1.7 respectively compared to their the non-sparse variant. Sparse training on mobilenet_v2 exhibits the largest variance between different sparsification ratios with the overhead becoming slightly smaller when fewer parameters are selected for all the networks. For resnet50 and densenet161 the sparse training time 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 (see Table 5.1), the effect of running the top_k sparsification is much bigger for resnet50. This is explained by the fact that resnet50 has a lot 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. For exact numbers, the reader is referred to Table 5.3.

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 (5.12c) with 1% sparsification ratio, which falls behind all the other approaches. In this set of experiments we notice that greater sparsification values (less gradients used per iteration) lead to better prequential performance, even better than the non-sparse variant, forming a larger gap with the full retraining. This is the reverse from what we saw in the MNIST experiments. We suspect that this is due to the fact that the top_k operator manages to select the most "important" gradients per iteration, ignoring non-important ones. We imagine this provides a regularizing effect in the sparse training process.

Results are more or less expected regarding the number of parameters changed (Table 5.4), depending on the sparsification ratio. With sparsification ratios of 1.0%, 0.1% or 0.01%, only 1.0%, 0.1% or 0.01% of the gradients per layer are changed per iteration. We see that for the extreme sparsification 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. The effect of the sparsification on the deployment cost is discussed later in Section 5.4.4.

DL Model	**Full**	$\boldsymbol{Proact_8}$	$\boldsymbol{Top_k}$ **0.01%**	$\boldsymbol{Top_k}$ **0.1%**	$\boldsymbol{Top_k}$ **1.0%**
mobilenet_v2	1381.2	442.2	560.1	679	726.9
resnet18	493.8	169.8	288.3	279.2	305
resnet50	3672.4	1042.5	3001.1	3159.1	3266.5
densenet161	6864.4	1923.3	3870.3	4032	4154.2

Table 5.3: Total training time in seconds full retraining (Full column), proactive training without sparsification ($Proact_8$ columns) and with varying top_k sparsification ratios (Top_k x% columns for $x = 0.01, 0.1, 1$).

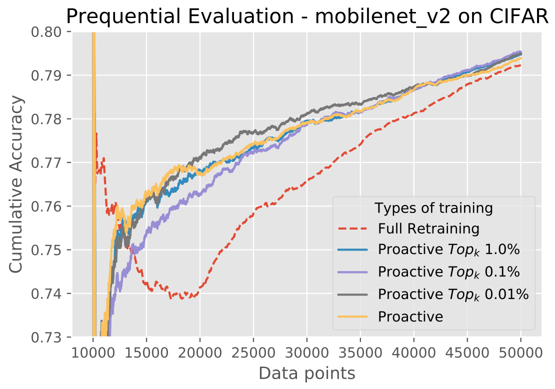
DL Model	**Total Params**	**No Sparsity**	$\boldsymbol{Top_k}$ **1.0%**	$\boldsymbol{Top_k}$ **0.1%**	$\boldsymbol{Top_k}$ **0.01%**
mobilenet_v2	2,236,682	2,110,358	22,305	2,287	333
resnet18	11,181,642	4,068,411	111,800	11,201	1146
resnet50	23,528,522	15,810,578	235,192	23,567	2425
densenet161	26,494,090	15,991,847	264,641	26,498	2886

Table 5.4: Total number of parameters("Total Params" column) and the average number of parameters changed per iteration for proactive training without sparsification ("No sparsity" column) and with varying top_k sparsification ratios ("Top_k x%" columns for $x = 0.01, 0.1, 1$).

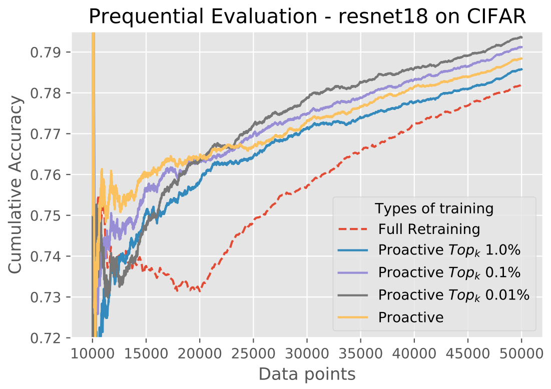
End model quality

In this section, we present results comparing the final model quality measuring accuracy on the reserved test set of 10,000 images. For each one of the models we compare the accuracy between the different (re)training approaches. We compare the models after the last full retraining with the models after the end of online training and proactive training with trigger size 8 without sparsity and with sparsity for different top_k sparsification ratios (1%, 0.1%, 0.01%). The results are shown in Table 5.5.

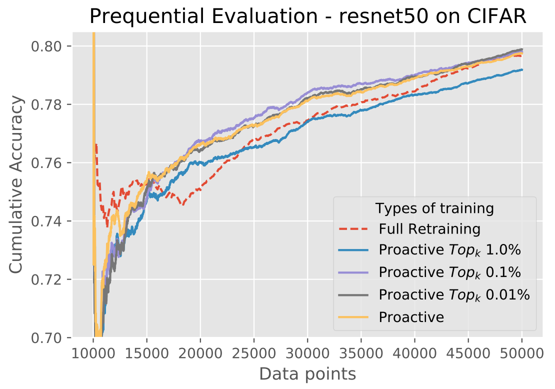
We see that non-sparse proactive training consistently achieves comparable but slightly worse accuracy than full retraining, with online training providing the worst final model quality at all the cases. Sparse proactive training with sparsification 1.0% is generally very close to its non-sparse variant, except for the case of resnet18 where it falls by about 0.04 achieving an accuracy more close to online training. Higher sparsification (0.1% and 0.01%) training outperforms not only the non-sparse variant in most cases, but also the full retraining with all models except for



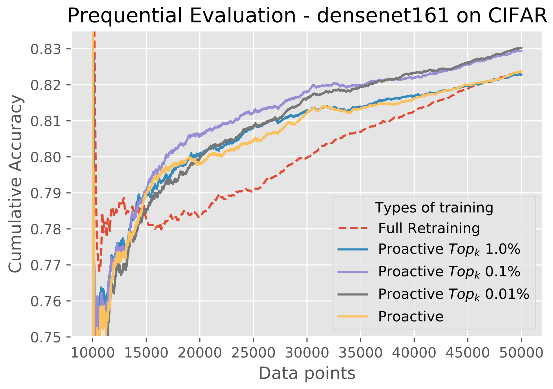
(a) Mobilenet_v2 cumulative accuracy on CIFAR10



(b) Resnet18 cumulative accuracy on CIFAR10



(c) Resnet50 cumulative accuracy on CIFAR10



(d) Densenet161 cumulative accuracy on CIFAR10

Figure 5.12: Prequential evaluation for SOTA DL models on CIFAR. Comparison between full retraining, and (sparse) proactive training with a trigger size of 8 and various sparsification ratios (1%, 0.1%, 0.01%) for the sparse selector top_k.

densenet161. This is probably due to the regularizing effect of the top_k sparse selector as we described it in the previous section. Among the sparse variants, the one with the most extreme sparsification (0.01%) consistently outperforms the others, with the exception of mobilenet_v2, where the proactive training with sparsification 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. For this study, we find no reason to compare the differences in accuracy achieved between different models and will therefore refrain from doing it.

In short, these experiments confirm that the model quality achieved by proactive training are comparable, if not superior to the ones we get from full retraining. It is important to note that the last full retraining has not trained in the last 10000 training examples. This slightly underestimates the full retraining's accuracy when the whole dataset is available, but the comparison is still fair if we think that full retraining will result in stale model's most of the time anyway.

DL model	**Full**	**Online**	$\boldsymbol{Proact_8}$	$\boldsymbol{Top_k}$ **1.0%**	$\boldsymbol{Top_k}$ **0.1%**	$\boldsymbol{Top_k}$ **0.01%**
mobilenet_v2	0.8173	0.7995	0.8129	0.8149	**0.8196**	0.8087
resnet18	0.8047	0.7999	0.8041	0.8003	0.8081	**0.8135**
resnet50	0.8217	0.8146	0.8216	0.8211	0.8224	**0.8273**
densenet161	**0.849**	0.8323	0.8428	0.8413	0.84	0.8464

Table 5.5: Test accuracy of the final models for full retraining (Full column), online training (Online) and proactive training with trigger size 8 without sparsity ($Proact_8$ column) and with different levels of sparsification using the top_k sparse selector (Top_k x% columns with $x = 1, 0.1, 0.01$). Highest accuracy achieved for each model is shown in bold.

5.4.4 Discussion

The CIFAR 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 top_k 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 up to 10,000x times the changed parameters per iteration; thus enabling the continuous communication of the gradients to a deployed model.

At the beginning of the prequential accuracy curves for both online and proactive training, we see that the trained models are performing worse than the initial. The problem could be either attributed to the warm-started values of the ADAM optimizer or to the fact that it needs a warmup stage. It can be fixed by not warm-starting and/or allowing for a warmup period before deploying updated models in place of the initial model.

Let us now analyse how much using the sparse selector top_k 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 any the top_k selector is 4000s for all the sparsification ratios (it is actually 3870.3s, 4032s and 4154s for sparsification 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 $\frac{training\_time}{\#\ iterations} \simeq \frac{4{,}000s}{5{,}000} = 800ms$. Let's compare the bandwidth consumed when sending parameters equal to number selected by top_k and its non-sparse variant, considering each parameter is a 32bit float:

- The non-sparse proactive training needs to send all the parameters every 8ms, consuming a bandwidth of $\frac{26{,}494{,}090*32\text{bits}}{800ms} = 132.47$ MBps. The non-sparse proactive training actually needs to do this in less time as it is faster than the top_k variant.

- The top_k 1.0% needs to send 618 parameters every 8ms, and also the indices of the parameters that we will assume are stored in 32bit integers. This consumes a bandwidth of $\frac{2*264{,}641*32\text{bits}}{8ms} \simeq 2.65$ MBps.

- The top_k 0.1% needs to send 67 parameters plus the indices of the changes every 8ms ,consuming a bandwidth of $\frac{2*26{,}498*32\text{bits}}{800ms} = 264.98$ kBps.

- The top_k 0.01% needs to send 13 parameters plus the indices of the changes every 8ms , consuming a bandwidth of $\frac{2*2{,}886*32\text{bits}}{800ms} = 28.86$ kBps.

We see that for a network like densenet, the bandwidth needed for naive continuous deployment is at 132.47 MBps, which would be hard to reliably keep in the real-world. It would also consume much of the bandwidth needed to receive 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 not-so-fast

GPU (Nvidia K40) by modern standards. According to the techpowerup website [1], 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.

A counter-intuitive finding is that at times sparse training has achieved better prequential performance than its non-sparse variant. We suspect that large models are highly redundant and the top_k operator manages to select the less redundant changes per iteration, ignoring the non-important ones. In that way, it manages to provide a "regularizing effect" in the sparse training process. Sparse training is not all about advantages; there is a price to pay when using it. More specifically, the top_k sparse selection incurs a non-negligible overhead that grows the proactive training time by 2x and 3x times for the larger models that we experimented with. We suspect that this is a small price to pay for a solution to continuous deployment of large DL models. Also, this cost can probably be improved by using data structures that make the top_k operator more performant. Still, even with this overhead the training times of sparse proactive training remain well below the corresponding total training time of full retraining.

We should note here, that our reported full retraining times are wildly underestimated due to the fact that we do training over only 25 epochs, instead of a few hundreds of epochs that the used models need to achieve SOTA performance. For example Densenets need 300 epochs to achieve SOTA performance on CIFAR10 [39]. Meanwhile, one can say that also the prequential evaluation performance of the full retraining is underestimated. 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 (see Section 5.2.2 for our simple choice of the hyperparameters).

5.5 Conclusion

In this chapter, we have presented our experiments with LeNet-5 on the MNIST dataset and with various SOTA models on the CIFAR dataset. We compared proactive training with online training and full retraining, simulating a scenario where data arrive in a streaming fashion. For proactive training, we explored different sparsification strategies and ratios to allow continuous deployment

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

at every iteration.

Concerning prequential evaluation, we find proactive training and full retraining to achieve better performance than online training at all the experiments. Compared to full retraining, proactive training achieves comparable or superior performance provided that we tune the number of iterations it will perform by controlling the trigger size. With respect to the training time, proactive training needs a fraction of the time of full retraining to achieve comparable or superior performance. When compared to online training, proactive training needs to train for exactly $\frac{batch\ size}{trigger\ size}$ more iterations, needing approximately this much more time.

Experiments on CIFAR show that the top_k sparsification used for continuous deployment can surprisingly improve the performance even more. It is as though the magnitude of the gradients is a good proxy for importance, a fact which brings a regularizing effect that can prevent or at least slow down overfitting to the historical dataset. Using the top_k sparse selector will bring an overhead to the training process that is more evident when training large models. With Resnet-18, Resnet-50 and Densenet-161 the training time increased by a factor close to 1.7, 3 and 2 respectively. This is expected as the top_k operator has a complexity that depends on the number of the parameters of a model. Nevertheless, sparse training has shown to achieve comparable - if not superior due to its regularizing effect - performance to normal training using arbitrarily large sparsification ratios up to 0.01%. The increase in training time can become negligible when we think that the communication cost per iteration for continuous deployment can drop 10,000x.

In short, proactive training keeps DL models fresh as soon as new data are available like online training and leads to model quality that is comparable - if not superior - to full retraining combining the advantages of both worlds. Although not directly measured in our experiments, it can overfit to historical data if triggered too often, because of the bias in the sampling process towards historical data. Sparse training can effectively reduce the size of communicated gradients by 10,000x and provides an unexpected regularizing effect that shows to reduce the overfitting to historical data. More importantly, it enables continuous deployment of very large models and opens a new road for continuous DL model training and deployment in streaming settings.

Concluding this chapter, we make a summary of our experimental findings in Section 5.5.1 and identify shortcomings of our experiments in Section 5.5.2.

5.5.1 Findings

Here, we explicitly identify some of our main findings regarding the use of sparse proactive training for continuous training and deployment of DL models.

Beginning of the Continuous Training

At the beginning of the prequential accuracy curves for both online and proactive training, we see that the trained models are performing worse than the initial. This behavior is more evident in the CIFAR experiments. The problem could be related to the warm-started values of the ADAM optimizer. It can be fixed by not warm-starting and/or allowing for a warmup period before deploying updated models rather than the initial model.

Proactive Training Trigger Size

Undeniably, the trigger size is one very important hyperparameter of proactive training. In Chapter 4 we came up with the following simple rule:

- 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 online training.

With respect to the training time, proactive training needs exactly $\frac{batch\ size}{trigger\ size}$ times more iterations and approximately the same amount of times more time to train. Preprocessing and early materialization of historical data samples [23] can provide optimizations that reduce this time.

Things are more complex when it comes to the effect of the trigger size on model quality. Theoretically, we expect more (or less) iterations induced by a smaller (or larger) trigger size to give a means of controlling the bias-variance tradeoff of the model. In the MNIST experiments (Section 5.3), we saw that as the trigger size is smaller and more training iterations are triggered, the prequential evaluation performance gets better. In the CIFAR experiments (Section 5.4), we saw that in certain cases higher trigger sizes were also good to obtain good performance. In general, we suspect that this may change according to the data and model properties and needs appropriate tuning. We also imagine enhancements of the approach that define trigger conditions that do not just depend on the size of the new examples, but incorporate domain knowledge or convergence criteria. For example, one could choose to trigger a proactive training iteration periodically as suggested initially by Derakhshan et al. [23] or when the loss on new data is above a certain threshold.

Sparse Selector and Sparsification Ratio

Already in the MNIST experiments we found more promise in the top_k sparse selector, rather than $random_k$ which slower not only in training but also in keeping a model fresh. For the top_k selector, we found that it is able to keep the prequential accuracy at levels comparable to non-sparse training even at extreme sparsification, using 10000x less gradients per iteration. At the CIFAR experiments greater sparsification (using less gradients), led to even better accuracy than the non-sparse variant. We speculate that this is due to the use of only a few ”important” gradients per iteration, producing a regularizing effect, which in our experiments can prevent from overfitting to historical data. We do not of course see this regularizing effect as a panacea to overfitting, but suspect that it can help alleviate part of the problem arisen from the sample bias of proactive training (see Section 4.1.2).

5.5.2 Shortcomings

We identify the following shortcomings of our experiments:

- Both the MNIST dataset and the CIFAR dataset are static and exhibit a relatively similar distribution of classes between the different batches. Our methods for continuous training and deployment are much more relevant with streaming data. We expect them to be more performant in the cases where some drift exists where their capacity to adapt can shine.
- Our experiments only use convolutional DL models for computer vision. To provide generalized results, it is important to try our continuous training and deployment approach for different flavors of DL models suchs as GANs [28], LSTMs [37] and Autoencoders applied in a variety of datasets covering multiple domains.
- We do not use any hyper-parameter tuning, learning rate scheduling or training for many epochs that is needed to achieve state-of-the-art performance. As an example the SOTA

accuracy of Densenet in CIFAR10 test set is 94.81 after training for 300 epochs [39], while our best Densenet model achieves 84.9 accuracy on the test set having trained for only 25 epochs. Our methods should be enhanced with continuous hyper-parameter tuning and learning rate scheduling strategies and compared with state-of-the-art training configurations.

- After the MNIST experiments (5.3), we rejected the $random_k$ sparse selector for the deployment case because it was generally slower than top_k and very unstable at the beginning of the training process.

  We identify two reasons why our argumentation may be flawed in that respect:

  - The efficiency of $random_k$ can be improved and be faster than top_k. The top_k operator has a complexity that depends on the number of the model's parameters, while good implementation of the $random_k$ can have a complexity that depends solely on the sample size and not the number of the model's parameters. So, the fact that $random_k$ was slower in our experiments is due to the lack of a fast random sampling function for tensors in the PyTorch framework. Implementing an efficient sampler for PyTorch tensors was out of the scope of this study, since the top_k operator was performing good enough to continue our experiments.
  - With the CIFAR experiments we confirmed that there is an instability during the beginning of the training that may be related to the warm-starting of the ADAM optimizer. It could be that this same instability was only more evident for the $random_k$ selector in the MNIST experiments. When this instability is fixed, it is time to revisit the $random_k$ selector once more.

- In all of our experiments we used the ADAM optimizer. In hindsight, it may be a crucial part of sparse training to use an optimizer that adapts the learning rate of each individual parameter like ADAM [47] or Adagrad [25], rather than vanilla SGD.

Later, talking about future work (Section 6.1), we discuss directions to follow in order to address these shortcomings.

6 Conclusion

In this work, we have identified and proposed to solve a mismatch in the DL model lifecycle. More specifically, while in several applications training data arrive continuously in a streaming fashion, the DL models that are used to make predictions on them are (re)trained and (re)deployed periodically. The most prominent method of retraining is to trigger a new training from scratch once the model performance degrades or there is enough new data available. This accounts for a highly compute-demanding process that results in stale models in production. Instead, our study proposes a framework for continuous training and deployment of DL models.

For continuous training, we leverage the proactive training method suggested by Derakhshan et al. [23] and initially evaluated on models with convex loss functions (SVM, Logistic Regression). Instead, we use it to train DL models and introduce the notion of the trigger size, that is the number of new elements that will trigger a proactive training iteration. After an initial training, proactive training continues the training process with mini-batch SGD iterations, where every mini-batch is formed using a combination of newly arrived data with a sample of historical data. After being used for training, new data become part of the historical dataset. Our experiments with LeNet-5 on MNIST and with various SOTA DL models on CIFAR10 show that proactive training can provide comparable, if not superior performance to full retraining at a fraction of the cost. Simulating the arrival of data in a streaming fashion, we see that proactive training is very effective at keeping models fresh.

For continuous deployment, we identify that our major concern is to reduce the communication of the model changes. We borrow an idea that comes from the domain of distributed DL training where the communication reduction is met with the enforced sparsification of the gradients. We enhance the proactive training approach of the previous paragraph and end up with sparse proactive training. It uses a sparse selector to reduce the gradients used for model changes per iteration, while keeping a memory that accumulates unused gradients. Sparse proactive training with the selector top_k that only selects to apply the gradients with the largest magnitude achieves comparable - and at times superior - performance to its non-sparse variant reducing the communication per iteration up to 10,000x. This comes at a price since the top_k operator incurs an overhead that depends on the size of the DL model it is applied to. For Densenet161 and Resnet50, the largest models we experimented with, the total training time is increased by a factor of 2 and 3, respectively. Nevertheless, we expect that this is a small price to pay for enabling the continuous deployment of even very large models, opening the road for new DL applications.

In short, this study brings a solution to the problem of continuous training and deployment of DL models, that is validated by our experiments. To conclude, we present directions for future work in Section 6.1.

6.1 Future Work

This thesis identifies multiple areas of future work:

- **Real-world deployment**. In this work, we have examined the cost of sparse training, its convergence rate and the resulting model quality. We have not sent any updates to a remotely deployed model. Future studies should deal with problems that arise in real-world networks like lost or delayed updates due to unstable connection.

- **Distributed continuous training**. Our methods have been proposed for centralized training, but with the modern model and dataset sizes we find it important that they are extended and tested for distributed training.

- **Streaming datasets, multiple DL models**. Both the MNIST dataset and the CIFAR dataset that we used in our experiments are static and exhibit a relatively similar distribution of classes between the different batches. Our methods for continuous training and deployment are much more relevant with streaming data. Meanwhile we only used convolutional DL models for computer vision. To provide generalized results, it is important to try our continuous training and deployment approach for different flavors of DL models suchs as GANs, LSTMs and Autoencoders applied in a variety of datasets covering multiple domains.

- **SOTA performance**. Our experiments were not meant to achieve state-of-the-art performance. As an example the SOTA accuracy of Densenet in CIFAR10 test set is 94.81 after training for 300 epochs [39], while our best Densenet model achieves 84.9 accuracy on the test set having trained for only 25 epochs. To achieve SOTA performance one should do careful hyper-parameter tuning, learning rate scheduling and training for many epochs. In a future work, our methods continuous training and deployment can be enhanced with those features and compared with state-of-the-art training configurations.

- **Unstable beginning of training**. In the CIFAR experiments (Section 5.4) we identified that the beginning of the continuous training is unstable and gives for some time models worse than the initial. We suspect that this is due to the ADAM optimizer not being adapted yet. We suppose it is needed to allow for a warmup period before updating a deployed model. It is important to address this issue in a future study. The $random_k$ sparse selector should also be revisited when addressing this issue as one of its main problems compared to top_k was that it was unstable at the beginning of the training process.

- **Adaptive learning rate for sparse training**. Our experiments only used the ADAM [47] optimizer. In retrospect, for sparse training it may be very important to use such an optimizer that adapts the learning rate of each individual parameter like ADAM [47] or Adagrad [25], rather than vanilla SGD. Given the knowledge of which parameter is being changed at each iteration, the learning rate of each individual parameter can probably be adapted better than what ADAM does. This can be guided by the fact that not using some parameters for some iterations is equivalent to training them with larger batch sizes [58]. Work on training with larger batches [29] has been successfully enabled by proper adjustments of the learning rate and we suppose a similar approach can be applied for sparse training. Future work will compare adaptive and non-adaptive strategies and propose new methods for adapting the learning rate of each individual parameter for sparse training.

List of Acronyms

DL	Deep Learning
ML	Machine Learning
DNN	Deep Neural Network
LSTM	Long Short Term Memory
IoT	Internet of Things
SVM	Support Vector Machine
GPU	Graphics Processing Unit
CPU	Central Processing Unit
GAN	Generative Adversarial Network
SOTA	State Of The Art

Bibliography

[1] Abadi, M., Barham, P., Chen, J., Chen, Z., Davis, A., Dean, J., Devin, M., Ghemawat, S., Irving, G., Isard, M., 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.

[2] Aji, A. F., and Heafield, K. Sparse communication for distributed gradient descent. *arXiv preprint arXiv:1704.05021* (2017).

[3] Alistarh, D., Grubic, D., Li, J., Tomioka, R., and Vojnovic, M. Qsgd: Communication-efficient sgd via gradient quantization and encoding. In *Advances in Neural Information Processing Systems* (2017), pp. 1709–1720.

[4] Amatriain, X., and Basilico, J. Netflix recommendations: beyond the 5 stars (part 1). *Netflix Tech Blog 6* (2012).

[5] Amodei, D., and Hernandez, D. Ai and compute. *Heruntergeladen von https://blog. openai. com/aiand-compute* (2018).

[6] Ash, J. T., and Adams, R. P. On the difficulty of warm-starting neural network training. *arXiv preprint arXiv:1910.08475* (2019).

[7] Bai, J., Lu, F., Zhang, K., et al. Onnx: Open neural network exchange. `https://github.com/onnx/onnx`, 2019.

[8] Baylor, D., Breck, E., Cheng, H.-T., Fiedel, N., Foo, C. Y., Haque, Z., Haykal, S., Ispir, M., Jain, V., Koc, L., et al. Tfx: A tensorflow-based production-scale machine learning platform. In *Proceedings of the 23rd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining* (2017), pp. 1387–1395.

[9] Baylor, D., Haas, K., Katsiapis, K., Leong, S., Liu, R., Menwald, C., Miao, H., Polyzotis, N., Trott, M., and Zinkevich, M. Continuous training for production {ML} in the tensorflow extended ({TFX}) platform. In *2019 {USENIX} Conference on Operational Machine Learning (OpML 19)* (2019), pp. 51–53.

[10] Bergstra, J., Breuleux, O., Bastien, F., Lamblin, P., Pascanu, R., Desjardins, G., Turian, J., Warde-Farley, D., and Bengio, Y. Theano: a cpu and gpu math expression compiler. In *Proceedings of the Python for scientific computing conference (SciPy)* (2010), vol. 4, Austin, TX, pp. 1–7.

[11] Berner, C., Brockman, G., Chan, B., Cheung, V., Debiak, P., Dennison, C., Farhi, D., Fischer, Q., Hashme, S., Hesse, C., et al. Dota 2 with large scale deep reinforcement learning. *arXiv preprint arXiv:1912.06680* (2019).

[12] Bottou, L. Large-scale machine learning with stochastic gradient descent. In *Proceedings of COMPSTAT'2010*. Springer, 2010, pp. 177–186.

[13] Bottou, L. Stochastic gradient descent tricks. In *Neural networks: Tricks of the trade*. Springer, 2012, pp. 421–436.

[14] Brown, T. B., Mann, B., Ryder, N., Subbiah, M., Kaplan, J., Dhariwal, P., Neelakantan, A., Shyam, P., Sastry, G., Askell, A., et al. Language models are few-shot learners. *arXiv preprint arXiv:2005.14165* (2020).

[15] Buciluă, C., Caruana, R., and Niculescu-Mizil, A. Model compression. In *Proceedings of the 12th ACM SIGKDD international conference on Knowledge discovery and data mining* (2006), pp. 535–541.

[16] Chen, T., Li, M., Li, Y., Lin, M., Wang, N., Wang, M., Xiao, T., Xu, B., Zhang, C., and Zhang, Z. Mxnet: A flexible and efficient machine learning library for heterogeneous distributed systems. *arXiv preprint arXiv:1512.01274* (2015).

[17] Cho, K., Van Merriënboer, B., Gulcehre, C., Bahdanau, D., Bougares, F., Schwenk, H., and Bengio, Y. Learning phrase representations using rnn encoder-decoder for statistical machine translation. *arXiv preprint arXiv:1406.1078* (2014).

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

[19] d2l.ai course. Convolutional neural networks (lenet). `https://d2l.ai/chapter_convolutional-neural-networks/lenet.html`. [Online, accessed on 08.08.2020].

[20] Dawid, A. P. Present position and potential developments: Some personal views statistical theory the prequential approach. *Journal of the Royal Statistical Society: Series A (General) 147*, 2 (1984), 278–290.

[21] Dean, J. 1.1 the deep learning revolution and its implications for computer architecture and chip design. In *2020 IEEE International Solid-State Circuits Conference-(ISSCC)* (2020), IEEE, pp. 8–14.

[22] Deng, J., Dong, W., Socher, R., Li, L.-J., Li, K., and Fei-Fei, L. Imagenet: A large-scale hierarchical image database. In *2009 IEEE conference on computer vision and pattern recognition* (2009), Ieee, pp. 248–255.

[23] Derakhshan, B., Mahdiraji, A. R., Rabl, T., and Markl, V. Continuous deployment of machine learning pipelines. In *EDBT* (2019), pp. 397–408.

[24] Devlin, J., Chang, M.-W., Lee, K., and Toutanova, K. Bert: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805* (2018).

[25] Duchi, J., Hazan, E., and Singer, Y. Adaptive subgradient methods for online learning and stochastic optimization. *Journal of machine learning research 12*, 7 (2011).

[26] Glorot, X., and Bengio, Y. Understanding the difficulty of training deep feedforward neural networks. In *Proceedings of the thirteenth international conference on artificial intelligence and statistics* (2010), pp. 249–256.

[27] Goodfellow, I., Bengio, Y., and Courville, A. *Deep learning.* MIT press, 2016.

[28] Goodfellow, I., Pouget-Abadie, J., Mirza, M., Xu, B., Warde-Farley, D., Ozair, S., Courville, A., and Bengio, Y. Generative adversarial nets. In *Advances in neural information processing systems* (2014), pp. 2672–2680.

[29] Goyal, P., Dollár, P., Girshick, R., Noordhuis, P., Wesolowski, L., Kyrola, A., Tulloch, A., Jia, Y., and He, K. Accurate, large minibatch sgd: Training imagenet in 1 hour. *arXiv preprint arXiv:1706.02677* (2017).

[30] Grigorescu, S., Trasnea, B., Cocias, T., and Macesanu, G. A survey of deep learning techniques for autonomous driving. *Journal of Field Robotics 37*, 3 (2020), 362–386.

[31] Han, S., Mao, H., and Dally, W. J. Deep compression: Compressing deep neural networks with pruning, trained quantization and huffman coding. *arXiv preprint arXiv:1510.00149* (2015).

[32] Hannun, A., Case, C., Casper, J., Catanzaro, B., Diamos, G., Elsen, E., Prenger, R., Satheesh, S., Sengupta, S., Coates, A., et al. Deep speech: Scaling up end-to-end speech recognition. *arXiv preprint arXiv:1412.5567* (2014).

[33] Hayes, T. L., Cahill, N. D., and Kanan, C. Memory efficient experience replay for streaming learning. In *2019 International Conference on Robotics and Automation (ICRA)* (2019), IEEE, pp. 9769–9776.

[34] He, K., Zhang, X., Ren, S., and Sun, J. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition* (2016), pp. 770–778.

[35] Hinton, G., Deng, L., Yu, D., Dahl, G. E., Mohamed, A.-r., Jaitly, N., Senior, A., Vanhoucke, V., Nguyen, P., Sainath, T. N., et al. Deep neural networks for acoustic modeling in speech recognition: The shared views of four research groups. *IEEE Signal processing magazine 29*, 6 (2012), 82–97.

[36] Hinton, G., Vinyals, O., and Dean, J. Distilling the knowledge in a neural network. *arXiv preprint arXiv:1503.02531* (2015).

[37] Hochreiter, S., and Schmidhuber, J. Long short-term memory. *Neural computation 9*, 8 (1997), 1735–1780.

[38] Howard, A. G., Zhu, M., Chen, B., Kalenichenko, D., Wang, W., Weyand, T., Andreetto, M., and Adam, H. Mobilenets: Efficient convolutional neural networks for mobile vision applications. *arXiv preprint arXiv:1704.04861* (2017).

[39] Huang, G., Liu, Z., Weinberger, K. Q., and van der Maaten, L. Densely connected convolutional networks. arxiv 2016. *arXiv preprint arXiv:1608.06993 1608* (2018).

[40] Jaini, P., Rashwan, A., Zhao, H., Liu, Y., Banijamali, E., Chen, Z., and Poupart, P. Online algorithms for sum-product networks with continuous variables. In *Conference on Probabilistic Graphical Models* (2016), pp. 228–239.

[41] Jia, Y., Shelhamer, E., Donahue, J., Karayev, S., Long, J., Girshick, R., Guadarrama, S., and Darrell, T. Caffe: Convolutional architecture for fast feature embedding. *arXiv preprint arXiv:1408.5093* (2014).

[42] Jouppi, N. P., Young, C., Patil, N., Patterson, D., Agrawal, G., Bajwa, R., Bates, S., Bhatia, S., Boden, N., Borchers, A., et al. In-datacenter performance analysis of a tensor processing unit. In *Proceedings of the 44th Annual International Symposium on Computer Architecture* (2017), pp. 1–12.

[43] Karpathy, A. What i learned from competing against a convnet on imagenet. *GitHub, Sep 2* (2014).

[44] Katsiapis, K., and Haas, K. Towards ml engineering with tensorflow extended (tfx). In *Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining* (2019), pp. 3182–3182.

[45] Kelley, H. J. Gradient theory of optimal flight paths. *Ars Journal 30*, 10 (1960), 947–954.

[46] Keskar, N. S., and Socher, R. Improving generalization performance by switching from adam to sgd. *arXiv preprint arXiv:1712.07628* (2017).

[47] Kingma, D. P., and Ba, J. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980* (2014).

[48] Klinger, J., Mateos-Garcia, J. C., and Stathoulopoulos, K. Deep learning, deep change? mapping the development of the artificial intelligence general purpose technology. *Mapping the Development of the Artificial Intelligence General Purpose Technology (August 17, 2018)* (2018).

[49] Koloskova, A., Stich, S. U., and Jaggi, M. Decentralized stochastic optimization and gossip algorithms with compressed communication. *arXiv preprint arXiv:1902.00340* (2019).

[50] Krizhevsky, A., Hinton, G., et al. Learning multiple layers of features from tiny images.

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

[52] kuangliu. Train cifar10 with pytorch. `https://github.com/kuangliu/pytorch-cifar`. [Online, accessed on 01.05.2020].

[53] LeCun, Y. The mnist database of handwritten digits. *http://yann. lecun. com/exdb/mnist/* (1998).

[54] LeCun, Y. Who is afraid of non-convex loss functions. In *2007 NIPS workshop on Efficient Learning, Vancouver, December* (2007), vol. 7, Citeseer.

[55] LeCun, Y., Bengio, Y., et al. Convolutional networks for images, speech, and time series. *The handbook of brain theory and neural networks 3361*, 10 (1995), 1995.

[56] LeCun, Y., Bengio, Y., and Hinton, G. Deep learning. *nature 521*, 7553 (2015), 436–444.

[57] LeCun, Y., et al. Lenet-5, convolutional neural networks. *URL: http://yann. lecun. com/exdb/lenet 20*, 5 (2015), 14.

[58] Lin, Y., Han, S., Mao, H., Wang, Y., and Dally, W. J. Deep gradient compression: Reducing the communication bandwidth for distributed training. *arXiv preprint arXiv:1712.01887* (2017).

[59] Lomonaco, V. *Continual Learning with Deep Architectures.* PhD thesis, alma, 2019.

[60] Lomonaco, V., Maltoni, D., and Pellegrini, L. Fine-grained continual learning. *arXiv preprint arXiv:1907.03799 1* (2019).

[61] Marcel, S., and Rodriguez, Y. Torchvision the machine-vision package of torch. In *Proceedings of the 18th ACM international conference on Multimedia* (2010), pp. 1485–1488.

[62] McCloskey, M., and Cohen, N. J. Catastrophic interference in connectionist networks: The sequential learning problem. In *Psychology of learning and motivation*, vol. 24. Elsevier, 1989, pp. 109–165.

[63] Mnih, V., Kavukcuoglu, K., Silver, D., Graves, A., Antonoglou, I., Wierstra, D., and Riedmiller, M. Playing atari with deep reinforcement learning. *arXiv preprint arXiv:1312.5602* (2013).

[64] Mohammadi, M., Al-Fuqaha, A., Sorour, S., and Guizani, M. Deep learning for iot big data and streaming analytics: A survey. *IEEE Communications Surveys & Tutorials 20*, 4 (2018), 2923–2960.

[65] Ng, Andrew. Deep learning specialization. Coursera: `https://www.coursera.org/specializations/deep-learning`, 2018. [Online, accessed on 08.08.2020].

[66] Olston, C., Fiedel, N., Gorovoy, K., Harmsen, J., Lao, L., Li, F., Rajashekhar, V., Ramesh, S., and Soyke, J. Tensorflow-serving: Flexible, high-performance ml serving. *arXiv preprint arXiv:1712.06139* (2017).

[67] Parisi, G. I., and Lomonaco, V. Online continual learning on sequences. In *Recent Trends in Learning From Data*. Springer, 2020, pp. 197–221.

[68] Paszke, A., Gross, S., Massa, F., Lerer, A., Bradbury, J., Chanan, G., Killeen, T., Lin, Z., Gimelshein, N., Antiga, L., et al. Pytorch: An imperative style, high-performance deep learning library. In *Advances in neural information processing systems* (2019), pp. 8026–8037.

[69] Pomponi, J., Scardapane, S., Lomonaco, V., and Uncini, A. Efficient continual learning in neural networks with embedding regularization. *Neurocomputing* (2020).

[70] Recht, B., Re, C., Wright, S., and Niu, F. Hogwild: A lock-free approach to parallelizing stochastic gradient descent. In *Advances in neural information processing systems* (2011), pp. 693–701.

[71] Samek, W., Montavon, G., Lapuschkin, S., Anders, C. J., and Müller, K.-R. Toward interpretable machine learning: Transparent deep neural networks and beyond. *arXiv preprint arXiv:2003.07631* (2020).

[72] Sandler, M., Howard, A., Zhu, M., Zhmoginov, A., and Chen, L.-C. Mobilenetv2: Inverted residuals and linear bottlenecks. In *Proceedings of the IEEE conference on computer vision and pattern recognition* (2018), pp. 4510–4520.

[73] Seide, F., Fu, H., Droppo, J., Li, G., and Yu, D. 1-bit stochastic gradient descent and its application to data-parallel distributed training of speech dnns. In *Fifteenth Annual Conference of the International Speech Communication Association* (2014).

[74] Shi, S., Wang, Q., Xu, P., and Chu, X. Benchmarking state-of-the-art deep learning software tools. In *2016 7th International Conference on Cloud Computing and Big Data (CCBD)* (2016), IEEE, pp. 99–104.

[75] Silver, D., Hubert, T., Schrittwieser, J., Antonoglou, I., Lai, M., Guez, A., Lanctot, M., Sifre, L., Kumaran, D., Graepel, T., et al. Mastering chess and shogi by self-play with a general reinforcement learning algorithm. *arXiv preprint arXiv:1712.01815* (2017).

[76] Stich, S. U., Cordonnier, J.-B., and Jaggi, M. Sparsified sgd with memory. In *Advances in Neural Information Processing Systems* (2018), pp. 4447–4458.

[77] Vanholder, H. Efficient inference with tensorrt, 2016.

[78] Vinyals, O., Babuschkin, I., Czarnecki, W. M., Mathieu, M., Dudzik, A., Chung, J., Choi, D. H., Powell, R., Ewalds, T., Georgiev, P., et al. Grandmaster level in starcraft ii using multi-agent reinforcement learning. *Nature 575*, 7782 (2019), 350–354.

[79] Wikipedia contributors. Artificial neural network. `https://en.wikipedia.org/wiki/Artificial_neural_network`. [Online, accessed on 08.08.2020].

[80] Wikipedia contributors. Mnist database. `https://en.wikipedia.org/wiki/MNIST_database`. [Online, accessed on 08.08.2020].

[81] Wilson, A. C., Roelofs, R., Stern, M., Srebro, N., and Recht, B. The marginal value of adaptive gradient methods in machine learning. In *Advances in neural information processing systems* (2017), pp. 4148–4158.

[82] Wu, P., Hoi, S. C., Xia, H., Zhao, P., Wang, D., and Miao, C. Online multimodal deep similarity learning with application to image retrieval. In *Proceedings of the 21st ACM international conference on Multimedia* (2013), pp. 153–162.

[83] Wu, Y., Schuster, M., Chen, Z., Le, Q. V., Norouzi, M., Macherey, W., Krikun, M., Cao, Y., Gao, Q., Macherey, K., et al. Google's neural machine translation system: Bridging the gap between human and machine translation. *arXiv preprint arXiv:1609.08144* (2016).