PARTIME: Scalable and Parallel Processing Over Time with Deep Neural Networks

Enrico Meloni∗†‡, Lapo Faggi∗‡, Simone Marullo∗†, Alessandro Bett†, Matteo Tiezzi∗, Marco Gori∗†, and Stefano Melacci∗

meloni@diism.unisi.it, {lapo.faggi, simone.marullo}@unifi.it, alessandro.betti@inria.fr, mttiezzi@diism.unisi.it, marco.gori@unisi.it, mela@diism.unisi.it

∗University of Siena, Italy
†University of Florence, Italy
‡Université Côte d’Azur, France

Abstract—In this paper, we present PARTIME, a software library written in Python and based on PyTorch, designed specifically to speed up neural networks whenever data is continuously streamed over time, for both learning and inference. Existing libraries are designed to exploit data-level parallelism, assuming that samples are batched, a condition that is not naturally met in applications that are based on streamed data. Differently, PARTIME starts processing each data sample at the time in which it becomes available from the stream. PARTIME wraps the code that implements a feed-forward multi-layer network and it distributes the layer-wise processing among multiple devices, such as Graphics Processing Units (GPUs). Thanks to its pipeline-based computational scheme, PARTIME allows the devices to perform computations in parallel. At inference time this results in scaling capabilities that are theoretically linear with respect to the number of devices. During the learning stage, PARTIME can leverage the non-i.i.d. nature of the streamed data with samples that are smoothly evolving over time for efficient gradient computations. Experiments are performed in order to empirically compare PARTIME with classic non-parallel neural computations in online learning, distributing operations on up to 8 NVIDIA GPUs, showing significant speedups that are almost linear in the number of devices, mitigating the impact of the data transfer overhead.

Index Terms—Pipeline parallelism, PARTIME, Continual Learning

I. INTRODUCTION

In the last few years, there has been a growing attention towards learning problems framed as continual or lifelong [1]. Even if many recent approaches exists in this direction, this setting remains extremely challenging. Applications well-suited for continual learning have access to a continuous stream of data, where an artificial agent is not only expected to use the data to make predictions, but also to adapt to changes in the environment, i.e. videos, stream of texts, etc. [2], [3]. In the case of neural nets, the most challenging context is one in which a simple online update of weights is applied at each time instant, given the information from the last received sample [4]. Despite having access to powerful computational resources for continual learning-based applications, current algorithmic solutions have not been paired with the development of software libraries designed to speed-up computations. In fact, storing and processing portions of the streamed data in a batch-like fashion is the most common approach, reusing classic non-continual learning tools. However, the artificial nature of this approach is striking. Motivated by the intuitions behind existing libraries for batched data [5] and by approaches that rethink the neural network computational scheme making it local in time and along the network architecture [6]–[9], we propose a different approach to pipeline parallelism specifically built for data sequentially streamed over time, where multiple devices work in parallel to speed-up computations. Considering D independent devices, such as D GPUs, the computational time of a feed-forward deep network empowered by our approach theoretically reduces by a factor 1/D. We experimentally show that the existing overheads due to data transfer among different devices are constant with respect to D in certain hardware configurations. On the reverse side, the higher throughput obtained by a pipeline parallelism are associated with a delay between the forward wave and the backward wave while they propagate through the network proportional to D, a feature that is not critical in applications in which data samples are non-i.i.d. and smoothly evolve over time.

The contributions of this paper are the following. (1) We describe and share a software library named PARTIME (PArallel processing over TIME)†, designed for efficient continual online learning in multi-GPU architectures using pipeline parallelism. Feed-forward neural architecture can be easily embraced by the library, distributing them among GPUs (Section III). (2) We leverage CUDA Streams‡ paired with CUDA

†https://github.com/sailab-code/partime
‡https://pytorch.org/docs/stable/generated/torch.cuda.Stream.html
Graphs\textsuperscript{3} to enable fast kernel scheduling, with a low-level abstraction of the parallel routines. We also provide automatic load balancing tools, suitable for the context (Section III, Section IV). (3) We experimentally evaluate the wall-clock running times with different numbers of devices, investigating the impact of the delays and of the data transfer overhead on several architectures. In some configurations we get empirical speedups close to the theoretical ones (Section V). (4) We evaluate the quality of learning with different pipeline configurations in continual learning-inspired settings (Section V).

II. RELATED WORK

The intuition of exploiting data- and/or model-level parallelism to speedup and scale training of deep networks\textsuperscript{[10], [11]} has been largely followed in the last decade\textsuperscript{[12], [13]}, also focusing on specific tasks or classes of neural nets\textsuperscript{[14]}. These approaches were motivated by the large memory requirements of recent neural models. Naively splitting the models into components distributed across different devices incurs in under-utilization of devices caused by device synchronization. Pipeline parallelism deals with this under-utilization issue, not only splitting the network across multiple devices but also splitting data into micro-batches and scheduling their forward propagation through the various stages, such that each device is left idle the least amount of time\textsuperscript{[15]}. Such computational paradigm yields an improvement on the number of inputs processed per seconds, based on the batch size, the number of stages and the communication overheads between devices. In fact, existing software assume mini-batch processing to accelerate learning on big datasets and SGD. In the following, we review popular pipeline parallelism-based libraries.

GPipe\textsuperscript{[5]} parallelizes the forward and backward phases across the micro-batches, with the latter being followed by a pipeline flush, where the accumulated gradients are used to update the weights. This induces a bubble of idle time for most GPUs, reducing the throughput. On the other hand, it replicates the gradients of a sequential processing by stashing the activation values of each stage for all micro-batches, thus incurring in a significant memory overhead.

PipeDream\textsuperscript{[16]} does not keep forward and backward stages completely separated, and they are in fact interleaved continuously, without leaving any GPU idle. Weights are updated at each step so that, in order to compute the correct gradients for any batch, PipeDream adopts weights stashing in addition to activation stashing. PipeDream allows for a higher throughput with respect to GPipe, but incurs in a heavy memory overhead not suitable for larger models. PipeDream-2BW\textsuperscript{[17]} improves PipeDream design by storing only two versions of the weights in each stage. When weights are updated, they are not used immediately to avoid inconsistencies in processing the data that already entered the pipeline, introducing a delay in the gradient computation, thus reducing the memory overhead while achieving a high speed-up at the cost of a small approximation of the gradients.

In the case of PARTIME, differently from other approaches we adapt the pipeline parallelism paradigm to the continual online learning scenario, where the input is provided in a sequential manner and there are no offline batches of data. The computation paradigm is related to that of PipeDream, forward and backward phases are interleaved continuously without any flush, allowing a theoretical speed-up of the processing frame-rate of D. Differently from PipeDream(-2BW), PARTIME does not stash any activation or weights, further reducing the memory overhead. Differently from GPipe and both PipeDream approaches, weights are updated at the end of the backward computation of each device, allowing continuous learning of the agent without any hiccup in the framerate, at the cost of an approximation of the gradients, that is not critical when the inputs slowly change over time. We depict in Fig. 1 the PARTIME approach.

The theoretical groundings behind the approach implemented in PARTIME can be traced back to early studies on how to distribute the sequential computations of the network layers in a pipeline-like scheme\textsuperscript{[18]}. More recent activity\textsuperscript{[6], [8]} described how such a computational pipeline could be used to train networks where layers are independent modules interconnected by a special class of constraints that can learn in parallel, such as that described in\textsuperscript{[8]} and in\textsuperscript{[9]}, thus we note that PARTIME could naturally implement alternatives to Backpropagation, even if this is beyond the scope of this work.

III. SCALABLE AND PARALLEL LOCAL COMPUTATIONS OVER TIME

Let us consider a feed-forward neural network composed of L layers. We indicate with $\ell_i(z, w_i)$ the function computed by the i-th layer when the input z is provided, while $w_i$ is the set of learnable parameters of that layer. If $f(x, \omega)$ is the function computed by the network when the input x is provided, being $\omega = \cup_{i=1}^{L} w_i$, we can formally describe f as

$$f(x, \omega) = (\ell_L(\cdot, w_L) \circ \ell_{L-1}(\cdot, w_{L-1}) \circ \ldots \circ \ell_1(\cdot, w_1))(x).$$

(1)

Given a scalar loss function $L$ that depends on the outputs and other available information $\gamma$ (e.g., supervisions), the Backpropagation algorithm allows computing $\nabla_w L$, the gradient of $L$ with respect to some $w_i$, by exploiting the composite

\textsuperscript{3}\url{https://pytorch.org/blog/accelerating-pytorch-with-cuda-graphs/}
structure of \( f \). Formally, if we indicate with \( o^j \) the output of the \( j \)-th layer,
\[
\nabla_w \mathcal{L}(f(x, \omega), \gamma) = \frac{\partial \mathcal{L}(o^L, \gamma)}{\partial o^L} \cdot \left( \frac{\partial \mathcal{L}(o^{L-1}, w_{L-1})}{\partial o_{L-1}} \cdot \frac{\partial \mathcal{L}(o^{L-2}, w_{L-2})}{\partial o_{L-2}} \right) \dots \frac{\partial \mathcal{L}(o^1, w_1)}{\partial w_1},
\]
where \( \cdot \) is the operator involved in the classic chain-rule. Let us assume that layers are divided into \( D \leq L \) non-overlapping sets also referred to as \emph{stages}, where the \( h \)-th stage collects consecutive layers with indices in \([a_h, b_h)\), \( 1 \leq a_h \leq b_h \leq L \), and \( a_{h+1} = b_h \). We indicate with \( p_h \) the \( h \)-th stage, involving layer functions \( \ell_{a_h}, \ell_{a_h+1}, \ldots, \ell_{b_h} \). The input/output of \( p_h \) are \( o_{a_h} \) and \( o_{b_h} \), respectively. Overloading the notation \( p_h \) to also refer to the function computed by the \( h \)-th stage, for simplicity, we can rewrite Eq. 1 as
\[
f(x, \omega) = (p_D(\vdots, \omega_D) \circ p_{D-1}(\vdots, \omega_{D-1}) \circ \cdots \circ p_1(\vdots, \omega_1))(x),
\]
Let us assume \( w_i \) to be the weights \( h \)-th stage \( p_h \), i.e., \( b_h \leq i \leq a_h \). Recalling that \( o_{a_h} = o_{b_h-1} \), for all valid \( h \), the gradient \( \nabla_w \mathcal{L}(\cdot) \) is then
\[
\nabla_w \mathcal{L}(f(x, \omega), \gamma) = \frac{\partial \mathcal{L}(o^D, \gamma)}{\partial o^D} \cdot \frac{p_D(o_{a_D}, \omega_D)}{\partial o_{a_D}} \cdot \frac{p_{a_D}(o_{a_D}, \omega_{a_D})}{\partial o_{a_{a_D}}(\ell_{a_{a_D}})} \cdot \frac{\mathcal{L}(o_{a_{a_D}}(\ell_{a_{a_D}}), \omega_{a_{a_D}})}{\partial \omega_{a_{a_D}}(\ell_{a_{a_D}})} \cdots \frac{\partial \mathcal{L}(\ell_{a_1}, w_1)}{\partial w_1},
\]
where the last term, \( \partial \mathcal{L}(\ell_{a_1}, w_1)/\partial w_1 \), can be computed by backpropagating the signal inside the \( h \)-th stage,
\[
\frac{\partial p_h(o_{a_h}, \omega_h)}{\partial w_i} = \frac{\partial \ell_{b_h}(o^{b_h-1}, w_{b_h})}{\partial o^{b_h-1}} \cdot \frac{\partial \ell_{b_h-1}(o^{b_h-2}, w_{b_h-1})}{\partial o^{b_h-2}} \cdots \frac{\partial \ell_{a_1}(o^{a_1}, w_1)}{\partial w_1},
\]
Let us consider a generic source stream \( S \), at each time instant \( t \) a new sample \( x^{(t)} \) is made available, and the following sample is provided after \( \Delta_S^{(t)} \) seconds. Without any loss of generality, we will consider the case in which \( \Delta_S^{(t)} \) is constant \( \forall t \), thus we will just use the notation \( \Delta_S \). For example, in the case of a video stream, \( \Delta_S = 1/\text{FPS} \), being FPS the constant frame rate of the video. Processing a data sample \( x^{(t)} \) requires computing \( f(x^{(t)}, \omega) \) and, in a lifelong learning horizon, also \( \nabla_w \mathcal{L}(f(x^{(t)}, \omega), \gamma) \), for all the valid \( i \)'s (plus the weight update operations), that is not an instantaneous processing, especially in very deep networks or when the input size is large. We indicate with \( \Delta_A \) such amount of processing time. Whenever \( \Delta_A > \Delta_S \), real-time learning is not possible, since buffering data would just create a constantly increasing queue and skipping frames would result in a loss of information that could limit the learning process. As a simple example, consider a network that learns by enforcing motion coherence over consecutive frames [2]. If there is too much distance (in time) between the frames of the pair, there might be poor correlation between them, thus making learning not feasible. Distributing computations over multiple devices is not enough, as the processing time is still limited by the sequential nature of the layer-wise computations.

In this paper we focus on an alternative way of organizing computations over time that is meant for hardware solutions equipped with \( D \) computational devices, in particular GPUs, each of them dedicated to the computations of a single stage \( p_h \). Before going into further details, we remark that what we propose is generic and essentially holds also for other types of devices that can work in parallel or in case of devices that include multiple parallel computational units. While a single GPU actually belongs to the latter category, its parallel computation capabilities are aimed at speeding up lower-level operations that usually exploit most of the GPU resources (e.g., CUDA blocks in NVIDIA cards [19]). As a result, a single GPU is not well-suited for speeding up, for example, multiple stages, even if scheduled for parallel execution. Let us assume that layer stages are created in a way to have very similar computational times for each stage in the target hardware. Hence, a device that process a single stage in \( \Delta_P \) seconds will require \( \Delta_A = D \Delta_P \) seconds to compute \( f(x^{(t)}, \omega) \), both in the case in which all the stages are sequentially executed on a single device, or if each stage is processed on an independent device, ignoring data transfer overheads. Conversely, a model equipped with pipeline parallelism is ready to process another sample whenever the first GPU has done processing the first stage, that only takes \( \Delta_P \), while the vanilla model takes \( D \Delta_P \) before being ready to process another sample from the stream. This implies that a pipeline scheme can real-time process the streamed data if \( \Delta_P \leq \Delta_S \), instead of \( \Delta_A = D \Delta_P \leq \Delta_S \), increasing the throughput by a factor \( D \).

As in classic pipeline parallelism, the \( h \)-th stage of the pipeline performs the forward and backward phases of the layers of \( p_h \), using the output that was provided by stage \( h-1 \) at the previous time step, and the gradients provided by stage \( h+1 \). In PARTIME, each stage also updates the weight values right after the backward computation has ended (Fig. 1). Thus, it is necessary to include an explicit time dependence in the set of weights characterizing each stage \( \omega_h^{(t)} \). Then, \( o_{b_h} \) at time \( t \) will be described by a temporal index on the weights \( \omega_h^{(t)} \),
\[
o_{b_h}^{(t)} = p_h(o_{b_h-1}^{(t-\Delta_P)}, \omega_h^{(t)}) = (p_h(\vdots, \omega_h^{(t)}) \circ \cdots \circ p_1(\vdots, \omega_1^{(t-(h-1)\Delta_P)}))(x^{(t-(h-1)\Delta_P)}),
\]
The network output at time \( t \) is \( o_D^{(t)} = p_D(o_{b_D-1}^{(t-\Delta_P)}, \omega_D^{(t)}) \), thus,
\[
f(x^{(t-(D-1)\Delta_P)}, \omega_D^{(t)}, \omega_D^{(t-(D-1)\Delta_P)}, \ldots, \omega_1^{(t-(D-1)\Delta_P)}) = (p_D(\vdots, \omega_D^{(t)}) \circ \cdots \circ p_1(\vdots, \omega_1^{(t-(D-1)\Delta_P)}))(x^{(t-(D-1)\Delta_P)}),
\]
that is essentially the analogous of (3) when making explicit the different time indices attached to the weights involved in processing a sample from the input stream. Of course, as
in every pipeline-based model, the output associated to each sample becomes available with a delay that is \( D\Delta p \), not a crucial issue in case of smoothly evolving streams with high frame rates, as the ones we consider in this paper.

The backward propagation in PARTIME follows a similar approach to the forward case, with gradients propagating through the pipeline from the loss function down to the first stage. Assuming \( u_i \) to belong to the weights \( \omega_h \) of stage \( p_h \) we generalize (4) to

\[
(\nabla_{w_i} L)^{(t)} = \frac{\partial \mathcal{L}(o_{D}, \gamma^{(t-(D-h)\Delta p)})}{\partial o_{D}^{b}} | _{o_{D}^{b} = o_{D}^{b}^{(t-(D-h)\Delta p)}} \frac{\partial p_{D}(o_{D-1}, \omega_{D}^{(t-(D-h)\Delta p)})}{\partial o_{D-1}^{b}} | _{o_{D-1}^{b} = o_{D-1}^{b}^{(t-(D-h)\Delta p)}} \frac{\partial p_{h+1}(o_{h}, \omega_{h+1}^{(t-\Delta p)})}{\partial o_{h}^{b}} | _{o_{h}^{b} = o_{h}^{b}^{(t-\Delta p)}} \frac{\partial p_{h}(o_{h-1}, \omega_{h})}{\partial w_{i}} | _{w_{i}^{(t-\Delta p)}}^{(t-\Delta p)}
\]

The last term \( \frac{\partial p_{h}(o_{h-1}^{(t-\Delta p)}, \omega_{h})}{\partial w_{i}} | _{w_{i}^{(t-\Delta p)}}^{(t-\Delta p)} \) is evaluated by means of a backpropagation of the signal within the given stage \( h \) according to (5), while the product of the previous terms is propagated through the stages. Notice that, focusing on a specific stage \( h \), the PARTIME computational structure is characterized by a delay consisting in \( 2(D - h) \) steps in-between the corresponding forward and backward “waves”. This leads to an approximation in the evaluation of the gradients since, during such a time interval, the input of the stages changes. Of course, the slower the input stream is varying, the less impacting is the approximation. A similar consideration holds for the values of weights, that get updated (thus they change) at each computational step, as we already anticipated in (7). Even if this will play a role in the evaluation of the gradients, a small learning rate can mitigate abrupt changes in the values of the weights, making the resulting approximation more appropriate.

IV. PARTIME

The PARTIME software library is written in PyTorch and it expects the user to provide a classic network of type torch.nn.Sequential, that is automatically converted into a format that will enable pipelined computations, both in the forward and backward phases. Despite the general formulation of the ideas behind PARTIME, the current version of the library leverages CUDA-based facilities, in particular CUDA Streams and CUDA Graphs, to setup the parallel execution scheme, thus it is designed for NVIDIA GPUs. In Algorithm 1 we provide a high-level description of the operations performed by each stage/device.

In order to activate the concurrent execution of independent queues of GPU tasks, PARTIME creates multiple CUDA Streams on each device. Each stream holds a queue of sequential tasks which are executed in-order, while the different streams interleave their tasks in no specified order. Since CUDA Streams are handled in an asynchronous manner (i.e., they do not wait the results of each enqueued task, even if such results are used in the rest of the code), PARTIME relies on specifically placed “events” (provided by the CUDA APIs) to ensure proper synchronization of the different streams.

Algorithm 1 Operations performed by each device/stage. Each stage expects input data to be stored in the so-called stable memory area, and it also has the use of a temporary one.

1: if 1st stage then
2: Copy \( x^{(t)} \) from host memory to the stable input memory area.
3: else
4: Copy temporary memory area to stable input memory area.
5: end if
6: Start forward step, using the stable input memory.
7: if last stage then
8: When 6 completes, copy last stage output to host memory.
9: else
10: Copy gradients from the next stage
11: When 6 completes, copy output to the next-stage temporary input memory area (avoids overwriting stable data).
12: end if
13: Compute stage-related gradients
14: Update weights.

A major drawback of a vanilla CUDA Stream-based implementation is the communication-overhead introduced when the CPU enqueues a task to the GPU streams. Such overhead quickly becomes negligible when dealing with tasks that run for a long time, but not when processing streams of data with the purpose of splitting computations into fast-processing parallel stages, where (in some configurations) enqueueing could take more time than the actual execution of each stage. We solve this by exploiting NVIDIA CUDA Graphs, that are able to handle the dependencies among the GPU tasks, generating a compact Directed Acyclic Graph (DAG) that summarizes all operations. This makes all the GPU tasks virtually collapse into a single one, with a scheduling overhead that is paid only once for all tasks. PARTIME also provides a stage balancing procedure that partitions network layers to make their computational times similar, using an algorithm comparable to that used in related libraries [20], but considering also data-transfer times.

V. Experiments

We performed several experiments to showcase the processing speedup obtained via the PARTIME library, which are presented in the following paragraphs.

A. Pixel-wise Predictions. Let us consider a neural architecture composed by 150 convolutional layers having fixed input/output resolution (i.e. no pooling or stride > 1), thus simulating the prediction of pixel-wise features. We assume the input tensors to have spatial resolution of \( R \times R \) pixels and 3 channels. Table I (top) shows the speedup in inference, averaged over 10000 forward steps, reporting in bold speedups that are greater than 80% of the theoretical speedup. We considered different settings, varying input resolutions \( R \) and number of output features \( F \) to evaluate the impact of the communication overheads with different data sizes. We also considered varying numbers of pipeline stages (#STAGES).
The results confirm the huge contribution of CUDA Graph in the low-resolution settings, where communications overheads are more impactful than computational times, while advantages of CUDA Graph are less evident with bigger input resolutions. In all the cases PARTIME provides significant improvements over the vanilla network. With \( R = 1024 \) PARTIME yields a speedup \( \approx 7 \), close to the theoretical speed-up of 8. In Table I (bottom) we report the learning case, that includes both the backward phase and optimization step. Speedups are even greater, as backward computations increase the computational cost of each single pipeline stage, reducing the relative impact of data-transfer overhead.

### B. Image Classification.

The second experimental activity is about neural architectures composed of 150 convolutional layers interleaved with pooling layers, with the final output pooled to a vector with \( F \) elements, the number of output classes (1 vector per image). We evaluate potential pipeline balancing issues due to the different spatial resolutions of the layers. Table II (top, same structure of Table I, reporting in bold the cases in which the speedups are greater than half of the theoretical one) shows that CUDA Graph yields the best results even in this experience. The aforementioned layer/stage balancing issue causes an under-utilization of some of the GPUs, that is more evident in the case of \( R = 1024 \), where the maximum speed-up is almost \( 8 \) even with 8 GPUs.

### C. Image Classification with Residual Connections.

We performed another image classification experience using a ResNet-152 architecture, customizing the final classification head to yield \( F \) output classes/features. Skip connections from layer \( i \) to layer \( j > i \) are propagated through all the stages in-between \( i \) and \( j \), by means of identity mappings. Therefore, we remark that skip connections introduce further copy operations between GPU devices. Nonetheless, the speed-up achieved by PARTIME are similar to the previous experience Settings with \#STAGES \in \{2, 4\} get closer to the theoretical speed-up, since the data that is about the skip connections need to be propagated through less stages.

### D. Continual Online Image Classification.

We simulated a continual online learning process where a neural model is trained on the CIFAR-10 dataset. The training procedure sequentially collects inputs using a sliding window to build up a “replay batch”, which is then fed to the pipeline. Each time a new sample is provided, the oldest input in the replay batch is replaced by the new one. This approximates the assumption of processing slowly-varying inputs. The learning rate is also chosen accordingly to avoid changing the weights too much between each step. We considered a ResNet-50 with 10-classes classification head, trained using the Adam optimizer with learning rate \( \mu = 0.001 \), streaming the whole dataset 5 times, testing different sliding-window/batch sizes in \{64, 256, 1024\}. Fig. 2 compares the accuracy on the test data for the different batch sizes. Training a 2-stage pipelined network takes around half of the time of the sequential model. When increasing the batch size, better models are learned as we approach the slowly-changing input condition.

### E. Continual Online Optical Flow Estimation.

Our last experimental activity considers the optical flow learning problem [21], replicating the experience of [22]. A CNN takes as input a frame pair and it outputs the displacement components for all the pixels. Training is performed using a single frame pair at every step, and the pairs are sequentially extracted from

---

**Table I**

| #STAGES | Without CUDA Graph | With CUDA Graph |
|---------|--------------------|-----------------|
| R / F   |                    |                 |
| 2       | 1.12 1.17 1.15     | 1.89 3.74 6.41  |
| 4       | 1.09 1.12 1.10     | 1.68 3.65 6.46  |
| 8       | 1.08 1.11 1.08     | 1.72 3.13 5.05  |
| 1024 / 1| 1.82 3.28 5.27     | 1.86 3.87 7.14  |
| 1024 / 10| 1.82 3.21 5.53    | 1.88 3.57 6.65  |
| 1024 / 100 | 1.72 2.79 4.44 | 1.69 2.99 4.81 |

**Table II**

| #STAGES | Without CUDA Graph | With CUDA Graph |
|---------|--------------------|-----------------|
| R / F   |                    |                 |
| 2       | 1.16 1.13 1.06     | 0.78 2.59 4.31  |
| 4       | 1.11 1.04 1.00     | 0.99 2.55 3.48  |
| 8       | 1.10 1.06 1.00     | 1.69 2.67 4.39  |
| 1024 / 1| 1.63 1.67 1.18     | 1.43 2.61 2.89  |
| 1024 / 10| 1.66 1.64 1.38    | 0.80 1.64 2.52  |
| 1024 / 100| 1.43 1.63 1.49   | 1.20 1.24 2.79  |

---

![Fig. 2. Test accuracy in the continual online image classification described in the main text. For reference, the baseline accuracies for a sequential network trained without pipelining are 0.489, 0.553, and 0.639 for batch sizes of 64, 256 and 1024, respectively.](image)
a video source without shuffling. Unsupervised learning is driven by the brightness constancy assumption coupled with spatial regularization (see [22] for details). As video source, we choose “1917”, a 2019 British war film by Sam Mendes. The film appears as a single continuous take, without artificial cuts. We report in Fig. 3 the optical flow estimated by the sequential and pipelined network. We selected a learning rate $\mu = 10^{-5}$ for both the sequential and the 2-stage pipeline, while it is set to $\mu = 10^{-9}$ for the 4-stage pipeline to account for the assumption of slowly changing gradients. The estimated flow from the pipelined network is qualitatively similar to that of the sequential network. Fig. 4 shows how the training loss changes every 1-minute window in the movie for all settings, showing that the pipeline approximation yields a similar learning curve.

VI. CONCLUSIONS AND FUTURE WORK

We presented and shared PARTIME, a Python software library designed for continual learning problems in which the data is streamed over time. PARTIME is built on a pipeline parallelism that speedups the computations of a neural network by a theoretical $D \times$, being $D$ the number of devices. We focused on the case of Graphics Processing Units (GPUs), showing that our implementation scales coherently with the expectations, as experimented in an environment with up to 8 GPUs. Future work will consider and improve the implementation for specific neural models and local approaches [6].

REFERENCES

[1] M. Delange, R. Aljundi, M. Masana, S. Parisot, X. Jia, A. Leonardis, G. Slabaugh, and T. Tuytelaars, “A continual learning survey: Defying forgetting in classification tasks,” IEEE Transactions on Pattern Analysis and Machine Intelligence, pp. 1–1, 2021.
[2] A. Betti, M. Gori, and S. Melacci, “Learning visual features under motion invariance,” Neural Networks, vol. 126, pp. 275–299, 2020.
[3] M. Maggini, G. Marra, S. Melacci, and A. Zugarini, “Learning in text streams: Discovery and disambiguation of entity and relation instances,” IEEE Transactions on Neural Networks and Learning Systems, vol. 31, no. 11, pp. 4475–4486, 2020.
[4] A. Betti, M. Gori, and S. Melacci, “Cognitive action laws: The case of visual features,” IEEE Transactions on Neural Networks and Learning Systems, vol. 31, no. 3, pp. 938–949, 2020.
[5] Y. Huang, Y. Cheng, A. Bapna, O. Firat, M. X. Chen, D. Chen, H. Lee, J. Ngiam, Q. V. Le, Y. Wu, and Z. Chen, “Gpipe: Efficient training of giant neural networks using pipeline parallelism,” 2018. [Online]. Available: https://arxiv.org/abs/1811.06065
[6] G. Marra, M. Tiezzi, S. Melacci, A. Betti, M. Maggini, and M. Gori, “Local propagation in constraint-based neural networks,” in 2020 International Joint Conference on Neural Networks (IJCNN), 2020, pp. 1–8.
[7] M. Tiezzi, G. Marra, S. Melacci, and M. Maggini, “Deep constraint-based propagation in graph neural networks,” IEEE Transactions on Pattern Analysis and Machine Intelligence, vol. 44, no. 2, pp. 727–739, 2022.
[8] A. Betti and M. Gori, “Backprop diffusion is biologically plausible,” 2019. [Online]. Available: https://arxiv.org/abs/1912.04635
[9] A. Betti, M. Gori, S. Marullo, and S. Melacci, “Developing constrained neural units over time,” in 2020 International Joint Conference on Neural Networks (IJCNN), 2020, pp. 1–8.
[10] O. Yadon, K. Adams, Y. Taigman, and M. Ranzato, “Multi-gpu training of convnets,” 2013. [Online]. Available: https://arxiv.org/abs/1312.5853
[11] A. Krizhevsky, “One weird trick for parallelizing convolutional neural networks,” 2014. [Online]. Available: https://arxiv.org/abs/1404.5997
[12] S. Li, Y. Zhao, R. Varma, O. Salpekar, P. Noothilias, T. Li, A. Paszke, J. Smith, B. Vaughan, P. Damania, and S. Chintala, “Pytorch distributed: Experiences on accelerating data parallel training,” Proceedings of the VLDB Endowment, vol. 13, no. 12, pp. 3005–3018, 2020.
[13] J. Rasley, S. Rajbhandari, O. Ruwase, and Y. He, “Deepspeed: System optimizations enable training deep learning models with over 100 billion parameters,” in KDD 2020, 2020, pp. 3505–3506.
[14] M. Shoeybi, M. Patwary, R. Puri, P. LeGresley, J. Casper, and B. Catanzaro, “Megatron-lm: Training multi-billion parameter language models using model parallelism,” 2019. [Online]. Available: https://arxiv.org/abs/1909.08053
[15] L. Guan, W. Yin, D. Li, and X. Lu, “Xpipe: Efficient pipeline model parallelism for multi-gpu dnn training,” 2019. [Online]. Available: https://arxiv.org/abs/1911.04610
[16] D. Narayanan, A. Harlap, A. Phanishayee, V. Seshadhri, N. R. Devanur, G. R. Ganger, P. B. Gibbons, and M. Zaharia, “Pipelread: generalized pipeline parallelism for dnn training,” in Proceedings of the 27th ACM Symposium on Operating Systems Principles, 2019, pp. 1–15.
[17] D. Narayanan, A. Phanishayee, K. Shi, X. Chen, and M. Zaharia, “Memory-efficient pipeline-parallel dnn training,” in Proceedings of the 38th International Conference on Machine Learning, ser. Proceedings of Machine Learning Research, vol. 139. PMLR, 18–24 Jul 2021, pp. 7937–7947.
[18] A. Petrowski, G. Dreyfus, and C. Girault, “Performance analysis of a pipelined backpropagation parallel algorithm,” IEEE Transactions on Neural Networks, vol. 4, no. 6, pp. 970–981, 1993.
[19] J. Cheng, M. Grossman, and T. McKercher, Professional CUDA c programming. John Wiley & Sons, 2014.
[20] C. Kim, H. Lee, M. Jeong, W. Baek, B. Yoon, I. Kim, S. Lim, and S. Kim, “torchgpipe: On-the-fly pipeline parallelism for training giant models,” arXiv preprint arXiv:2004.09910, 2020.
[21] T. Brox, A. Bruhn, N. Papenberg, and J. Weickert, “High accuracy optical flow estimation based on a theory for warping,” in European conference on computer vision. Springer, 2004, pp. 25–36.
[22] S. Marullo, M. Tiezzi, A. Betti, L. Faggì, E. Meloni, and S. Melacci, “Continual Unsupervised Learning for Optical Flow Estimation with Deep Networks,” in Proceedings of the 1st Conference on Lifelong Learning Agents (CoLLAs). PMLR, 2022. [Online]. Available: https://sailab.disim.unisi.it/continual-unsupervised-learning-for-optical-flow-estimation/