ON ANYTIME LEARNING AT MACROSCALE

Lucas Caccia
McGill University, Mila
Facebook AI Research

Jing Xu
Facebook AI Research

Myle Ott
Facebook AI Research

Marc'Aurelio Ranzato[†*]
Facebook AI Research

Ludovic Denoyer[‡*]
Facebook AI Research

ABSTRACT

In many practical applications of machine learning data arrives sequentially over time in large chunks. Practitioners have then to decide how to allocate their computational budget in order to obtain the best performance at any point in time. Online learning theory for convex optimization suggests that the best strategy is to use data as soon as it arrives. However, this might not be the best strategy when using deep non-linear networks, particularly when these perform multiple passes over each chunk of data rendering the overall distribution non i.i.d.. In this paper, we formalize this learning setting in the simplest scenario in which each data chunk is drawn from the same underlying distribution, and make a first attempt at empirically answering the following questions: How long should the learner wait before training on the newly arrived chunks? What architecture should the learner adopt? Should the learner increase capacity over time as more data is observed? We probe this learning setting using convolutional neural networks trained on classic computer vision benchmarks as well as a large transformer model trained on a large-scale language modeling task. Code is available at `www.github.com/facebookresearch/ALMA`.

1 INTRODUCTION

In many practical applications of machine learning, data is not static but arrives sequentially in large chunks (or mega-batches). For instance, deployed language modeling systems need to be updated every few months to accommodate new snapshots of the Common Crawl dataset[1]. Similarly, visual object recognition systems need to be updated as new labeled data is gathered thanks to users interacting with the system. Moreover, as computing clusters are equipped with more memory and compute, machine learning practitioners would like to train bigger and bigger models on the ever increasing amount of data, since bigger models are often more accurate. In this setting, they face a dilemma: How to maximize the performance of the system at any point in time while satisfying a certain computational budget?

This question has certainly been studied before, most notably in the online learning literature (Cesa-Bianchi & Lugosi, 2006). For instance, in a contextual bandit setting the learner observes one example at the time and receives a reward after making a prediction. Of course, this can be extended to the case where the input is not just a single example but a set of examples (hereinafter referred to as mega-batch).

While prior works on online learning set a sound theoretical framework, there are some subtle issues that make it not quite applicable to the practical setting described above. First, computation is seldom explicitly taken into account, while in practice algorithms that are too computationally intensive cannot be considered at scale. Second, the vast majority of these works assumes linearity of predictors and convexity of optimization problems, whereby the order of examples does not change the optimum solution. Instead, in many practical applications (like language modeling) we are interested in using deep neural networks which are highly non-linear and which map to non-convex optimization problems. The lack of linearity hinders theoretical analysis, and it has profound practical implications. For instance, according to online learning theory the best case scenario is achieved when there are no delays (Joulani et al., 2016; Flaspohler et al., 2021), meaning that examples and their error signal are best to be consumed right away without any staleness in the model parameters. To use the language of the practitioner training a language model, this means that
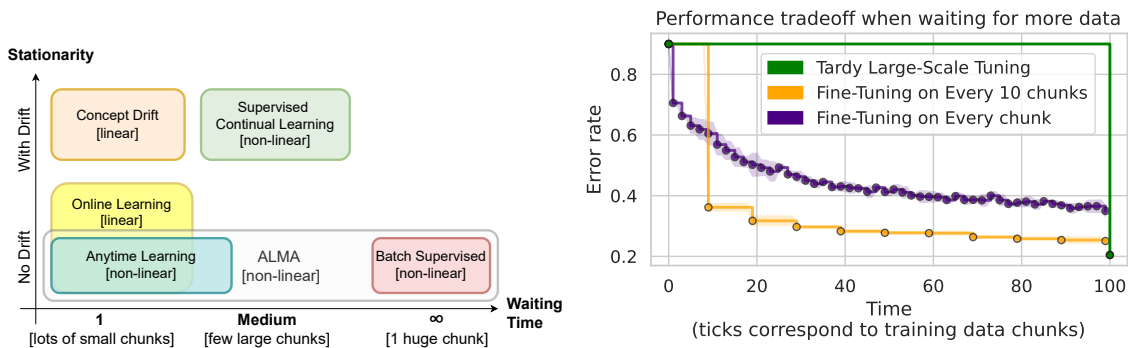

Figure 1: Left: ALMA compared to other learning frameworks. In ALMA, mega-batches of data are drawn from the same distribution (no drift) and arrive sequentially, but the learner can decide how long to wait before training on them. In the limit, if the learner waits till the end of the stream then learning reduces to standard batch supervised learning. Right: Examples of CIFAR 10 learning curves varying how long to wait before updating the model. Waiting for a small number of mega-batches before updating the parameters results in lower anytime error rate (smaller area under the learning curve).

according to convex online learning the best strategy is to train one mega-batch at the time. This however might not be a good strategy.

Consider what would happen if the deep neural network does multiple passes over each mega-batch before processing the next, and compare its performance to the one of a learner that waits for all the mega-batches to arrive before shuffling all data and applying the same stochastic gradient descent optimization algorithm as shown on the right part of Fig. 1. The latter setting is the standard procedure used in supervised learning (green curve): the learning algorithm optimizes a fixed objective (i.e the empirical risk over the entire training dataset) that is known to produce good predictors. While this predictor obtains the best final performance, it also attains the worst anytime performance since its predictions were random throughout the learning experience. In the former setting, by updating after each new mega-batch (purple curve), we can expect to maintain a good predictor all along the training experience, overcoming the problem described previously. However in this case, the learner is facing a changing learning objective, since each new mega-batch defines a slightly different empirical risk (Jothimurugesan et al., 2018). While we can expect this effect to be negligible when using linear models which eventually will converge to the same global optimum when all mega-batches are available, this is not the case when using non-linear predictors like deep neural networks. In that case, the sequence of optimization problems generated by the sequence of mega-batches may lead the learner to a completely different (local) optimum than the supervised learning setting, and thus to a completely different predictor. There is thus an open question about how different models behave when performing sequential learning over a stream of mega-batches.

In this paper, we empirically analyze several deep learning models (§4) under the assumption that data comes as a sequence of mega-batches, all drawn from the same distribution for simplicity. Since we are interested in models that attain good performance at any point in time and since we evaluate only after learning on each mega-batch but not during the learning of each individual mega-batch, we dub this learning setting Anytime Learning at MAcroscale (ALMA) (§3).

Through extensive empirical analysis (§5) we provide supporting evidence that waiting for a few mega-batches before updating the model is often the best strategy, although how long to wait depends on several factors such as the time horizon and model size relative to the amount of data in each mega-batch. Second, bigger models are more statistically efficient and generalize better. Third, none of the approaches we tried for growing the architecture were more effective than simpler alternatives which used fixed architectures, like ensembling. Overall, this study provides clear directions of future research, and also a platform for benchmarking new approaches against well tuned baselines (code available in supplementary material).

2 RELATED WORK

ALMA relates to several other learning frameworks as illustrated on the left of Figure 1. i) It shares the same assumptions of classical batch supervised learning (Vapnik, 1998) at the level of each mega-batch. However, it overall violates the assumptions of i.i.d. observations, because data points come in a stream of mega-batches and because the learner typically makes several passes over each mega-batch. Moreover, in ALMA the learner can choose how long to wait before training. In this sense, batch supervised learning can be thought of as an extreme case of ALMA (single mega-batch because learner waited till the end of the stream to train). ii) As mentioned in the previous section, ALMA relates to online learning (Bottou, 1998) because data comes sequentially and because in both cases we measure performance in terms of regret (although in §3.1 our cumulative error lacks a reference oracle since this is not known in

our setting). However, in ALMA we are also explicit about the computational budget used by the model and aim at striking a good trade-off between regret and computational cost. In our current work, we restrict ALMA to stationary distributions, while online learning is more general and encompasses also non-stationary distributions. Finally and most importantly, in ALMA we focus on non-linear predictors while typical literature on online learning considers linear predictors. iii) Similarly, ALMA relates to concept drift (Lu et al., 2018) because of the sequential nature of the observations. However, literature on concept drift often focuses on linear predictors. iv) ALMA can be seen as a degenerate case of supervised continual learning, where the task distribution is stationary. However, in supervised continual learning there is often a focus on attaining a predictor that represents the entire distribution of tasks by the end of learning, while in ALMA we measure cumulative error like in prequential learning. v) 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. vi) Finally, 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). However, unlike traditional anytime learning, in this work we are not interested in assessing the anytime learning ability at the level of each mega-batch, but only at a coarser granularity, at the level of the entire stream of mega-batches. Lastly, we note that while anytime learning operates in a similar setting as online learning (see Fig. 1), it is often used with non-linear predictors in a supervised learning setting.

To the best of our knowledge, the most relevant prior work is by Sahoo et al. (2018) which considers a setting similar to ours, except that their stream is composed by individual examples and in their setting there is no concept of waiting time nor revisiting data points several times. However, they also benchmark against methods that increase capacity over time, although their analysis was limited to fully connected networks.

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 $y \in \{1, \ldots, C\}$. For the sake of simplicity of exposition, in this work we restrict ourselves to classification problems, but similar arguments can be made for regression, for instance. The key 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 $\mathcal{D}_i$ in such a way to minimize its objective function. Since the data is observed as a stream of mega-batches, the learner cannot have access to future mega-batches, and cross-validation of model hyper-parameters can only be performed using a subset of the current mega-batch. In other words, the learner can only do one pass over the stream. However, the learner typically does multiple passes over the current mega-batch if this improves its generalization ability. In fact, the learner might make several passes over the current and some previous mega-batches, although replaying too much might eventually deplete its computational budget.

Either way, since the learner makes several passes over each mega-batch, the overall data distribution observed by the learner by the end of the stream is not i.i.d., even though mega-batches are drawn from the same underlying distribution $p(x, y)$ and samples drawn from each mega-batch are i.i.d.. For instance, in the limit case where each mega-batch consists of a single example from a set of n examples and a learner performing k passes over each mega-batch, the stream will consist of a sequence of examples (in a certain order) each replicated k times, which is different from drawing uniformly at random $k * n$ examples from the original set of n examples. This implies a trade-off between fitting the current data well versus generalizing well by the end of the stream.

In ALMA, the learner has an additional hyper-parameter compared to other learning frameworks: It can decide how long to wait before updating its parameters. We measure such waiting time in terms of number of consecutive mega-batches. For instance, a model with a waiting time equal to k, aggregates k consecutive mega-batches before updating its parameters. This will sacrifice a bit its performance during the waiting period, but might ultimately yield better generalization since the model can better shuffle the data and get closer to the ideal i.i.d. data distribution required by stochastic gradient descent optimization.

3.1 METRICS

We evaluate learners in the ALMA setting across three axes, namely: error rate, 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.

Algorithm 1 Training in the ALMA setting

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

---

We compute the error rate of model m at time t (after the arrival and potential update over 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:

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

where $m(x; \theta_t)$ is the model at time t equipped with parameters θ_t, $\mathcal{D}^{\text{Ts}}$ is the test set (common for all mega-batches in the stream), $|\mathcal{D}^{\text{Ts}}|$ 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 entire 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.

Notice that the above metrics are measured by the environment as training progresses, and will be used in our empirical assessment (§5). However, the learner does not have access to the test set. The learner has only access to the validation set of the current mega-batch, and can only use that to select its own hyper-parameters.

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). We also evaluate models in the setting where they can 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 and initialized from the parameters of the model trained on the previous mega-batch, unless otherwise specified.

Ensemble of Models (*Ens*):   The second approach is the simplest modular approach, consisting of an ensemble of N neural networks with the same architecture but different random initialization seed, each being trained independently on the same data. The output of the overall model at test time is the average probability distribution produced by each component[2]. 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, meaning that training and testing use different model predictors. *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 a fixed number of networks that are, at each stage, trained over the new chunk of data, *gEns* replaces this step by a growing step where k new neural networks are added. In our implementation, only these k neural networks are trained over the new data, while the other neural networks (trained on previous mega-batches) are kept fixed. Therefore, when starting with a single component and until the next growing step, the cost of training *gEns* is equal to *SM* for the same model architecture. Unless otherwise specified, we use $k = 1$ for the experiments in the paper.

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 when 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 gradually over time. This has a tree structured gating function where leaves correspond to experts. At each layer, we calculate each expert's contribution to the total loss by summing the losses of the examples routed through that expert. We then "split" the expert responsible for the largest contribution to the 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 further details.

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 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 second 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 third dataset, used for our large-scale language modeling

---

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

evaluation, is a portion of the collection of English language text introduced in Liu et al. (2019), 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 ($B = 100$ for CIFAR, $B = 500$ for MNIST and $B = 4$ 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 §6.

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 four model sizes. For methods with expanding architectures, we try different configurations of hyper-parameters controlling *when* to grow, and *how much* to grow. For simplicity, we limit expansion phases to occur in between megabatches. 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 various hidden units size, ranging from 4 to 64 (which we refer to as [small] and [large], 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*.

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) as in (Wu et al., 2020), where the number of intermediate feature maps is the same for each layer, ranging from 4 to 64. 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. While this approach performs slightly worse than the one described in §4.2, it is easier to implement at scale. 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 mega-batch sizes. During each mega-batch, we train all language models for exactly 120000 gradient steps (results in Fig. 5) unless otherwise specified (e.g. Tab. 1). This makes it easier to compare models for the same computational budget at the mega-batch level.

6   RESULTS

6.1   VISUAL RECOGNITION

Since conclusions are rather similar, we focus our analysis on the more challenging CIFAR 10 dataset, and report results also on MNIST in Appendix C.

Smallest waiting time might not be optimal: We begin our analysis in the setting without replay, shown in Fig. 2. We first observe that an intermediate waiting time (in this case equal to 10) strikes the best trade-off between Cumulative Error Rate (CER) and both training cost (left) and memory cost (right). As shown in Fig. 3-top, where the test error rate is plotted as a function of the number of mega-batches received, greedy methods using waiting time equal to 2 achieve a lower error rate only during the very beginning of the stream, but are outperformed later on. Tardy predictors waiting for all 100 mega-batches before training obtain the best final accuracy, but have no predictive capabilities throughout
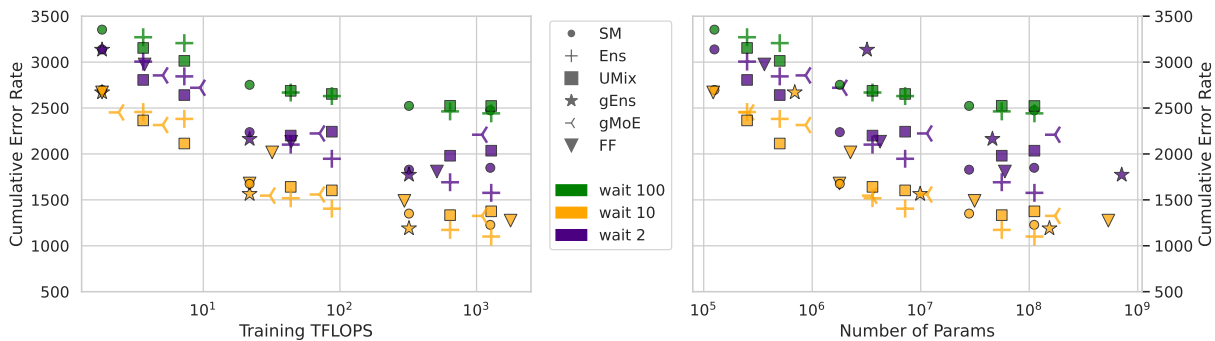
Figure 2: CIFAR 10 results: Cumulative error rate versus cumulative flops and number of parameters without replay. For the same model type we vary the size of the backbone architecture and the waiting time.
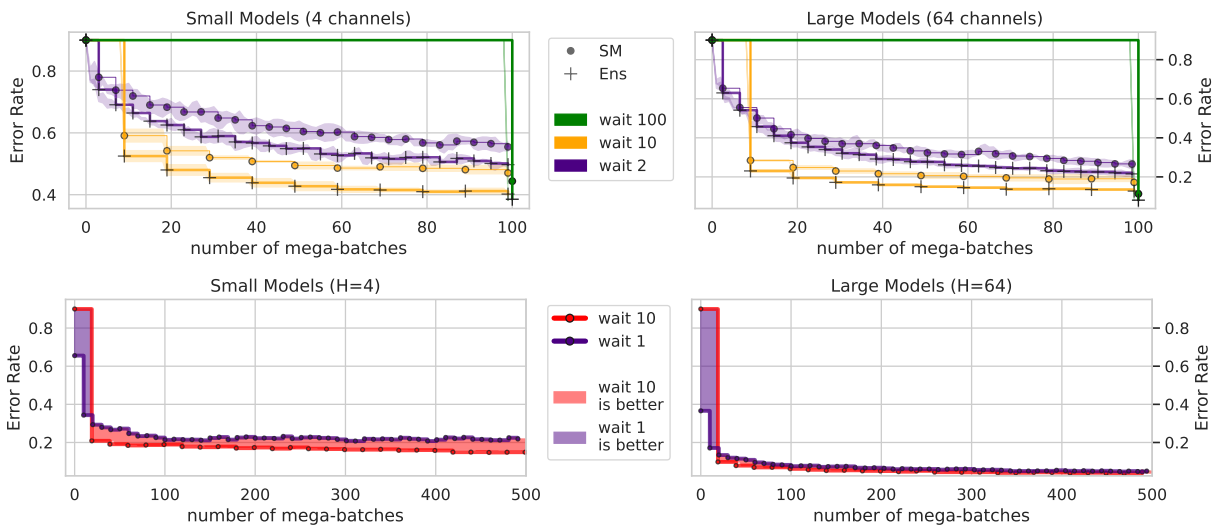

Figure 3: Error rate over time of small models (left) and large models (right) on CIFAR 10 (top) and MNIST (bottom).

the first 99 mega-batches. Instead, methods with an intermediate waiting time (shown in orange) can quickly deliver a reasonable predictor early in the stream, and obtain a final error rate that is very close to the lower bound obtained by tardy methods. Thus, a waiting time of 10 yields the lowest area under the curve (or CER) on CIFAR 10.

On MNIST however, an intermediate waiting time is best only for small models, as shown in Fig. 3-bottom. Very greedy models do not converge as well in this setting, which leads to a significant penalty in terms of CER. However, bigger networks converge very fast in just a few megabatches, making smaller waiting times more desirable. Therefore, the optimal waiting time depends on several factors such as the model size, the time horizon, how difficult the task is and how quickly the model learns. In such non-convex setting, it is certainly not necessarily true that learning on the data as soon as it becomes available attains always the best trade-off between error rate and compute.

Larger models are more statistically efficient: Second, we observe that bigger models (*SM* and *Ens*) not only generalize better but they are also statistically more efficient: on the small *Ens* obtained almost 40% error rate by the end of its learning experience (Fig. 3-top left), 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., 2020), but also to fully connected (we obtained similar results on MNIST, see Fig. 3-bottom) and convolutional models.

Growing does not improve: If we focus our attention on the three approaches with growing architectures, namely *gMoE*, *gEns*, and *FF*, we find that there is no clear winner among them. When comparing across a fixed computational budget (Fig. 2 left), *gEns* overall performs better than *gMoE* and *FF*. However, when we fix the memory budget instead (Fig. 2 right), *gEns* is, on average the worst performing method.

Next, we investigate the efficiency of growing, since in principle, we would expect that adapting model capacity with the amount of data should strike a better trade-off between accuracy and memory/compute usage. For a fixed computation

or memory budget, it is always better to **start with a larger model**, rather than growing it over time. Indeed, we find that on both graphs of Fig. 2, *SM* almost always outperforms *gMoE* and *FF*, a trend that is especially visible for higher budgets of TFLOPS and parameters. In other words, a *gMoE* or *FF* that starts small and finishes big will typically be outperformed by a *SM* model of average size.

Finally, *Ens* is more efficient than *gEns* in terms of memory, but vice versa in terms of training compute. However, should we look at the inference cost of both methods, we would find that *Ens* outperforms its growing counterpart, whose inference cost grows over time while it is fixed for *Ens*. Once again, the best strategy is to pick the largest fixed-capacity model for a given computational budget. Notice that these conclusions apply to the methods considered in this study, and improving approaches that dynamically adapt their architecture over time is clearly a worthwhile avenue of future research.

Operating point matters: We proceed by contrasting *UMix* and *Ens*, where the former averages predictions during training between different components, while the latter trains each component independently. In all our experiments when working with smaller models, *UMix* has a slight edge on both memory and compute fronts; however as the size of each component gets bigger the trend reverses, and *Ens* outperfoms *UMix*. We surmise that smaller models suffer the most from the inherent inefficiency of ensembling which forces each component to learn the same set of features. When capacity is limited, it is better to coordinate learning among the components instead. Overall, this finding highlights how conclusions about which model works best really depends on the operating point. Only when we consider the full spectrum of model sizes, can we conclude which approach works best.

***Ens* strikes the best trade-off:** More generally, ***Ens* is the best performing method for larger models** across all our experiments, including the language models reported in §6.2. This is a remarkable finding given the simplicty of the approach and how easy it is to parallelize their training process. Ensembling makes it very easy to increase model capacity early on, and it is so far the best way to utilize compute at scale, a possible indication of the inefficiency of training large models using alternative approaches, which highlights yet another worthwhile avenue of future research.

Replaying past mega-batches does not improve: We now consider the same approaches as before but with models trained on all megabatches seen so far. Therefore, at the very last training step, models are trained on the entire dataset (concatenation of all megabatches). In Fig. 4 we report the results when the waiting time is equal to 10. In all cases, replaying data gives better results at the expense of an increase in compute. Except for *gEns*, these gains are roughly the same for all methods, as all segments are parallel to each other. *gEns* gains less as the last component which is trained on the full dataset has disproportionate influence in the model average which includes components trained on fewer megabatches. However, this last component essentially coincides with *SM* trained on the full dataset. Hence the two methods converge to the same performance when using replay. We provide additional results with replay in Appendix C.2, which shows that there are benefits from replaying only at higher computational budgets where also the optimal waiting time reduces to 1.
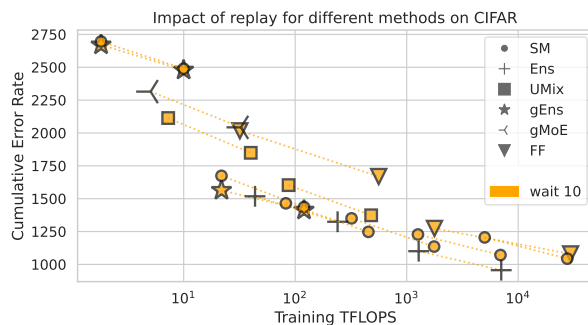

Figure 4: Impact of replay on the CIFAR-10 dataset with a wait time of 10. For each method we show a line from the result without replay (left) and with replay (right).

More importantly, we observe that **replay does not yield a significantly better trade-off** between CER and compute. For the same computational budget, methods using replay attain similar CER of methods that do not use replay. Other factors such as the size of the backbone architecture or the waiting time matter more.

6.2 LANGUAGE MODELING EXPERIMENTS

For the large-scale language modeling experiments, we consider two model sizes (base and large, see §5), with an inference cost per input of 42 and 126 GFLOPS, respectively. The number of experts is set to 4, 8 and 12 for *SM*, and it does not affect the inference cost since only one expert per input is selected regardless of the total number of experts. Due to the computational burden of these experiments (in total more than 200 GPU days), we limit our analysis to four mega-batches. Nevertheless, this scale (of model and data) and type of application are rather representative of a typical ALMA setting. Please refer to Tab. 2 in Appendix D for a comprehensive analysis, as here we will only highlight the major findings.
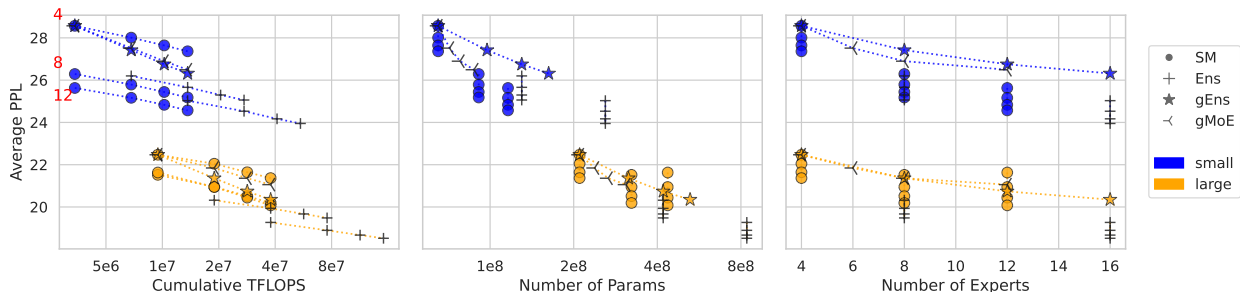
Figure 5: Language modeling trade-offs: average perplexity (PPL) versus cumulative compute, number of parameters and number of experts. Numbers in red refer to the number of experts in the corresponding *SM* runs.

Method	PPL $k = 3$, iid	PPL $k = 3$, seq.	PPL $k = 4$, iid	PPL $k = 4$, seq.
Small *Ens* 4@2	**24.30**	24.57	**24.13**	24.35
Big *Ens* 4@2	**18.04**	19.14	**17.88**	18.92

Table 1: Ablation on the effect of learning sequentially (seq.) as opposed to learning with fully i.i.d. data, for the same amount of data and compute. The model is an ensemble with 2 components each of which with 4 experts per block.

The main results are presented in Fig. 5. Each line is a trajectory with four points, one for each mega-batch in the stream, as we report average as opposed to cumulative perplexity. For a given model size and for a given computational budget, there are three *SM* models, one for each number of experts we consider, namely 4, 8 and 12.

Larger models are more efficient: In agreement with our results on computer vision tasks, we observe that bigger models tend to generalize better and are more sample efficient. For instance, the large model after a single mega-batch outperforms all base models, including base models after four mega-batches which have seen four times more data. This finding is consistent across all methods tried for this experiment.

Growing does not improve: Once again, there is no clear winner among growing methods. For larger models, *gEns* outperforms *gMoE* for the same compute, and perform similarly for base models. However, for all model sizes, *gMoE* is more memory efficient, therefore the optimal approach among them will depend on both compute and memory budget. More importantly, we observe that models with fixed capacity are more compute and memory efficient than models that grow over time. Looking at the average perplexity as a function of the number of experts, we see that methods which start with a small number of experts and later grow are outperformed by similar fixed architecture which have an intermediate number of experts. This highlights the importance of having more capacity at the start of training, rather than at the end.

Ensembles perform the best: Third, *Ens* thrives in the larger capacity setting. Looking at the orange markers in the graph, we see that for equal computation budget, *Ens* methods outperform all other methods, which is consistent with the computer vision results. In the base setting instead, versions of *SM* (see the lowest blue points) strike a better tradeoff in both compute and memory.

Learning sequentially is harder: We argued initially that once the learner makes several passes over each megabatch, the data distribution cannot be considered i.i.d. anymore, relative to the empirical distribution of the union of all megabatches. It is however unclear how much this issue has a practical impact in the performance of the model. In order to assess this we run one last experiment using our best performing approach, namely *Ens*. We compare a model trained on k mega-batches sequentially with the same model trained all at once on the aggregation of the same k mega-batches. Since both approaches have the same computation budget, the same architecture and are fed with the same data, we can disentangle the effect of the non-i.i.d nature of the data in ALMA. The results shown in Tab. 1 confirm that ALMA's sequential (seq.) training is indeed more challenging. Across all four configurations, models incur a drop in performance when compared to regular i.i.d training, and even more so when the model is larger. This gap offers another opportunity of future research on ways to make sequential training more effective when using deep non-linear neural networks.

7 Conclusions

In the abstract we promised the reader to provide an empircal answer to several questions:

1) <u>How long should the learner wait before training on the newly arrived mega-batches?</u> There is no single answer to this question. We have seen that on CIFAR 10 but also on MNIST when using smaller architectures and when using replay with smaller compute budgets, an intermediate waiting time strike the best trade-off. However, there is no known formula for deriving the waiting time, as it depends on several factors such as the time horizon, the initial performance of the model and how quickly a model learns, to name a few. The firm conclusion is that greedily updating the model as soon as data becomes available, as advocated by literature on convex online learning, might not always be the best strategy when using deep neural networks, In practice, also waiting too long, to the point that the learner does not even have time to perform a single pass over the aggregated mega-batches, might be suboptimal.

2) <u>What architecture should the learner adopt?</u> Our study indicates that, among all methods we tested, ensembling strikes the best trade-off in general. Ensembling is simple and easily parallelizable, and it offers a straightforward way to increase capacity. Starting off with a larger model, for instance via ensembling, is an excellent way to obtain good anytime performance.

3) <u>Should the learner increase capacity over time as more data is observed?</u> The answer is negative, currently. It is better to start off with the largest architecture fitting into memory and keeping that fixed. A cynical interpretation of this conclusion could make the reader believe that growing the architecture size should not be a topic of interest. However, as data is added over time so is computation and memory. It is often the case that researchers working on large-scale learning instantiate (rightly so) 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 to integrate it into an existing model? We believe these are great avenues of future research, and that our ALMA framework (learning and evaluation protocol, codebase, baselines) provides a good abstraction of the practical setting, and a sound tool to pursue such investigation.

8 Reproducibility Statement

We have made several efforts to ensure that the results provided in the paper are fully reproducible. We first provide an easy-to-use 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.

9 Acknowledgements

References

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

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

Nicolo Cesa-Bianchi and Gábor Lugosi. <u>Prediction, learning, and games</u>. Cambridge university press, 2006.

Ludovic Denoyer and Patrick Gallinari. Deep sequential neural networks. <u>EWRL</u>, 2015.

David Eigen, Ilya Sutskever, and Marc'Aurelio Ranzato. Learning factored representations in a deep mixture of experts. <u>ICLR</u>, 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.

Genevieve Flaspohler, Francesco Orabona, Judah Cohen, Soukayna Mouatadid, Miruna Oprescu, Paulo Orenstein, and Lester Mackey. Online learning with optimism and delay. In International Conference on Machine Learning, 2021.

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.

Ellango Jothimurugesan, Ashraf Tahmasbi, Phillip B. Gibbons, and Srikanta Tirthapura. Variance-reduced stochastic gradient descent on streaming data. In Neural Information Processing Systems, 2018.

Pooria Joulani, Andras Gyorgy, and Csaba Szepesvari. Delay-tolerant online convex optimization: Unified analysis and adaptive-gradient algorithms. In Association for the Advancement of Artificial Intelligence, 2016.

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, 2020.

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.

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, 2019.

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.

Jie Lu, Anjin Liu, Fan Dong, Feng Gu, Joao Gama, and Guangquan Zhang. Learning under concept drift: A review. IEEE Transactions on Knowledge and Data Engineering, 31(12):2346–2363, 2018.

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.

Doyen Sahoo, Quang Pham, Jing Lu, and Steven C.H. Hoi. Online deep learning: Learning deep neural networks on the fly. In International Joint Conferences on Artificial Intelligence Organization, 2018.

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.

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

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.

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

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. 6, new experts are added at each layer (Fritsch et al., 1996).

The key design considerations are: <u>when</u> to grow, <u>what</u> to grow and <u>how</u> 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 <u>losing</u> 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.
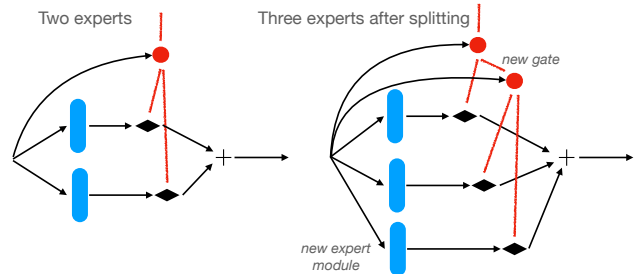

Figure 6: Illustration of a growth step in a tree structured mixture of experts. A network is composed of several layers like this. The blue squares are experts (e.g VGG layers). The red elements corresponds to the gatings which, given an input compute a score for each expert. When splitting an expert (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 where the losing expert is split into two experts with exactly the same parameters as illustrated in Fig. 6: 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.

If we consider each path in the MoE model to be a different model, then with L layer of k MoE components, there are k^L many possible paths through the MoE model, hence the number of paths grows exponentially with the number of layers. You can think of this as an ensemble with exponentially many components, but this is still tractable because components share parameters.

Algorithm 2 gMoE

1: k: number of mega-batches to aggregate
2: $\mathcal{D} = \emptyset$
3: **function** TRAIN($\mathcal{D}_i, i$)
4:     $\mathcal{D} += \mathcal{D}_i$
5:     **if** $i \bmod k == 0$ **then**
6:         Extract $\mathcal{D}^{\text{VAL}}$ and $\mathcal{D}^{\text{TR}}$ from $\mathcal{D}$
7:         **while** m is not converged: **do**
8:             $(x, y) \sim \mathcal{D}^{\text{TR}}$ ▷ In practice, sample mini-batches.
9:             m.update(x, y)
10:         $\mathcal{D} = \emptyset$
11:         m.grow($\mathcal{D}^{\text{VAL}}$)   ▷ Growth step can be done at a different rate too.
12: **function** GROW($\mathcal{D}^{\text{VAL}}$)
13:     **for** each layer in the network **do**
14:         Let i be the losing expert on $\mathcal{D}^{\text{VAL}}$, i.e. the expert incurring the largest cumulative loss.
15:         Turn corresponding gating output in an internal node and derive 2 gate children
16:         Initialize the new experts by copying the parameters from the old parent expert.
17:         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 also trained for 100 epochs 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, 2015) using $\beta_1 = 0.9$, $\beta_2 = 0.98$, $\epsilon =$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 DETAILED COMPUTER VISION RESULTS

C.1 MNIST

We show equivalent figures to the ones presented for CIFAR (e.g. Fig 2 and 3). We note for a given waiting time, different models rank similarly as in the CIFAR results. The main difference with the other computer vision dataset is on the optimal waiting time. As we saw in Fig. 3, on MNIST a predictor obtains good performance using very few mega-batches, making small waiting time competitive. Nevertheless, we do see that in terms of final error rate, a small waiting time underperforms, especially for small models.
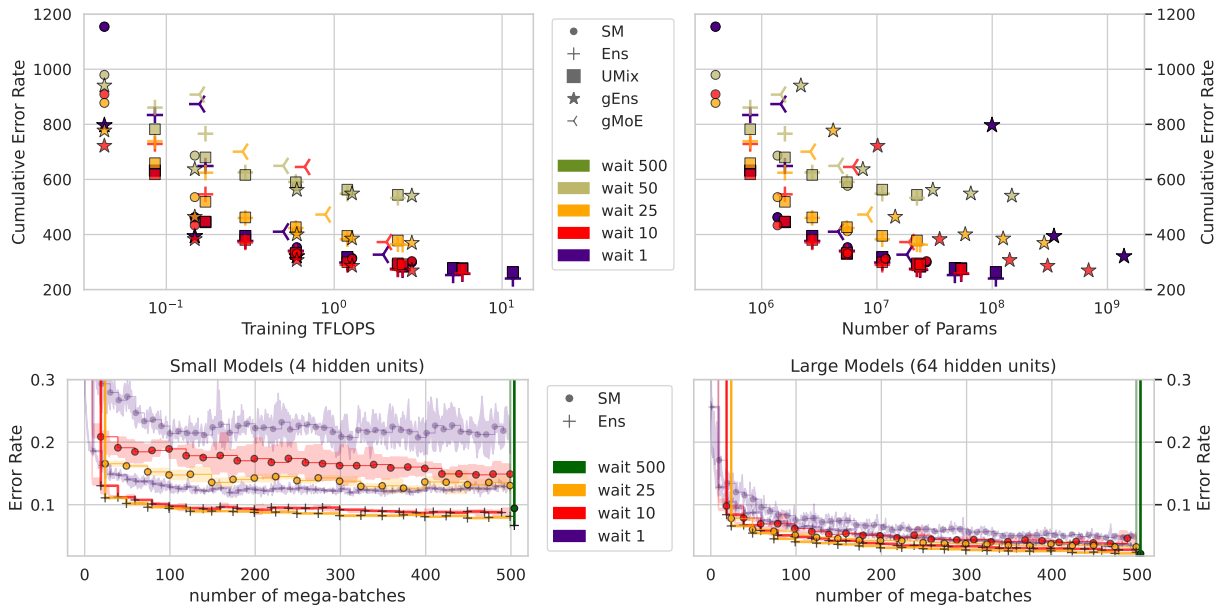

Figure 7: (top) Cumulative error rate versus cumulative flops and number of parameters without replay. For the same model type we vary the size of the backbone architecture and the waiting time. (bottom) Anytime Error Rate for the same methods on MNIST

15

C.2 CIFAR-10

For this dataset, we provide more results when using replay across a variety of methods and waiting times. We note in this setting, as the computational budget increases, the optimal waiting time decreases. This is because as more mega-batches is received, the training distribution gets closer to the ideal i.i.d scenario. It can therefore bypass the optimization issues faced when training for multiple iterations on a small dataset. Again, we emphasize that this is not the case when using a small waiting time and no replay.
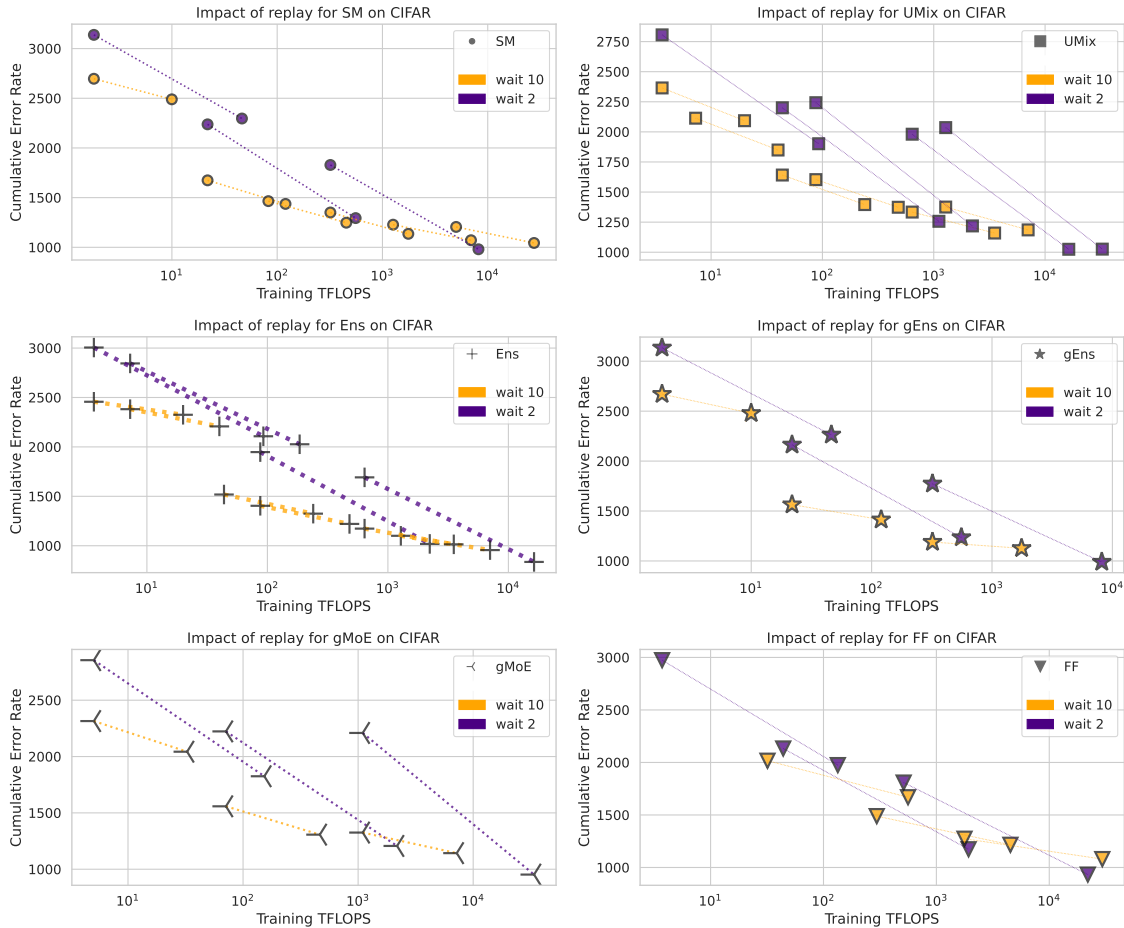

Figure 8: Impact of Replay across different methods and waiting times for CIFAR-10.

D FULL LM RESULTS

Below we present the full quantitative results for our language modeling experiments.

setting	# experts	$|\theta|$	Base model perplexity t_0	t_1	t_2	t_3	$|\theta|$	Large model perplexity 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*_w1	8	91M	26.29	25.29	24.74	24.40	323M	21.52	20.38	19.64	19.22
*SM*_w1	12	116M	25.63	24.70	24.17	23.78	436M	21.63	20.26	19.44	18.98
*SM*_w3	8	91M	*	*	25.18		323M	*	*	19.29	
*SM*_w3 (3x steps)	8	91M	*	*	24.21		323M	*	*	18.48	
*SM*_w4	12	116M	*	*	*	24.41	436M	*	*	*	19.01
*SM*_w4 (4x steps)	12	116M	*	*	*	22.87	436M	*	*	*	17.70
*Ens*_w1	4@2	130M	26.20	25.12	24.57	24.35	420M	20.32	19.55	19.14	18.92
*Ens*_w1	4@4	260M	25.03	24.03	23.45	23.29	840M	19.27	18.52	18.22	18.07
*Ens*_w3	4@2	130M	*	*	25.52		420M	*	*	19.11	
*Ens*_w3 (3x steps)	4@2	130M	*	*	24.30		420M	*	*	18.04	
*Ens*_w4	4@2	130M	*	*	*	25.49	420M	*	*	*	19.03
*Ens*_w4 (4x steps)	4@2	130M	*	*	*	24.13	420M	*	*	*	17.84
*gEns*_w1	4@1	65M	28.57				210M	22.47			
*gEns*_w1	4@2	130M		26.27			420M		20.25		
*gEns*_w1	4@3	195M			25.41		630M			19.49	
*gEns*_w1	4@4	260M				25.01	840M				19.18
*gMoE*_w1	4	65M	28.57				210M	22.47			
*gMoE*_w1	6	78M		26.46			266M		21.22		
*gMoE*_w1	8	91M			25.66		323M			20.39	
*gMoE*_w1	12	116M				25.28	436M				20.15

Table 2: 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.

E EXTENDED FIGURE 1

In this section, we add runs with replay to Fig. 1. We note that runs with replay are not directly comparable to runs without replay, because they have a higher computational cost. Indeed, runs that fine-tune every 10 chunks cost **4.5x** the cost of non-replay runs, and runs fine-tuning every chunk cost **122.5x**.
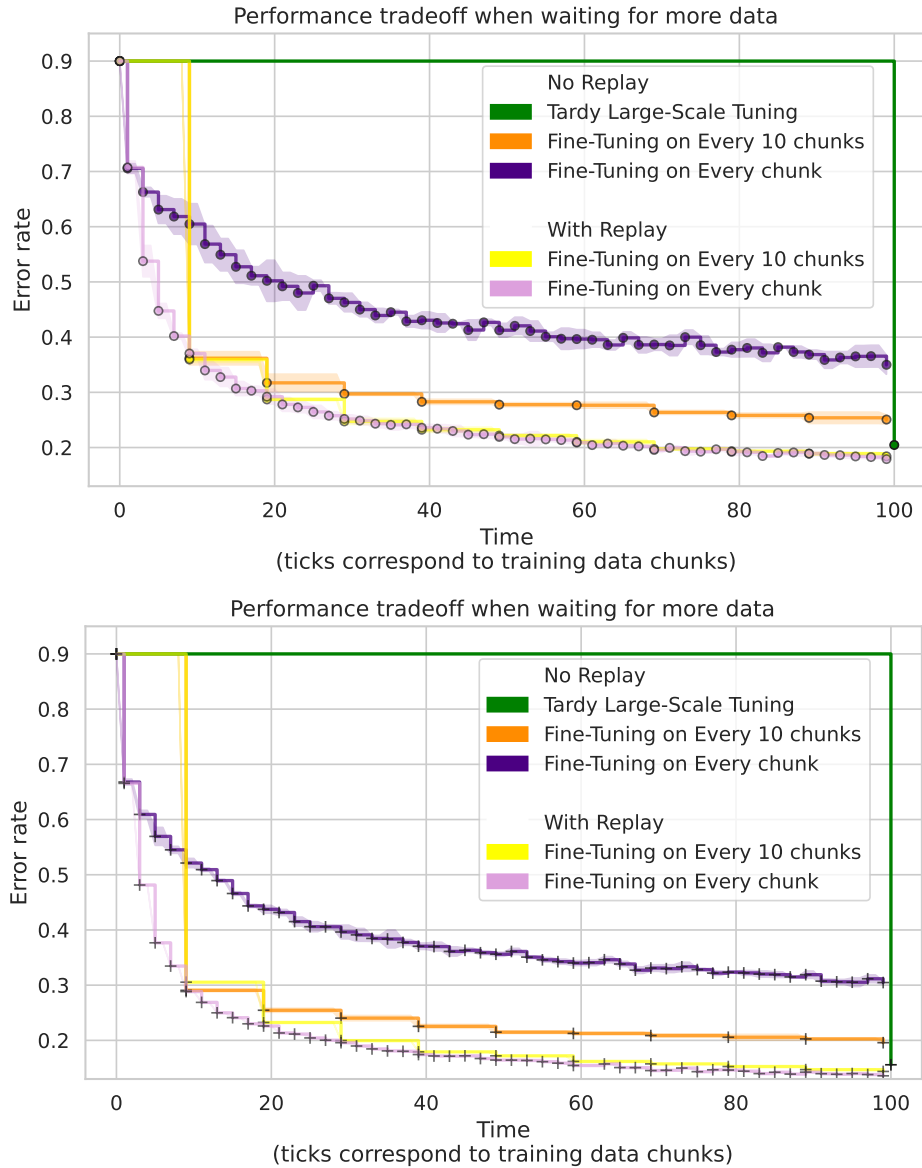

Figure 9: Fixed architecture runs (top) and growing ensemble runs (bottom)