ON ANYTIME LEARNING AT MACROSCALE

Anonymous authors
Paper under double-blind review

ABSTRACT

Classical machine learning frameworks assume access to a possibly large dataset in order to train a predictive model. In many practical applications however, data does not arrive all at once, but in large batches over time. This creates a natural trade-off between accuracy of a model and time to obtain such a model. A greedy predictor could produce non-trivial predictions by immediately training on batches as soon as these become available but, it may also make sub-optimal use of future data. On the other hand, a tardy predictor could wait for a long time to aggregate several batches into a larger dataset, but ultimately deliver a much better performance. In this work, we consider such a streaming learning setting, which we dub *anytime learning at macroscale* (ALMA). It is an instance of anytime learning applied not at the level of a single chunk of data, but at the level of the entire sequence of large batches. We first formalize this learning setting, we then introduce metrics to assess how well learners perform on the given task for a given memory and compute budget, and finally we test about thirty baseline approaches on three standard benchmarks repurposed for anytime learning at macroscale. Our findings indicate that no model strikes the best trade-off across the board. While replay-based methods attain the lowest error rate, they also incur in a 5 to 10 times increase of compute. Approaches that grow capacity over time do offer better scaling in terms of training flops, but they also underperform simpler ensembling methods in terms of error rate. Overall, ALMA offers both a good abstraction of the typical learning setting faced everyday by practitioners, and a set of unsolved modeling problems for those interested in efficient learning of dynamic models.

1 INTRODUCTION

Empirical risk minimization (Vapnik, 1998) is the dominant framework to formalize the learning process of a supervised task, and it has been critical to the success of large scale training of deep learning systems on a wide variety of applications. Within this framework, training data is assumed to be provided to the learner all at once. Alternatively, when the dataset is very large (essentially infinite), data is streamed to the learner one minibatch at the time, assuming that the rate at which samples are received matches the model's processing time to learn from them.

In practice however, training data often comes at a *slower* rate than a model can process it. Moreover, it comes in relatively large batches once in a while. For instance, teams deploying language modeling applications, build models that are trained on large amounts of data like filtered versions of Common Crawl, which are dumps of the internet. However, new snapshots are available every month, as new content is organically generated. Therefore datasets keep getting bigger every few months and models need to be retrained accordingly. Similarly, visual object recognition datasets used in deployed applications are often extended every few months to include new images with their corresponding annotations.

There are two main approaches to integrate information present in a new batch of data in an existing model. If a lot of computational resources are available, a new and bigger model is instantiated and trained *from scratch* on the union of the old training set with the new batch of data. However, since this is a computationally very intensive process, retraining is typically done only rarely, once several batches of data have been collected. We call this approach "tardy" large-scale learning, since a predictor is available only at a later time. Another option, particularly suitable when computational

* Authors contributed equally

resources are scarce and a predictor is needed quickly, is to simply finetune the old model on the new data as this arrives.

This trade-off is emblematic of *anytime learning*, a learning setting where a learner has to provide good predictions at any point in time, while improving its performance over time as more and more data is observed. From an anytime learning perspective, both training a large model after all data is received and finetuning on the newly added batch of data are not satisfying. The former approach is a poor anytime learner because one needs to wait for a long time before obtaining a useful predictor. The latter approach is a poor anytime learner because it typically cannot leverage very well future batches of data since the model has a constrained capacity, determined on a small portion of the overall dataset and because inherently the model is trained on non i.i.d. data.

In this work, we aim at exploring this accuracy versus time trade-off of anytime learning, not at the level of a single batch of data, but at the macroscale of the entire sequence of batches. This is a setting which more closely mimics practical applications, that we call *anytime learning at mascroscale* (ALMA). In this learning setting, we assume that the time to train a model is negligible compared to the interval of time between two consecutive batches of data (and therefore we do not care about how quickly a learner adapts to a new batch), yet efficiency matters in the sense that for the same performance a predictor that uses less compute and memory is preferable. In summary, we are interested in a learner that i) yields high accuracy, ii) can make non-trivial predictions at any point in time while iii) limiting its computational and memory resources.

Our first contribution is to formalize the ALMA problem and to introduce metrics to evaluate learners (§3). We consider three different axes: error rate, memory and amount of computation. By measuring these quantities against time, via an area under the curve, we account not only for the final performance but also for the whole training trajectory over the sequence of large batches of data.

Our second contribution is an extensive empirical evaluation (§5) of various models (§4) that strike different trade-offs between accuracy and time to obtain a useful predictor. In particular, we explore models that fall in between greedy finetuning and tardy large-scale learning, and investigate models that leverage batches of data at an intermediate rate. We also consider a rich family of modular architectures, from plain ensembling methods to hierarchical mixture of experts, and several variants thereof, including those that have access to a replay buffer storing all previous batches of data and those that can grow capacity over time.

Our findings across three different benchmarks, including a large scale language modeling one, can be summarized as follows. a) An intermediate waiting time offers the best trade-off between accuracy and time to yield such a predictor. However, b) there is no single approach striking the best trade-off between performance and efficiency for various model sizes. c) Retraining from scratch a big model does offer the lowest error rate but sacrifices efficiency. d) Interestingly, large models are the most statistically efficient even when considering small datasets (like MNIST) and fully connected networks. e) While approaches to grow capacity exhibit gains in terms of computational efficiency, these do not even outpeform simple ensembles. Overall, our work points at several research opportunities to improve modeling in a streaming setting of broad practical relevance, rather then pointing at any particular solution. We have also released code to reproduce our experiments and the entire platform implementing ALMA.

2 Related Work

ALMA relates to several other learning frameworks. It shares the same assumptions of classical empirical risk minimization (ERM) (Vapnik, 1998) at the level of each batch of data. However, it overall violates ERM's assumptions of i.i.d. observations, because data points come in a stream of data chunks. Because of this, ALMA relates to continual learning (CL) (Ring, 1994; Thrun, 1994; Ring, 1997; Thrun, 1998), with the key difference that the data distribution across batches (or tasks) is assumed stationary in ALMA. Therefore, ALMA can be seen as a special case of CL. ALMA relates also to online learning (Bottou, 1998). However, in online learning examples are streamed one at the time (or at random from a large dataset), while in ALMA the learner receives large batches of data sequentially. Finally, ALMA relates more broadly to transfer learning (Pan & Yang, 2010), as the problem of adapting to a new batch of data can be interpreted as leveraging knowledge acquired on previous batches to more effciently learn from the new batch of data.

Of course, ALMA relates to anytime learning (Grefenstette & Ramsey, 1992; Ramsey & Grefenstette, 1994), which has been recently applied to compare various autoML frameworks (Liu et al., 2020). Unlike traditional anytime learning however, in this work we are not interested in assessing the anytime learning ability at the level of each chunk of data, but only at a coarser granularity, at the level of the entire stream of chunks. Inspired by Liu et al. (2020), we consider the area under the curve of error rate against time to measure performance, but in order to account also for compute and memory budget, we add to our evaluation metrics also the area under the curve for memory and compute.

From the more theoretical side, there has been work about sub-bagging (Bühlmann & Yu, 2002) (bagging using subsets of a larger dataset) which is similar to our setting but without the sequential aspect of it. In this context, Breiman (1999) proposed a model similar to our growing ensembling (gEns), Bühlmann & Yu (2002) studied sub-bagging as a way to make the prediction of tree classifiers more robust while Zou et al. (2021) studied the consistency of the estimator in this setting. We defer to future studies the analysis of ALMA, while in this work we focus on the empirical evaluation.

Shifting the discussion to prior work on models that adjust their capacity dynamically, Waterhouse & Robinson (1995) introduced an approach to grow a hierarchical mixture of experts model (Jordan & Jacobs, 1994). This is a tree structured model where experts are at the leaves and gating functions are at non-terminal nodes. The tree determines a hierarchical partition of the input space into regions that are associated to each expert. This approach was made more efficient in later work by (Fritsch et al., 1996). In this work we consider a baseline (gMoE) that extends this prior work to hierarchical mixture of experts (Eigen et al., 2014; Denoyer & Gallinari, 2015; Lepikhin et al., 2020).

Growing architectures have also been studied in CL. For instance, Fernando et al. (2017) and Veniat et al. (2021) proposed a modular architecture that is assembled for every task, possibly reusing previously trained modules. The major difference with our work is that in our case routing is input dependent as opposed to task dependent. Yoon et al. (2018) instead proposed a method to incrementally and smoothly add hidden units. Similarly, Wen et al. (2020) proposed a heuristic approach to automatically adjust the network depth. Wang et al. (2017) considered growing both depth and width when finetuning to a new task. Liu et al. (2019a) and Wu et al. (2020) proposed approaches to grow architectures in depth and width by leveraging Taylor approximation and greedy selection. In our work, we benchmark against this last variant. None of these approaches have been applied to the ALMA setting to date.

Finally, some of our findings are built upon and extend recent empirical evaluations studying the scaling properties of language models (Kaplan et al., 2020a; Li et al., 2020b). In this study, we confirm the conclusion that bigger models generalize better and are more statistically efficient, not only in language modeling tasks using a transformer architecture, but also in smaller scale computer vision tasks using both fully connected and convolutional architectures.

3 Learning Setting

In anytime learning at macroscale (ALMA), we assume that there exists an underlying data distribution $p(x, y)$ with input $x \in \mathbb{R}^D$ and desired label $y \in \{1, \ldots, C\}$. Notice that extensions to regression and unsupervised learning (where y is missing) are trivial, and therefore in this work we focus on classification problems for simplicity of exposition. A important property of ALMA is that data is presented to the learner as a stream $\mathcal{S}_B$ of B consecutive batches of examples. Let $\mathcal{D}_i$ be a collection of $N \gg 0$ i.i.d. samples randomly drawn from $p(x, y)$, for $i \in \{1, \ldots, B\}$. The stream is then defined as the ordered sequence $\mathcal{S}_B = \{\mathcal{D}_1, \ldots, \mathcal{D}_B\}$. We refer to each dataset $\mathcal{D}_i$ as *mega-batch*, as it is composed by a large number of examples. Typically a learner $m : \mathbb{R}^D \to \{1, \ldots, C\}$ updates its parameters by processing a *mini-batch* of $n \ll N$ examples at the time from each mega-batch $\mathcal{D}_i$, and by iterating several times over each mega-batch before being presented with the next mega-batch. Since the learner cannot access future mega-batches, overall the data distribution is not i.i.d., even though samples drawn from each mega-batch are i.i.d., and cross-validation is performed using a subset of the current mega-batch. A learner could decide to use previous mega-batches when learning on the current mega-batch, but this will increase its compute usage.

Finally, we assume that the time it takes a learner to update its internal parameters after having observed a mega-batch is much less than the interval between the arrival of two consecutive mega-

batches. In other words, the rate at which data arrives is slower than the processing time of the model, and therefore the model could decide to iterate several times over the data at its disposal to improve its prediction accuracy.

3.1 METRICS

We evaluate learners in the ALMA setting across three axes, namely: accuracy, memory and computation. Let t be the time at which the t-th mega-batch arrives; this data can be used by the model to update its parameters or it is simply aggregated to previous mega-batches for later use.

We compute the error rate of model m at time t (after the arrival of the t-th mega-batch) and compute the area under the curve obtained varying t from 0 till the total number of mega-batches B; the resulting cumulative error rate (CER) is:

$$\text{CER} = \sum_{t=0}^{B} \frac{1}{|\mathcal{D}^{\text{Ts}}|} \sum_{(x,y)\in\mathcal{D}^{\text{Ts}}} |m(x;\theta_t) \neq y| \tag{1}$$

where $m(x;\theta_t)$ is the model at time t equipped with parameters θ_t, $\mathcal{D}^{\text{Ts}}$ is the test set, $|\mathcal{D}^{\text{Ts}}|$ is the number of examples in the test set, and $|m(x;\theta_t) \neq y|$ is one if the model prediction does not match the ground truth label and zero otherwise. The outer sum computes the discrete integral of the error rate over time. CER is going to be small only when the error rate is small throughout the whole stream . CER is instead large for a tardy model that waits till the very last mega-batch to update the model, even though eventually this may obtain a very low final error rate.

Similarly, we compute the cumulative memory usage and compute as:

$$\text{Mem} = \sum_{t=0}^{B} |\theta_t|, \quad \text{Comp} = \sum_{t=0}^{B} \mathcal{O}(m(\cdot;\theta_t)) \tag{2}$$

where $|\theta_t|$ is the number of free parameters of the model at time t, and $\mathcal{O}(m(\cdot;\theta_t))$ is the number of flops used by the model to process the t-th mega-batch. Once again, by measuring the area under the curves obtained by tracking these quantities over time we obtain a holistic assessment of memory and compute throughout the whole stream. A model can obtain small Mem and Comp only if it does not consume memory and if it is computationally parsimonious throughout the entire duration of the stream.

4 LEARNING ALGORITHMS

In this section, we describe the methods we tested in the ALMA setting. They generally follow the learning procedure shown in Algorithm 1. At a high level, we consider two families of models, those with a monolithic architecture and those with a modular architecture (e.g., ensembling). The latter are amenable to grow over time by adding new modules to the existing set. We will start by describing fixed architectures (§4.1) and then conclude with growing architectures (§4.2). All models are also given the option to replay previous mega-batches.

4.1 FIXED ARCHITECTURES

The first family of methods trains models with a fixed architecture. These models are sequentially trained over new mega-batches and exhibit a fixed memory footprint. We consider three models:

Single Model (*SM*): This is a standard multi-layer neural network (e.g., fully connected neural network or transformer) trained by stochastic gradient descent. It can be initialized from random or from the parameters of the model trained on the previous mega-batch. The initializaiton choice is determined via cross-validation.

Ensemble of Models (*Ens*): The second approach is the simplest modular approach, consisting of an ensemble of N neural networks with the same architecture, each being trained independently on the same sequence of data. The output of the overall model at test time is the average probability

Algorithm 1 Training in the ALMA setting

procedure TRAIN(m, w, replay, grow) ▷ m is the model, w is the waiting time
  $t \leftarrow 1$
  $\mathcal{D} \leftarrow \emptyset$
  while $t < B$ **do** ▷ For each stage
    if replay **then** ▷ Acquire w mega-batches
      $\mathcal{D} \leftarrow \mathcal{D} \cup \mathcal{D}_t \cup ... \cup \mathcal{D}_{t+w-1}$
    else
      $\mathcal{D} \leftarrow \mathcal{D}_t \cup ... \cup \mathcal{D}_{t+w-1}$
    $t \leftarrow t + w$
    if grow **then**
      $m.grow()$ ▷ Grow the model if the model is a growing model
    $m.train(\mathcal{D})$ ▷ Fine-tune or retrain from scratch m on the collected dataset

distribution produced by each component[1]. The advantage of *Ens* is that training and inference can be trivially parallelized, enabling to scale up model parameters very easily. The disadvantange is that inference requires N times more compute than what is required by each component.

Uniform Mixture of Models (*UMix*): A potential drawback of *Ens* is that evaluation and training are inconsistent. *UMix* addresses this by training a model whose prediction is the average (in logit space) of the predictions produced by N networks. While this requires synchronization during training, now both training and evaluation use the same model.

4.2 GROWING ARCHITECTURES

In the previous section, the number of parameters and the architecture of the model are fixed throughout the model's lifetime. However, as more data is observed, it is interesting to consider dynamic architectures that grow over time, because these may save compute and memory during the earlier stages of learning while providing more predictive power during the later stages. We consider three growing approaches:

Growing Ensemble (*gEns*): Like the *Ens* model, *gEns* is also a combination of neural networks trained independently. While *Ens* considers N networks that are, at each stage, trained over the new chunck of data, *gEns* replaces this step by a growing step where n neural networks are added. In our implementation, only these n neural networks are trained over the new data, while the other neural networks (trained on previous mega-batches) are kept fixed.

Growing Mixture of Experts (*gMoE*): A hierarchical mixture of experts models (MoE) is an architecture where at layer l the output representation z^l is: $z^l = \sum_{j=1}^{k} g(j|z^{l-1})h(z^{l-1}|j)$, where g is the gating or routing function and $h(\cdot|j)$ is the j-th expert. Compared to *Ens*, MoE has exponentially many more components albeit with a lot of parameter sharing. Another advantage is that by selecting only one (or a few) experts, the computational cost is independent of the number of experts, assuming the cost of gating is negligible compared to the cost or executing the experts. The main issue is that MoE are notoriously harder to train (Eigen et al., 2014; Denoyer & Gallinari, 2015; Lepikhin et al., 2020). In this work, we consider a growing version of MoE, which we denote with *gMoE*, whereby experts are added over time. This has a tree structured gating function where leaves correspond to experts. We calculate each expert's contribution to the total loss by summing the losses of the examples routed through that expert, and "split" the expert responsible for the largest loss. The split is performed by adding an expert with the same parameters, and turning the corresponding leaf node of the gate into a binary internal node with a child leaf for the old and new expert. This process guarantees that right before and right after a growth step the loss is the same. See Appendix A for more details.

[1]Classical bagging approaches and majority vote strategies have been also explored without significant difference.

Firefly (Wu et al., 2020) (*FF*): *FF* is a method which progressively grows neural networks, jointly optimizing both the model architecture and parameters. Growth includes both a width expansion by adding new hidden units (or feature maps) as well as a depth expansion by adding new layers. Importantly, this is an example of non-modular method unlike *Ens* or *gMoE*, which is potentially more expressive but also more inefficient at inference time because there is no structured sparsity that can be leveraged to speed up computation.

5 EXPERIMENTS

In this section we first describe how standard benchmarks can be repurposed for ALMA, we then provide the details of the models we tested, and we finally conclude with an analysis of the results we obtained, aiming to understand which method attains the best trade-off between time, accuracy, compute and memory usage.

Datasets We consider a variety of datasets. The first dataset is MNIST (LeCun et al., 1998), which consists of a training set with $60{,}000$ quasi-binary handwritten digits of size 28x28 pixels, and a test set with $10{,}000$ examples.

The second dataset is CIFAR 10 (Krizhevsky, 2009) that has a training set with $50{,}000$ images of size 32x32 pixels belonging to 10 classes such as bird, car, horse, ship, truck, etc.

The third dataset, used for our large-scale language modeling evaluation, is a portion of the collection of English language text introduced in Liu et al. (2019b), consisting of Books, Wikipedia and Common Crawl. We consider 4 (large) mega-batches for training and one additional mega-batch for evaluation, each consisting of approximately 440M words; we also hold out a validation set with approximately 0.5M words of Common Crawl for model selection. We use a byte-pair encoding (BPE) (Sennrich et al., 2016) vocabulary with $50{,}000$ units, following Radford et al. (2019). This dataset is fairly representative of what practitioners might face when maintaining a deployed system with new data arriving every few months.

Given a dataset like any of the above, we construct a benchmark for ALMA evaluation as follows: 1) we randomly partition the training set into B mega-batches with equal number of training examples ($B = 50$ for MNIST and CIFAR 10, and 4 for the text dataset), 2) from each mega-batch we extract 10% of the data to build the mega-batch validation set (except for the large scale language modeling dataset where we use the provided validation set), and 3) we create a learning experience by doing one pass over the sequence of mega-batches. For each mega-batch, the learner can query as many mini-batches as desired. The learner can also decide not to train on the data of a mega-batch right away but instead to wait and accumulate data across a few consecutive mega-batches. While the learner observes data, it is also tested on the test set. This is not used for validation purposes, but only for final reporting as shown in §5.1.

Models We evaluate the six approaches presented in §4, and for each of them we consider various waiting times, a version with and without replay, and at least two model sizes. For each setting, we cross validate over several hyper-parameters such as initializaiton type, learning rate, stopping criterion, growth rate, etc.

Next, we describe in details the architecture used on each dataset. Further experimental details to aide reproducibility are reported in Appendix B. On MNIST the backbone architecture of *SM* is a three layer fully connected neural network with ReLU units. We considered two hidden units sizes, namely 4 and 32 (denoted by [s] and [b], respectively), which let us simulate the regime of big data relative to the size of the network and explore how to grow architectures without worrying about overfitting. Similarly, the components of *Ens*, *gEns* and *UMix* are *SM* networks of the same size as stated above; *gMoE* also starts off as *SM* and adds modules (at the first two layers) that have the same size as the original layer of *SM*. When varying the waiting time, i.e., the number of mega-batches that are aggregated before initiating a new training session, we use the suffix "_w" to indicate its value.

On CIFAR 10, the methods and notations are the same as in MNIST. The only difference is that the backbone architecture is a scaled down version of a VGG19 convolutional neural network (Simonyan & Zisserman, 2015), where the number of intermediate feature maps is the same for each layer and

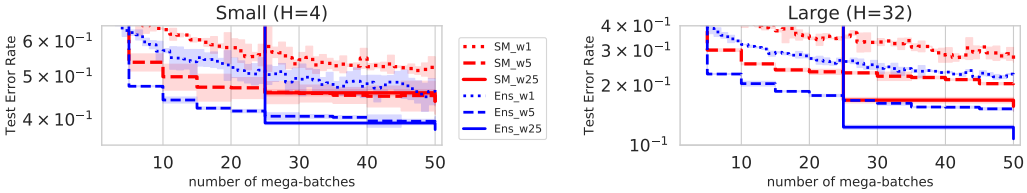

Figure 1: Accuracy over time for small models (left) and large models (right) on CIFAR 10 with no replay.

Method	$\lvert\theta\rvert$	Cum. Train TFLOPS	Inference MFLOPS	CER
SM	3210	0.02 (0.09)	0.003	$1054_{\pm 25}$ ($1062_{\pm 61}$)
	16330	0.09 (0.48)	0.016	$631_{\pm 10}$ ($616_{\pm 9}$)
	26506	0.14 (0.79)	0.026	$582_{\pm 12}$ ($554_{\pm 5}$)
	132127	0.71 (3.92)	0.132	$504_{\pm 8}$ ($485_{\pm 4}$)
	269322	1.45 (7.98)	0.269	$497_{\pm 8}$ ($474_{\pm 4}$)
Ens	16050	0.09 (0.47)	0.080	$870_{\pm 9}$ ($885_{\pm 18}$)
	132530	0.71 (3.93)	0.661	$517_{\pm 6}$ ($493_{\pm 5}$)
UMix	16050	0.09 (0.47)	0.080	$668_{\pm 17}$ ($646_{\pm 18}$)
	132530	0.71 (3.93)	0.661	$508_{\pm 10}$ ($486_{\pm 5}$)
gEns	3210 $\rightarrow$ 32100	0.02 (0.09)	0.003 $\rightarrow$ 0.032	$1020_{\pm 39}$ ($1027_{\pm 29}$)
	26506 $\rightarrow$ 265060	0.14 (0.79)	0.026 $\rightarrow$ 0.264	$551_{\pm 6}$ ($543_{\pm 7}$)
gMoE	7950 $\rightarrow$ 50610	0.16 (1.08)	0.008 $\rightarrow$ 0.050	$863_{\pm 17}$ ($766_{\pm 16}$)
	54318 $\rightarrow$ 304626	0.97 (6.55)	0.054 $\rightarrow$ 0.304	$573_{\pm 8}$ ($535_{\pm 5}$)

Table 1: MNIST results with waiting time equal to 5. Rows with parameter size in blue refer to small networks (e.g., *Ens* totalling 16050 parameters has five components of the same size of *SM* shown in blue); similarly, rows in orange refer to the large network size. Numbers in parentheses are for versions with replay. For all metrics smaller is better.

equal to either 4 or 32. On this dataset, we also consider *FF* starting off from the same VGG19 backbone.

For the language modeling task *SM* is a Switch Transformer (Fedus et al., 2021), which is a hard mixture of experts model with an additional load balancing loss term and hard capacity constraint applied during training to prevent uneven expert utilization. Following Fedus et al. (2021), we fix the weight of the balancing loss term to 0.01 and use a capacity factor of 1, ensuring relatively uniform expert utilization. We train the model using Adam (Kingma & Ba, 2015) and tune the learning rate and dropout on the validation set. In the growing setting we copy the expert weights and gating network weights corresponding to the top-k experts incurring the largest loss, where k is typically between 2 and 4. This growing procedure preserves a flat mixture and adds multiple experts at the time which is simpler to get to work at scale, as consider that only the experiments reported in Tab. 3 took over 100 GPU days to complete. We consider two model sizes: a *base* model with 6 layers and model dimension of 512, for a total of 40M shared parameters and 6M additional parameters per expert; and a *large* model with 12 layers and model dimension of 768, for a total of 96M shared parameters and 28M additional parameters per expert. We use an input sequence length of 512 tokens and we do not use replay given the large chunk sizes.

5.1 Results

In Fig. 1 we start by analyzing learning curves on CIFAR 10 for a subset of the methods as a function of the waiting time. We then dive into analyzing all methods on both MNIST (Tab. 1) and CIFAR 10 (Tab. 2), using the optimal empirical value of waiting time. We conclude by confirming the major findings at scale on the language modeling task (Tab. 3).

Fig. 1 shows the test error rate as a function of the number of mega-batches received for both the small (left) and the large (right) model. We observe that an intermediate waiting time (in this case equal to 5) strikes the best trade-off between accuracy and time for all methods, since curves with waiting

Method	$\lvert\theta\rvert$	Cum. Train TFLOPS	Inference MFLOPS	CER
SM	2510	3.6 (20.0)	0.4	$2706_{\pm 132}$ ($2643_{\pm 39}$)
	11710	15.0 (82.5)	1.7	$2038_{\pm 32}$ ($1842_{\pm 51}$)
	31660	38.6 (212.5)	4.3	$1764_{\pm 43}$ ($1561_{\pm 7}$)
	140970	164.9 (907.0)	18.3	$1524_{\pm 23}$ ($1305_{\pm 15}$)
	1358510	1546.4 (8505.3)	171.8	$1307_{\pm 19}$ ($1118_{\pm 17}$)
Ens	12550	18.2 (99.8)	10.1	$2440_{\pm 29}$ ($2311_{\pm 26}$)
	704850	824.5 (4534.9)	458.1	$1230_{\pm 10}$ ($1046_{\pm 10}$)
UMix	12550	18.2 (99.8)	10.1	$2087_{\pm 36}$ ($1840_{\pm 35}$)
	704850	824.5 (4534.9)	458.1	$1502_{\pm 36}$ ($1286_{\pm 28}$)
gEns	2510 $\rightarrow$ 25100	3.6 (20.0)	0.4 $\rightarrow$ 4.0	$2727_{\pm 70}$ ($2542_{\pm 24}$)
	140970 $\rightarrow$ 1409700	164.9 (907.0)	18.3 $\rightarrow$ 183.2	$1348_{\pm 17}$ ($1283_{\pm 8}$)
gMoE	5214 $\rightarrow$ 29550	10.1 (65.3)	0.5 $\rightarrow$ 1.7	$2401_{\pm 61}$ ($2089_{\pm 28}$)
	27544 $\rightarrow$ 1485690	555.2 (3638.9)	26.2 $\rightarrow$ 97.2	$1448_{\pm 29}$ ($1222_{\pm 18}$)
FF	2446 $\rightarrow$ 32519	26.1 (222.3)	0.4 $\rightarrow$ 5.5	$2272_{\pm 26}$ ($1947_{\pm 44}$)
	140458 $\rightarrow$ 1646809	646.3 (7166.3)	18.3 $\rightarrow$ 138.9	$1450_{\pm 36}$ ($1189_{\pm 20}$)

Table 2: CIFAR 10 results with waiting time equal to 5. We use the same notations and colors than in Table 1.

time equal to 5 have the lowest area under the curve. Greedy methods using waiting time equal to 1 achieve lower error rate only during the very beginning of the stream. Second, we observe that bigger models (*SM* and *Ens*) not only generalize better but they are also statistically more efficient: the small *Ens* obtained almost 35% error rate by the end of its learning experience, which is worse than the error rate obtained by the large *Ens* just after having observed one tenth of the entire stream. The statistical efficiency of large models does not apply only to large transformers (Kaplan et al., 2020a), but also to fully connected (we obtained similar results on MNIST) and convolutional models.

Next, using the waiting time that yielded the lowest cumulative error rate, we compare all methods discussed in §4, focusing our discussion on Tab. 2 of CIFAR 10 as same conclusions apply to MNIST as well (see Tab. 1).

First, replay lowers the CER by a relative amount of about 10% at the cost of increasing the cumulative training flops by a factor of more than 5, which is rather substantial. Notice that retraining from scratch using memory replay, as reported here in parentheses, is nowadays the dominant approach to deal with sequential datasets.

Second, *Ens* works better than *UMix* for larger models, and vice versa. We surmise that ensembling may alleviate overfitting of large models, but coordinating the components of the ensemble like *UMix* does, is more effective in an underfitting regime (i.e with small models). *Ens* thus looks like a good method to train large architectures without suffering of the overfitting aspect and may be used when the complexity of the task is not known *a priori*.

Third, all growing approaches perform rather similarly, particularly when starting from larger backbones, although they strike slightly different trade-offs. For instance, *gMoE* is the most efficient at test time, while *FF* yields a lower error rate. Interestingly, none of the approaches that grow architectures currently manages to beat *Ens* in terms of error rate when starting from a large backbone, although they require substantially fewer flops at inference time. Finally, while methods derived from *SM* (for the same size of the initial backbone, see rows with the same color in the table) all manage to beat *SM*, it is also worth noting that for the same number of parameters *SM* is still the best performing method, unless there is overfitting. In particular, *Ens* with 12550 parameters achieves a CER of 2440 while *SM* with 11710 parameters obtains a CER of 2038 while requiring much less compute; same considerations apply also to the *gMoE* with 29550 parameters compared to *SM* with 31660 parameters. Therefore there is no single model striking a much better trade-off, and more advanced approaches do not outperform simpler methods like *Ens*.

The results on the large scale language modeling task reported in Tab. 3 show that bigger models perform better (the larger the number of parameters the lower the PPL for a given model class) and are also more statistically efficient (for instance the base *SM*_w1 attains 26.53 after seeing the whole stream, while the large *SM*_w1 obtains 22.47 just after seeing the first chunk of data), consistent

		Base model perplexity					*Large model perplexity*				
setting	# experts	$\|\theta\|$	t_0	t_1	t_2	t_3	$\|\theta\|$	t_0	t_1	t_2	t_3
*SM*_w1	4	65M	28.57	27.45	26.91	26.53	210M	22.47	21.62	20.84	20.54
*SM*_w3	8	91M	*	*	25.18		323M	*	*	19.29	
*SM*_w4	12	116M	*	*	*	24.41	436M	*	*	*	19.01
*Ens*_w1	4@2	130M	26.20	25.12	24.57	24.35	420M	20.32	19.55	19.14	18.92
	4@4	260M	25.03	24.03	23.45	23.29	840M	19.27	18.52	18.22	18.07
*gEns*_w1	4@1	65M	28.57				210M	22.47			
	4@2	130M		26.27			420M		20.25		
	4@3	195M			25.41		630M			19.49	
	4@4	260M				25.01	840M				19.18
*gMoE*_w1	4	65M	28.57				210M	22.47			
	6	78M		26.46			266M		21.22		
	8	91M			25.66		323M			20.39	
	12	116M				25.28	436M				20.15

Table 3: Large scale language modeling results. For *Ens* and *gEns*, 4@3 means 3 components in the ensemble, each of which has 4 experts per block, for instance.

with recent related work (Kaplan et al., 2020b; Li et al., 2020a). We also observe that *Ens* is a strong performer, with *Ens*_w1 and *gEns*_w1 models dominating *SM* models in all settings. Surprisingly, ensembles trained on distinct data chunks (*gEns*_w1; t_1 or t_3) perform no better than ensembles trained on a single data chunk (*Ens*_w1; t_0). For instance, among Base 2-model ensembles (4@2), *Ens*_w1 achieves a perplexity of 26.20 using a single data chunk (t_0), while *gEns*_w1 achieves a perplexity of 26.27 using models trained on each of the two data chunks (t_1). Finally, if test time inference is a concern, then *gMoE* is a preferable choice since its runtime is comparable to *SM*.

6 Conclusion and Perspectives

In this work we introduced the anytime learning at macroscale (ALMA) setting, which is an instance of anytime learning under the assumption that data is observed as a sequence of large batches. ALMA better mimics the learning scenarios faced by machine learning practitioners, who want to efficiently solve a task, but time to time they receive more data to train on. We introduced metrics that enable the assessment in terms of error rate, memory usage and compute throughout the entire learning experience. Equipped with these tools, we then evaluated several approaches on three different datasets, including large scale language modeling. We found that methods that update parameters at an intermediate rate tend to yield a better trade-off, and that bigger models tend to generalize better. In particular, models that grow capacity over time generalize better particularly when the initial model is smaller, and ensembling is a very strong baseline.

A cynical interpretation of our finding that bigger models generalize better, could take the reader to the conclusion that it can all be solved by starting with a big model. However, as data is added over time so is computation. It is often the case that researchers working on large-scale learning instantiate the biggest possible model to train on their task, but few months later they can manage to launch even bigger models thanks to compute and engineering advances. How can the larger model leverage what has been learned from the previously trained model? Is there a modeling choice that strikes a better trade-off than retraining from scratch? More generally, what are good approaches to extract information from a new batch of data? How can such information be integrated in an existing model? How can we accomplish never ending learning and adapt capacity over time while striking a good balance between generalization and memory/compute resources? While we do not provide a full answer to these questions, we do offer a framework to study them and several strong baseline approaches to compare against and build upon.

7 REPRODUCIBILITY STATEMENT

We have made several efforts to ensure that the results provided in the paper are fully reproducible. We first provide a clean codebase from which all the computer vision results in this paper are generated. In this codebase, one can find the exact hyperparameters used for each method in the provided configurations. We have attached a readme to the code in order to guide users running our code. For the LM experiments, as stated in the appendix we use the fairseq (Ott et al., 2019) and provide the required information to replicate our results.

REFERENCES

Yoshua Bengio, Nicholas Léonard, and Aaron C. Courville. Estimating or propagating gradients through stochastic neurons for conditional computation. *CoRR*, abs/1308.3432, 2013.

Léon Bottou. Online algorithms and stochastic approximations. In David Saad (ed.), *Online Learning and Neural Networks*. Cambridge University Press, Cambridge, UK, 1998.

Leo Breiman. Pasting small votes for classification in large databases and on-line machine learning. 36(85-–103), 1999.

Peter Bühlmann and Bin Yu. Analyzing bagging. *The annals of Statistics*, 30(4), 2002.

Ludovic Denoyer and Patrick Gallinari. Deep sequential neural networks. *EWRL*, 2015.

David Eigen, Ilya Sutskever, and Marc'Aurelio Ranzato. Learning factored representations in a deep mixture of experts. *ICLR*, 2014.

William Fedus, Barret Zoph, and Noam Shazeer. Switch transformers: Scaling to trillion parameter models with simple and efficient sparsity. *arXiv preprint arXiv:2101.03961*, 2021.

Chrisantha Fernando, Dylan Banarse, Charles Blundell, Yori Zwols, David Ha, Andrei A. Rusu, Alexander Pritzel, and Daan Wierstra. Pathnet: Evolution channels gradient descent in super neural networks. *CoRR*, abs/1701.08734, 2017. URL `http://arxiv.org/abs/1701.08734`.

Jürgen Fritsch, Michael Finke, and Alex Waibel. Adaptively growing hierarchical mixtures of experts. In *Advances in Neural Information Processing Systems*, 1996.

John J. Grefenstette and Connie Loggia Ramsey. Approach to anytime learning. In *Proceedings of the Ninth International Conference on Machine Learning*, 1992.

Michael I. Jordan and Robert A. Jacobs. Hierarchical mixtures of experts and the em algorithm. *Neural Computation*, 6:181–214, 1994.

Jared Kaplan, Sam McCandlish, Tom Henighan, Tom B. Brown, Benjamin Chess, Rewon Child, Scott Gray, Alec Radford, Jeffrey Wu, and Dario Amodei. Scaling laws for neural language models. *CoRR*, abs/2001.08361, 2020a.

Jared Kaplan, Sam McCandlish, Tom Henighan, Tom B Brown, Benjamin Chess, Rewon Child, Scott Gray, Alec Radford, Jeffrey Wu, and Dario Amodei. Scaling laws for neural language models. *arXiv preprint arXiv:2001.08361*, 2020b.

Diederik P Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization. In Yoshua Bengio and Yann LeCun (eds.), *3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7-9, 2015, Conference Track Proceedings*, 2015. URL `http://arxiv.org/abs/1412.6980`.

Alex Krizhevsky. Learning multiple layers of features from tiny images. *University of Toronto, technical report*, 2009.

Yann LeCun, Leon Bottou, and and Patrick Haffner Yoshua Bengio. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324, 1998.

Dmitry Lepikhin, HyoukJoong Lee, Yuanzhong Xu, Dehao Chen, Orhan Firat, Yanping Huang, Maxim Krikun, Noam Shazeer, and Zhifeng Chen. Gshard: Scaling giant models with conditional computation and automatic sharding. *CoRR*, abs/2006.16668, 2020.

Zhuohan Li, Eric Wallace, Sheng Shen, Kevin Lin, Kurt Keutzer, Dan Klein, and Joey Gonzalez. Train big, then compress: Rethinking model size for efficient training and inference of transformers. In *International Conference on Machine Learning*, pp. 5958–5968. PMLR, 2020a.

Zhuohan Li, Eric Wallace, Sheng Shen, Kevin Lin, Kurt Keutzer, Dan Klein, and Joey Gonzalez. Train big, then compress: Rethinking model size for efficient training and inference of transformers. In Hal Daumé III and Aarti Singh (eds.), *Proceedings of the 37th International Conference on Machine Learning*, volume 119 of *Proceedings of Machine Learning Research*, pp. 5958–5968. PMLR, 2020b.

Qiang Liu, Lemeng Wu, and Dilin Wang. Splitting steepest descent for growing neural architectures. In *Advances in Neural Information Processing Systems*, 2019a.

Yinhan Liu, Myle Ott, Naman Goyal, Jingfei Du, Mandar Joshi, Danqi Chen, Omer Levy, Mike Lewis, Luke Zettlemoyer, and Veselin Stoyanov. Roberta: A robustly optimized bert pretraining approach. *arXiv preprint arXiv:1907.11692*, 2019b.

Zhengying Liu, Zhen Xu, Shangeth Rajaa, Meysam Madadi, Julio C. S. Jacques Junior, Sergio Escalera, Adrien Pavao, Sebastien Treguer, Wei-Wei Tu, and Isabelle Guyon. Towards automated deep learning: Analysis of the autodl challenge series 2019. In Hugo Jair Escalante and Raia Hadsell (eds.), *Proceedings of the NeurIPS 2019 Competition and Demonstration Track*, volume 123 of *Proceedings of Machine Learning Research*, pp. 242–252. PMLR, 2020.

Myle Ott, Sergey Edunov, Alexei Baevski, Angela Fan, Sam Gross, Nathan Ng, David Grangier, and Michael Auli. fairseq: A fast, extensible toolkit for sequence modeling. *arXiv preprint arXiv:1904.01038*, 2019.

Sinno Jialin Pan and Qiang Yang. A survey on transfer learning. *TKDE*, 2010.

Alec Radford, Jeffrey Wu, Rewon Child, David Luan, Dario Amodei, and Ilya Sutskever. Language models are unsupervised multitask learners. *OpenAI blog*, 1(8):9, 2019.

Connie Loggia Ramsey and John J. Grefenstette. Case-based anytime learning. In *AAAI Technical Report WS-94-01*, 1994.

Mark B. Ring. *Continual Learning in Reinforcement Environments*. PhD thesis, University of Texas at Austin, Austin, Texas 78712, 1994.

Mark B Ring. CHILD: A first step towards continual learning. *Machine Learning*, 1997.

Rico Sennrich, Barry Haddow, and Alexandra Birch. Neural machine translation of rare words with subword units. In *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 1715–1725, 2016.

Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition. In *International Conference on Learning Representations*, 2015.

Sebastian Thrun. A lifelong learning perspective for mobile robot control. *Proceedings of the IEEE/RSJ/GI Conference on Intelligent Robots and Systems*, 1994.

Sebastian Thrun. Lifelong learning algorithms. In *Learning to learn*. Springer, 1998.

Vladimir Vapnik. *Statistical learning theory*. Wiley New York, 1998.

Tom Veniat, Ludovic Denoyer, and MarcAurelio Ranzato. Efficient continual learning with modular networks and task-driven priors. In *International Conference on Learning Representations*, 2021.

Yu-Xiong Wang, Deva Ramanan, and Martial Hebert. Growing a brain: Fine-tuning by increasing model capacity. In *IEEE Conference on Computer Vision and Pattern Recognition*, 2017.

Steve Waterhouse and Anthony Robinson. Constructive algorithms for hierarchical mixtures of experts. In *Advances in Neural Information Processing Systems*, 1995.

Wei Wen, Feng Yan, and Hai Li. Autogrow: Automatic layer growing in deep convolutional networks. In *Proceedings of the 26th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, 2020.

Lemeng Wu, Bo Liu, Peter Stone, and Qiang Liu. Firefly neural architecture descent: a general approach for growing neural networks. In *Advances in Neural Information Processing Systems*, 2020.

Jaehong Yoon, Eunho Yang, Jeongtae Lee, and Sung Ju Hwang. Lifelong learning with dynamically expandable networks. In *6th International Conference on Learning Representations, ICLR 2018, Vancouver, BC, Canada, April 30 - May 3, 2018, Conference Track Proceedings*. OpenReview.net, 2018. URL `https://openreview.net/forum?id=Sk7KsfW0-`.

Matthew D Zeiler. Adadelta: an adaptive learning rate method. *arXiv preprint arXiv:1212.5701*, 2012.

Tao Zou, Xian Li, Xuan Liang, and Hansheng Wang. On the subbagging estimation for massive data. *arXiv preprint arXiv 2103.00631*, 2021.

APPENDIX

A GROWING MIXTURES OF EXPERTS

Growing Mixture of Experts (*gMoE*): A mixture of expert (*MoE*) is a sequence of non-linear functions, each of which is potentially a mixture of experts (omitting the dependence on parameters):

$$m(x) = f^l(f^{l-1}(\dots f^1(x)\dots)), \text{ with } f^i(z) = \sum_{j=1}^{k} g^i(j|z)h^i(z|j)$$

where g^i is the gating function at the i-th layer which outputs a categorical distribution over the number of experts, and $h^i(\cdot|j)$ is the j expert at layer i. The gating function can be "soft" in which case it outputs non-zero weights for each expert via a softmax, or "hard" version in which case only one expert is selected through a multinomial sampling (and learned through the straight-through estimator in this paper (Bengio et al., 2013)). At test time in the "hard" case, we select the expert with the largest probability. The interest of mixtures of experts is they have a high expressivity, and experts can be easily added to increase the capacity of the model. The *gMoE*model is the growing version where, at each stage as illustrated in Fig. 2, new experts are added at each layer – details about the precise expansion process are given in Appendix.

The key design considerations are: *when* to grow, *what* to grow and *how* to grow. Here, we will refer to our default setting which favors simplicity, unless otherwise specified.

A growth step is triggered at each stage, ensuring a linear growth over time. We grow by adding one expert at each layer, making sure that all experts within a layer have the same architecture albeit with different parameters. In order to grow, we look at which expert has associated the largest cumulative loss; we call such expert the *losing* expert. The cumulative loss is defined as the sum of the losses of examples on the validation set that have been routed through a particular expert; each expert has associated a cumulative loss value. The rationale is to identify at each layer the expert responsible for the largest contribution to the total loss.

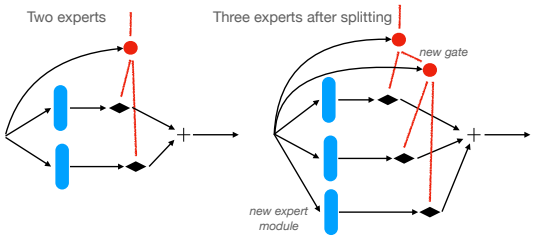

Figure 2: Illustration of a growth step in a tree structured mixture of experts. A network is composed of several layers like this. The blue squares are experts (e.g VGG layers). The red elements corresponds to the gatings which, given an input compute a score for each expert. When splitting an expert (right), the gating structure is updated by creating a child gate, and an additional expert is added to the mixture.

To avoid drop in the loss function and to keep its differentiability when splitting an expert, we propose a tree-based approach we the losing expert is split such expert into two experts with exactly the same parameters as illustrated in Fig. 2: Two children leaves are derived and we instantiate a new gating for the children which decides whether an input example routed to the old expert, should now go to the right or left expert child. The parameters of the new gate are initialized at random while the parameters of the new experts are exact copies of the ones of the losing expert that we split.

More formally, if s is the losing expert then the term $g^i(s|z)h^i(z|s)$ is replaced by:

$$\sum_{k=1}^{2} g^i(s|z)g^i(k|z,s)h^i(z|s,k) \tag{3}$$

where $g^i(k|z,s)$ is the newly introduced gate, and z is the input of the gating and experts.

Over time, the gating function learns to partition its input space into a binary tree (if we start from a single expert), and the gating value of an expert is the product of the gating probabilities on the path from root to the leaf expert. Both the gating tree structure and the particular initialization scheme guarantee that the growth step is smooth and fully differentiable, in particular, the loss before and after the growth step is the same.

Algorithm 2 gMoE

k: number of mega-batches to aggregate
$\mathcal{D} = \emptyset$
function TRAIN($\mathcal{D}_i$, i)
    $\mathcal{D} \mathrel{+}= \mathcal{D}_i$
    if i mod k == 0 **then**
        Extract $\mathcal{D}^{\text{VAL}}$ and $\mathcal{D}^{\text{TR}}$ from $\mathcal{D}$
        while m is not converged: **do**
            $(x, y) \sim \mathcal{D}^{\text{TR}}$ ▷ In practice, sample mini-batches.
            m.update(x, y)
        $\mathcal{D} = \emptyset$
        m.grow($\mathcal{D}^{\text{VAL}}$) ▷ Growth step can be done at a different rate too.
function GROW($\mathcal{D}^{\text{VAL}}$)
    for each layer in the network **do**
        Let i be the losing expert on $\mathcal{D}^{\text{VAL}}$, i.e. the expert incurring the largest cumulative loss.
        Turn corresponding gating output in an internal node and derive 2 gate children
        Initialize the new experts by copying the parameters from the old parent expert.
        Initialize the new gating between the two siblings at random.

B HYPER-PARAMETER SETTINGS

B.1 COMPUTER VISION EXPERIMENTS

For each megabatch received, we keep 10% of the data to perform cross-validation. All experiments are run on a single 16GB Quadro GP100 GPU. We apply data normalization for each dataset considered. A training minibatch size of 128 is used. *UMix* and *Ens* models have $N = 5$ in all experiments. for *gEns*, we train one model $n = 1$ at every mega-batch, so the total number of models depends on the amount of mega-batches. For Firefly we use a growth rate of 0.25, meaning that at every growth phase, we add approximately a quarter of the initial number of parameters.

B.1.1 MNIST

Models are trained for 100 epochs, and we report results with soft gating. We use the AdaDelta (Zeiler (2012)) optimizer with default learning rate of 1. We use a MLP with 2 hidden layers of varying width (e.g. 4,8 or 32 neurons).

B.1.2 CIFAR-10

Models are trained for 200 epochs, as this was shown to be long enough to allow the model to converge with a learning rate of 0.01. We use Stochastic Gradient Descent with momentum value of 0.9 and weight decay of 1×10^{-4}. During training, we apply random horizontal flips and select random image crops with padding of 4 pixels. For the architecture, we use the same reduced VGG with batch normalization as prescribed in Wu et al. (2020). All layers are initialized with the same number of channels (e.g. 4, 8, or 32 channels). For the Firefly experiments, we keep all the Firefly-specific hyperparameters to the default values suggested in the author's public codebase. We make one exception to this, namely we adapt the growth ratio to result in linear (rather than exponential) growth.

B.2 LANGUAGE MODELING EXPERIMENTS

All the language models are trained using fairseq (Ott et al., 2019) with a maximum of eight 32GB GPUs (NVIDIA V100), optimized with Adam (Kingma & Ba, 2014) using $\beta_1 = 0.9$, $\beta_2 = 0.98$, ϵ =1e-8. The learning rate is warmed up over the first several hundred updates (between 500 and 4000) and then linearly decayed to 0 over the remaining updates, with a peak value tuned between 2e-4 and 5e-3. Models are trained up to 120,000 updates with local batch size of 8 sequences per GPU, with gradient accumulation as needed to achieve a total batch size of 192 sequences; each

sequence has 512 tokens. We fix the Switch Transformer balancing loss term to 0.01 and use a capacity factor of 1, following Fedus et al. (2021).

C ADDITIONAL COMPUTER VISION RESULTS

In this section we show the impact of several variants of our framework. Namely, we report results for (a) a varying number of mega-batches, (b) whether to use preemption or not, and (c) whether to initialize from scratch or simply finetuning when replay is performed.

C.1 CIFAR

In the following results, we vary the number of megabatches. Below you can find results for $MB = 20$

C.1.1 DIFFERENT MBS

Method	$\|\theta\|$	Cum. Train TFLOPS	Inference MFLOPS	CER
SM	2510	3.6 (20.0)	0.4	$2705.8_{\pm 132.5}$ ($2643.1_{\pm 39.2}$)
	11710	15.0 (82.5)	1.7	$2038.2_{\pm 32.2}$ ($1842.3_{\pm 51.0}$)
	31660	38.6 (212.5)	4.3	$1763.5_{\pm 42.7}$ ($1560.6_{\pm 6.9}$)
	140970	164.9 (907.0)	18.3	$1524.3_{\pm 23.5}$ ($1305.0_{\pm 14.9}$)
	1358510	1546.4 (8505.3)	171.8	$1307.3_{\pm 19.1}$ ($1118.4_{\pm 17.0}$)
Ens	12550	18.2 (99.8)	10.1	$2440.3_{\pm 29.3}$ ($2311.1_{\pm 26.0}$)
	704850	824.5 (4534.9)	458.1	$1230.2_{\pm 9.5}$ ($1045.9_{\pm 9.8}$)
UMix	12550	18.2 (99.8)	10.1	$2087.0_{\pm 36.1}$ ($1840.3_{\pm 34.8}$)
	704850	824.5 (4534.9)	458.1	$1502.3_{\pm 36.0}$ ($1286.4_{\pm 27.7}$)
gEns	2510 $\rightarrow$ 25100	3.6 (20.0)	0.4 $\rightarrow$ 4.0	$2726.9_{\pm 70.3}$ ($2542.1_{\pm 24.1}$)
	140970 $\rightarrow$ 1409700	164.9 (907.0)	18.3 $\rightarrow$ 183.2	$1348.3_{\pm 16.8}$ ($1282.9_{\pm 8.5}$)
gMoE	5214 $\rightarrow$ 29550	10.1 (65.3)	0.5 $\rightarrow$ 1.7	$2400.6_{\pm 61.1}$ ($2089.3_{\pm 28.5}$)
	275442 $\rightarrow$ 1485690	555.2 (3638.9)	26.2 $\rightarrow$ 97.2	$1448.1_{\pm 29.3}$ ($1222.4_{\pm 17.5}$)
FF	2446 $\rightarrow$ 32519	26.1 (222.3)	0.4 $\rightarrow$ 5.5	$2272.5_{\pm 26.5}$ ($1946.9_{\pm 44.0}$)
	140458 $\rightarrow$ 1646809	646.3 (7166.3)	18.3 $\rightarrow$ 138.9	$1449.7_{\pm 35.7}$ ($1189.3_{\pm 19.6}$)

Table 4: CIFAR-10 $MB = 10$ results in the paper

Method	$\|\theta\|$	Cum. Train TFLOPS	Inference MFLOPS	CER
SM	2510	3.6 (38.1)	0.4	$2047.1_{\pm 28.0}$ ($1867.6_{\pm 35.6}$)
	140970	164.9 (1731.5)	18.3	$1171.2_{\pm 52.9}$ ($944.1_{\pm 16.7}$)
UMix	12550	18.2 (190.6)	10.1	$1615.1_{\pm 32.1}$ ($1300.2_{\pm 25.0}$)
	704850	824.5 (8657.5)	458.1	$1233.9_{\pm 23.9}$ ($876.4_{\pm 29.8}$)
gEns	2510 $\rightarrow$ 50200	3.6 (38.1)	0.4 $\rightarrow$ 8.1	$2000.5_{\pm 30.8}$ ($1833.9_{\pm 25.2}$)
	140970 $\rightarrow$ 2819400	164.9 (1731.5)	18.3 $\rightarrow$ 366.5	$1047.5_{\pm 47.1}$ ($949.6_{\pm 48.7}$)
FF	2446 $\rightarrow$ 22353	8.7 (315.4)	0.4 $\rightarrow$ 1.4	$1788.8_{\pm 57.9}$ ($1446.1_{\pm 42.9}$)
	140458 $\rightarrow$ 803761	300.9 (9646.7)	18.3 $\rightarrow$ 47.7	$1140.4_{\pm 38.8}$ ($799.2_{\pm 19.8}$)

Table 5: CIFAR-10 $MB = 20$ results

C.1.2 PREEMPTED RESULTS

We also consider the use of a patience term when training the model. When the validation accuracy has not improved over 25 consecutive epochs, we stop training for the given learning phase. As expected, we observe gains on compute efficiency, with a small loss in performance.

Method	$\|\theta\|$	Cum. Train TFLOPS	Inference MFLOPS	CER
SM	2510	1.2 (6.8)	0.4	$2844.6_{\pm 36.5}$ ($2788.2_{\pm 76.9}$)
	11710	4.4 (25.9)	1.7	$2223.0_{\pm 41.9}$ ($1929.3_{\pm 43.3}$)
	31660	11.2 (64.1)	4.3	$1898.6_{\pm 33.6}$ ($1615.8_{\pm 23.5}$)
	140970	39.9 (321.2)	18.3	$1574.6_{\pm 36.3}$ ($1374.9_{\pm 25.8}$)
	1358510	416.4 (2699.7)	171.8	$1366.1_{\pm 28.6}$ ($1158.1_{\pm 17.5}$)
Ens	12550	10.7 (61.9)	10.1	$2482.0_{\pm 37.8}$ ($2334.6_{\pm 18.2}$)
	704850	330.0 (3192.1)	458.1	$1238.7_{\pm 13.2}$ ($1048.5_{\pm 8.0}$)
UMix	12550	5.2 (35.5)	10.1	$2379.7_{\pm 37.4}$ ($1998.9_{\pm 33.9}$)
	704850	209.4 (1617.3)	458.1	$1675.9_{\pm 57.8}$ ($1376.4_{\pm 51.5}$)
gEns	2510 $\rightarrow$ 25100	1.0 (5.6)	0.4 $\rightarrow$ 4.0	$3048.6_{\pm 171.0}$ ($2688.4_{\pm 36.0}$)
	140970 $\rightarrow$ 1409700	42.8 (302.4)	18.3 $\rightarrow$ 183.2	$1432.8_{\pm 33.7}$ ($1325.2_{\pm 19.9}$)
gMoE	5214 $\rightarrow$ 29550	3.0 (25.6)	0.5 $\rightarrow$ 1.7	$2631.9_{\pm 92.5}$ ($2258.0_{\pm 46.7}$)
	275442 $\rightarrow$ 1485690	107.7 (1276.5)	26.2 $\rightarrow$ 97.2	$1562.7_{\pm 38.0}$ ($1298.4_{\pm 30.9}$)
FF	2446 $\rightarrow$ 35843	4.5 (100.3)	0.4 $\rightarrow$ 3.1	$2570.3_{\pm 119.2}$ ($2022.9_{\pm 85.4}$)
	140458 $\rightarrow$ 1622818	148.1 (2819.9)	18.3 $\rightarrow$ 134.2	$1537.9_{\pm 36.5}$ ($1253.3_{\pm 43.0}$)

Table 6: CIFAR-10 $MB = 10$ results, with preemption

Method	$\|\theta\|$	Cum. Train TFLOPS	Inference MFLOPS	CER
SM	2510	0.8 (9.9)	0.4	$2404.0_{\pm 71.1}$ ($2002.3_{\pm 28.0}$)
	140970	32.9 (431.7)	18.3	$1242.7_{\pm 43.2}$ ($965.8_{\pm 17.6}$)
Ens	12550	9.3 (121.2)	10.1	$1828.3_{\pm 20.4}$ ($1646.8_{\pm 14.5}$)
	704850	257.1 (4788.2)	458.1	$962.1_{\pm 20.6}$ ($724.3_{\pm 22.3}$)
UMix	12550	4.1 (53.3)	10.1	$1820.3_{\pm 55.2}$ ($1377.1_{\pm 51.3}$)
	704850	197.8 (2459.3)	458.1	$1334.5_{\pm 46.8}$ ($931.4_{\pm 21.2}$)
gEns	2510 $\rightarrow$ 50200	0.9 (10.0)	0.4 $\rightarrow$ 8.1	$2325.4_{\pm 93.3}$ ($1944.2_{\pm 60.8}$)
	140970 $\rightarrow$ 2819400	34.5 (482.0)	18.3 $\rightarrow$ 366.5	$1133.5_{\pm 29.7}$ ($932.7_{\pm 27.1}$)
FF	2446 $\rightarrow$ 17359	2.0 (71.6)	0.4 $\rightarrow$ 1.4	$2034.3_{\pm 44.0}$ ($1576.5_{\pm 97.9}$)
	140458 $\rightarrow$ 791701	57.9 (2312.4)	18.3 $\rightarrow$ 50.2	$1249.7_{\pm 65.1}$ ($886.9_{\pm 28.7}$)

Table 7: CIFAR-10 $MB = 20$ results, with preemption

C.1.3 Initializing from scratch

Below we show results, comparing the performance of re-training models from scratch on all the data seen so far vs simply finetuning the current model(s) on all the data. Main numbers are **finetuned models**, numbers in parentheses are **trained from scratch**.

Table 8: CIFAR-10 $MB = 10$ results with Replay. Numbers in () are models (re)initialized from scratch at the start of a new MB

C.2 MNIST

Method	$\|\theta\|$	Cum. Train TFLOPS	Inference MFLOPS	CER
SM	3210	0.02 (0.09)	0.003	$1054_{\pm 25}$ ($1062_{\pm 61}$)
	16330	0.09 (0.48)	0.016	$631_{\pm 10}$ ($616_{\pm 9}$)
	26506	0.14 (0.79)	0.026	$582_{\pm 12}$ ($554_{\pm 5}$)
	132127	0.71 (3.92)	0.132	$504_{\pm 8}$ ($485_{\pm 4}$)
	269322	1.45 (7.98)	0.269	$497_{\pm 8}$ ($474_{\pm 4}$)
Ens	16050	0.09 (0.47)	0.080	$870_{\pm 9}$ ($885_{\pm 18}$)
	132530	0.71 (3.93)	0.661	$517_{\pm 6}$ ($493_{\pm 5}$)
UMix	16050	0.09 (0.47)	0.080	$668_{\pm 17}$ ($646_{\pm 18}$)
	132530	0.71 (3.93)	0.661	$508_{\pm 10}$ ($486_{\pm 5}$)
gEns	3210 $\rightarrow$ 32100	0.02 (0.09)	0.003 $\rightarrow$ 0.032	$1020_{\pm 39}$ ($1027_{\pm 29}$)
	26506 $\rightarrow$ 265060	0.14 (0.79)	0.026 $\rightarrow$ 0.264	$551_{\pm 6}$ ($543_{\pm 7}$)
gMoE	7950 $\rightarrow$ 50610	0.16 (1.08)	0.008 $\rightarrow$ 0.050	$863_{\pm 17}$ ($766_{\pm 16}$)
	54318 $\rightarrow$ 304626	0.97 (6.55)	0.054 $\rightarrow$ 0.304	$573_{\pm 8}$ ($535_{\pm 5}$)

Table 9: MNIST Results from the paper, with $MB = 10$

Method	$\|\theta\|$	Cum. Train TFLOPS	Inference MFLOPS	CER
SM	16330	0.04 (0.19)	0.016	$637_{\pm 10}$ ($608_{\pm 10}$)
	132127	0.25 (1.84)	0.132	$507_{\pm 5}$ ($490_{\pm 5}$)
	269322	0.50 (3.11)	0.269	$491_{\pm 6}$ ($478_{\pm 5}$)
Ens	16050	0.06 (0.44)	0.080	$880_{\pm 14}$ ($877_{\pm 20}$)
	132530	0.37 (3.05)	0.661	$521_{\pm 4}$ ($496_{\pm 5}$)
UMix	16050	0.04 (0.19)	0.080	$687_{\pm 18}$ ($636_{\pm 21}$)
	132530	0.25 (1.85)	0.661	$507_{\pm 7}$ ($488_{\pm 4}$)
gMoE	7950 $\rightarrow$ 50610	0.06 (0.44)	0.008 $\rightarrow$ 0.050	$849_{\pm 28}$ ($829_{\pm 38}$)
	54318 $\rightarrow$ 304626	0.33 (2.71)	0.054 $\rightarrow$ 0.304	$572_{\pm 11}$ ($534_{\pm 6}$)

Table 10: MNIST $MB = 10$ using preemption with patience value of 25 epochs

Method	$\|\theta\|$	Cum. Train TFLOPS	Inference MFLOPS	CER
SM	16330	0.48 (0.48)	0.016	$615.6_{\pm 9.0}$ ($591.0_{\pm 8.7}$)
	132127	3.92 (3.92)	0.132	$485.5_{\pm 3.6}$ ($481.2_{\pm 3.3}$)
	269322	7.98 (7.98)	0.269	$474.0_{\pm 3.7}$ ($473.3_{\pm 5.8}$)
Ens	16050	0.47 (0.47)	0.080	$885.4_{\pm 18.4}$ ($881.6_{\pm 23.3}$)
	132530	3.93 (3.93)	0.661	$493.1_{\pm 4.9}$ ($495.6_{\pm 2.3}$)
UMix	16050	0.47 (0.47)	0.080	$645.9_{\pm 18.1}$ ($617.1_{\pm 14.6}$)
	132530	3.93 (3.93)	0.661	$485.6_{\pm 5.0}$ ($480.6_{\pm 4.2}$)
gMoE	7950 $\rightarrow$ 50610	1.08 (1.08)	0.008 $\rightarrow$ 0.050	$765.6_{\pm 15.9}$ ($778.6_{\pm 16.0}$)
	54318 $\rightarrow$ 304626	6.55 (6.55)	0.054 $\rightarrow$ 0.304	$535.4_{\pm 5.2}$ ($543.0_{\pm 7.7}$)

Table 11: MNIST $MB = 10$ results with replay, numbers in parentheses are models initialized from scratch at the start of every MB