Describing the trade-offs between training time, cost, and model performance for HuggingFace distributed training on Amazon SageMaker

Samantha Stuart, Inchara Bellavara Diwakar

KEY TAKEAWAYS

We present here the results of a benchmarking study investigating the impact of scaling compute instances and dataset sample size on HuggingFace model (distilBERT) training time, performance, and job cost in Amazon SageMaker. The instances investigated were the ml.p3.2xlarge (1GPU), ml.p3.8xlarge (4GPUs), ml.p3.16xlarge (8GPUs), and multi-node training with SageMaker Data Parallelism (2 Node ml.p3.16xlarge (16GPUs), 4 Node ml.p3.16xlarge (32GPUs)). The per GPU batch size was fixed during benchmarking (32), and we observed an exponential decrease in training job execution time (18h decreased to <1h) with scaling compute (number of GPUs in instance type) and global batch size. Training job cost ranged from $65-$100USD on average, with 2-Node data parallelism on 16xl instances yielding the best value with respect to execution time and job cost (2 hours, $80USD). However, scaling the global batch size (by means of scaling instance type and at a fixed per GPU batch size) led to a marked decrease in model performance, moreso for the training jobs with 100,000 samples than 1,000,000 samples. While scaling compute can significantly reduce model training times, it is advisable to anticipate how the global batch size, learning rate, and per device batch size may need to be adjusted to mitigate changes in performance with scaling compute. Pitfalls and best practices learned from executing this study are provided to help data scientists get started with running HuggingFace on SageMaker while scaling compute with SageMaker data parallel distributed training.

TABLE OF CONTENTS

INTRODUCTION

In 2021, a series of machine learning features were released on AWS to help customers fine-tune and deploy pretrained natural language models quickly with HuggingFace. Hugging Face Deep Learning Containers (DLCs), the HuggingFace framework in the SageMaker Python SDK, and built-in compatibility with SageMaker data parallelism together reduce undifferentiated heavy lifting for customers interested in fine-tuning ("training" used equivalently to fine-tuning herein) pretrained HuggingFace models to meet their business objectives.

Some important factors for customers to plan for at the outset of a HuggingFace project are the downstream impacts of model size. HuggingFace pretrained models range from having multi-millions to billions of model parameters to fine-tune on customer datasets. Selecting a right-sized compute instance for the job of fine-tuning a pretrained model can reduce training times on such large models from days to minutes with Amazon SageMaker.

However, anyone who is new to working with HuggingFace and SageMaker shares the same common questions:

- How much will SageMaker distributed data parallelism reduce my model training time?
- How will the cost of running training jobs change?
- How will model performance change, if at all, by using distributed data parallelism to reduce training time?
- How do all of these factors change again for different amounts of sample data used in the fine-tuning process?

We have performed a benchmarking study to help customers and data scientists dive deeper into these trade-offs between training time, model performance, and cost when fine-tuning HuggingFace models on SageMaker with distributed training. The results from this study will help customers and data scientists build intuition around points of diminishing returns when selecting compute resources to allocate for their machine learning projects with HuggingFace and SageMaker.

METHODS

Experimental Design

Model training time, model performance, and training job cost were the responses of interest in this benchmarking study. The independent variables during benchmarking were scaling compute resourcing (i.e. instance type, number of nodes) and dataset size (i.e. number of training samples) as described in Table 1: Experimental Controls All experiments with data parallelism utilized the built-in SageMaker Data Parallelism.

Table 1: Experimental Controls

Instance Type	*Num. GPUs*	*Data Parallel Enabled?*	*Per Device Train Batch Size*	*Global Batch Size*	*Learn Rate*	*Num. Total Steps*	*Num. Samples Tested*
ml.p3.2xlarge	1	No -1 Node	32	32	5E-05	84375	100,000 (30 epoch), 600,000 (5 epoch), 1,000,000 (3 epoch)
ml.p3.8xlarge	4	No -1 Node	32	128	5E-05	21094	100,000 (30 epoch), 600,000 (5 epoch), 1,000,000 (3 epoch)
ml.p3.16xlarge	8	Yes -1 Node	32	256	5E-05	10547	100,000 (30 epoch), 600,000 (5 epoch), 1,000,000 (3 epoch)
ml.p3.16xlarge	16	Yes - 2 Nodes	32	512	5E-05	5273	100,000 (30 epoch), 600,000 (5 epoch), 1,000,000 (3 epoch)
ml.p3.16xlarge	32	Yes - 4 Nodes	32	1024	5E-05	2637	100,000 (30 epoch), 600,000 (5 epoch), 1,000,000 (3 epoch)

Dataset & Task Selection

The HuggingFace Datasets Hub contains hundreds of natural language datasets suitable for benchmarking tasks, or exploring different HuggingFace models.

The open source amazon_polarity dataset was selected for this experiment from the HuggingFace Datasets Hub. The modeling task was binary sentiment classification. The amazon_polarity dataset is a labelled collection of over 35 million Amazon product reviews and their sentiment (positive, or negative). The review text column (labeled ‘content’) and corresponding binary sentiment labels were loaded into the benchmarking repository, subset into 100,000, 600,000, and 1,000,000 samples, and then loaded to S3 for benchmarking according to the experimental design.

Pretrained Model Selection

The pretrained "bidirectional encoder representations from transformers," or BERT, model represents the state of the art in natural language processing for text classification tasks. distilBERT is the smallest model in the BERT family in the HuggingFace Hub, with 66,955,010 fine-tunable parameters. distilBERT has 40% fewer parameters than BERT base, runs 60% faster, and preserves 95% of BERT's performance. These unique benefits make distilBERT a valuable model to benchmark, and an appropriate choice for use cases requiring fast iteration. To maximize time and cost savings, 'distilbert-base-uncased' was selected for this benchmarking project. "Uncased" means the model does not distinguish capitalized words as distinct from uncapitalized words.

Using 'distilbert-base-uncased' with the HuggingFace AutoModel class simplified the process of fine-tuning in SageMaker. A different AutoModel could be benchmarked equivalently in the future using the present repository by changing the environment variable HF_MODEL in .env to another AutoModel checkpoint name.

Selecting Number of Training Epochs

To facilitate reasonable comparisons across datasets with differing numbers of samples, a fixed number of steps was enforced for a given global batch size and dataset size by fixing the number of epochs. Steps refer to the points where model weight updates ("learning") occurs.

$$N_{steps\ per\ epoch} = \frac{N_{samples}}{Global\ Batch\ Size}$$

$$N_{epochs} = \frac{N_{steps\ total}}{N_{steps\ per\ epoch}}$$

For example, in the case of a 1GPU experiment, a number of total steps of 84375 was selected through guess-and-check. In all experiments, 90% of the samples were reserved for training, 10% for validation. Calculating the number of training epochs to enforce equivalent steps in all dataset sample sizes:

Table 2: Calculating Number of Training Epochs

	100,000 samples	*600,000 samples*	*1,000,000 samples*
Num. training epochs	$\frac{84375}{(\frac{100{,}000}{32}*0.9)}$ = 3 Epochs	$\frac{84375}{(\frac{600{,}000}{32}*0.9)}$ = 5 Epochs	$\frac{84375}{(\frac{1{,}000{,}000}{32}*0.9)}$ = 30 Epochs

Equivalent epoch enforcements (3, 5, and 30) were made as global batch size increased to allow dataset comparisons, and consequently the number of steps performed at a given compute configuration decreased as global batch size increased. Figure 1 shows the downstream impact of enforcing a consistent number of steps during model training across dataset sizes – the model training times measured for 100,000 sample runs (left) and 1,000,000 sample runs (right) are the same. Therefore, as a result of fixing the steps at a given dataset size and compute configuration,

differences in model performance with number of samples used in training can be quantified. This is examined later on in Figure 6.

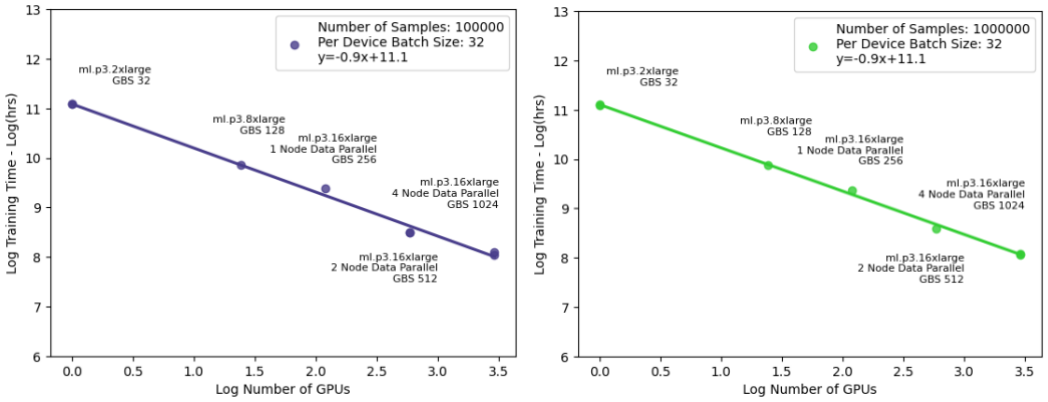

Figure 1: By fixing the number of steps at a given dataset size and global batch size in this experiment (methodology described in Table 2) the downstream training times for 100,000 sample runs (left) and 1,000,000 sample runs (right) are the same. Therefore, differences in model performance with respect to the number of samples used in training (Figure 6) can be benchmarked. Global batch sizes indicated above with GBS. A log transformation was applied on the recorded training time and number of GPUs to map the equivalent linear relationships.

SageMaker Data Parallelism

SakeMaker Data Parallelism is an optimized method of distributed training machine learning models using one line of additional code in the SageMaker Python SDK. Implementing distribution in a training job can decrease training time and increase throughput.
In data parallelism, each GPU available in the training job contains an identical copy of the model being trained. The training dataset is divided over each copy of the model, and the GPUs execute the model training in parallel, while periodically synchronizing model parameters using the Amazon AllReduce Algorithm. For a detailed background on SageMaker data parallelism, see its overview announcement from the AWS News Blog [1].

Another advantage of data parallelism is that with its implementation across multi-node compute clusters, a practitioner can fit larger global batch sizes in system memory. Increasing global batch size (and thereby decreasing the number of steps in training) is one mechanism of decreasing training time [2] and is the core focus of the present benchmarking study.

Generating Experimental Design

A baseline experiment with 15 runs was designed and evaluated for quality using a statistical software (JMP Custom Design Tool), and loaded into the present repository under data/raw. The field of experimental design is rigorous and comprehensive, and many alternative approaches to designing an experiment in the future could be pursued as desired by the reader.

The generated runs and the accompanying colour map on correlations are shown below in Figure 2. As evidenced by the colour map, the design is orthogonal, meaning that there is low (<0.2 Pearson correlation coefficient) to no confounding effects present in measuring the number of

nodes tested and the dataset size as proposed. The runs comprising the baseline design are indexed as Run 0 to Run 14.

	num_nodes	dataset_size
0	4	600000
1	4	100000
2	4	1000000
3	2	1000000
4	1	100000
5	4	600000
6	1	1000000
7	1	100000
8	4	1000000
9	1	600000
10	2	100000
11	4	100000
12	1	1000000
13	2	100000
14	2	600000

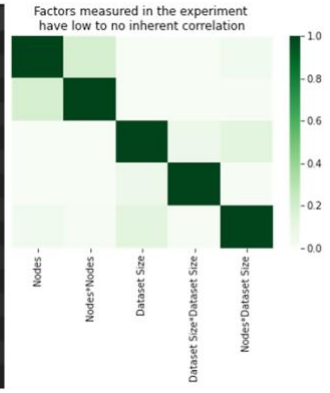

Figure 2: Baseline experimental design with 15 runs, and corresponding correlation colour map.

Several additional centrepoint runs were added to the experiment to test for additional curvature in the relationships between compute, dataset size and training responses of interest. Additional runs tested for additional variation in 1-node training jobs with different instance types (ml.p3.8xlarge, ml.p3.16xlarge), dataset sizes, and degrees of parallelism. The ultimate experimental design including centrepoint runs is displayed in Figure 3. Additional important details are also shared below such as EC2 instance type, number of GPUs, and hourly instance price.

	run_id	dataset_name	automodel_name	num_parameters_tuned	s3_bucket	per_device_train_batch_size	learning_rate	epochs	instance_type	num_gpus	global_batch_size	num_steps	hourly_price	volume_size	parallel_enabled	num_nodes	dataset_size
0	0	amazon_polarity	distilbert-base-uncased	66955010	hf-benchmarking-samstu	32	0.00005	5.0	ml.p3.16xlarge	32	1024	2637.0	28.152	30	True	4	600000
1	1	amazon_polarity	distilbert-base-uncased	66955010	hf-benchmarking-samstu	32	0.00005	30.0	ml.p3.16xlarge	32	1024	2637.0	28.152	30	True	4	100000
2	2	amazon_polarity	distilbert-base-uncased	66955010	hf-benchmarking-samstu	32	0.00005	3.0	ml.p3.16xlarge	32	1024	2637.0	28.152	30	True	4	1000000
3	3	amazon_polarity	distilbert-base-uncased	66955010	hf-benchmarking-samstu	32	0.00005	3.0	ml.p3.16xlarge	16	512	5273.0	28.152	30	True	2	1000000
4	4	amazon_polarity	distilbert-base-uncased	66955010	hf-benchmarking-samstu	32	0.00005	30.0	ml.p3.2xlarge	1	32	84375.0	3.825	1024	False	1	100000
5	5	amazon_polarity	distilbert-base-uncased	66955010	hf-benchmarking-samstu	32	0.00005	5.0	ml.p3.16xlarge	32	1024	2637.0	28.152	30	True	4	600000
6	6	amazon_polarity	distilbert-base-uncased	66955010	hf-benchmarking-samstu	32	0.00005	3.0	ml.p3.2xlarge	1	32	84375.0	3.825	1024	False	1	1000000
7	7	amazon_polarity	distilbert-base-uncased	66955010	hf-benchmarking-samstu	32	0.00005	30.0	ml.p3.2xlarge	1	32	84375.0	3.825	1024	False	1	100000
8	8	amazon_polarity	distilbert-base-uncased	66955010	hf-benchmarking-samstu	32	0.00005	3.0	ml.p3.16xlarge	32	1024	2637.0	28.152	30	True	4	1000000
9	9	amazon_polarity	distilbert-base-uncased	66955010	hf-benchmarking-samstu	32	0.00005	5.0	ml.p3.2xlarge	1	32	84375.0	3.825	1024	False	1	600000
10	10	amazon_polarity	distilbert-base-uncased	66955010	hf-benchmarking-samstu	32	0.00005	30.0	ml.p3.16xlarge	16	512	5273.0	28.152	30	True	2	100000
11	11	amazon_polarity	distilbert-base-uncased	66955010	hf-benchmarking-samstu	32	0.00005	30.0	ml.p3.16xlarge	32	1024	2637.0	28.152	30	True	4	100000
12	12	amazon_polarity	distilbert-base-uncased	66955010	hf-benchmarking-samstu	32	0.00005	3.0	ml.p3.2xlarge	1	32	84375.0	3.825	1024	False	1	1000000
13	13	amazon_polarity	distilbert-base-uncased	66955010	hf-benchmarking-samstu	32	0.00005	30.0	ml.p3.16xlarge	16	512	5273.0	28.152	30	True	2	100000
14	14	amazon_polarity	distilbert-base-uncased	66955010	hf-benchmarking-samstu	32	0.00005	5.0	ml.p3.16xlarge	16	512	5273.0	28.152	30	True	2	600000
15	15	amazon_polarity	distilbert-base-uncased	66955010	hf-benchmarking-samstu	32	0.00005	5.0	ml.p3.8xlarge	4	128	21094.0	14.688	1024	False	1	600000
16	16	amazon_polarity	distilbert-base-uncased	66955010	hf-benchmarking-samstu	16	0.00005	5.0	ml.p3.16xlarge	8	128	10547.0	28.152	1024	True	1	600000
17	17	amazon_polarity	distilbert-base-uncased	66955010	hf-benchmarking-samstu	32	0.00005	30.0	ml.p3.16xlarge	8	256	10547.0	28.152	1024	True	1	100000
18	18	amazon_polarity	distilbert-base-uncased	66955010	hf-benchmarking-samstu	32	0.00005	3.0	ml.p3.16xlarge	8	256	10547.0	28.152	1024	True	1	1000000
19	19	amazon_polarity	distilbert-base-uncased	66955010	hf-benchmarking-samstu	32	0.00005	30.0	ml.p3.8xlarge	4	128	21094.0	14.688	1024	False	1	100000
20	20	amazon_polarity	distilbert-base-uncased	66955010	hf-benchmarking-samstu	32	0.00005	3.0	ml.p3.8xlarge	4	128	21094.0	14.688	1024	False	1	1000000

Figure 3: Complete experimental design with 21 runs.

Capturing Metrics

As noted previously, the responses of interest in this benchmark are model training time, model performance, and cost. Metrics were systematically captured by the SageMaker training job corresponding to each experimental run.

Model training time was captured by the training job execution time, model performance by validation F1 score, and job cost calculated given the training job billable seconds per instance, number of instances, and hourly instance price.

Limitations of Experimental Design

The results of this benchmarking study should be interpreted in the context of the deep learning hyperparameters applied in the training jobs. Specifically, the results should be contextualized given the learning rate was fixed global batch size increased with compute scaling. The implications of the experimental design on the results, and some alternative approaches that could be pursued for benchmarking in the future, are expanded upon in Results & Discussion.

RESULTS & DISCUSSION:

Training Time & Job Cost

Training job execution time decreased exponentially as compute resources increased, as shown in Figure 4 and described in Table 3. By fixing the per device batch size (i.e. the batch size per GPU) to 32 samples in all cases, the global batch size increased proportionally with any increase in the number of GPUs in the instance configuration.

The greatest time saving was in vertically scaling from one ml.p3.2xlarge instance containing a single GPU to an ml.p3.8xlarge with 4 GPUs, saving nearly 13 hours in training time for an incremental cost increase running the job of $8.44. Interestingly, as noted in Figure 5 and Table 3, two node data parallelism on ml.p3.16xlarge pays for itself by maintaining the same cost as training on one ml.p3.8xlarge instance, in approximately 4h less time. In view of all experiments performed, two-node data parallelism on 16xl instances delivered the best value.

Table 3: Time and money saved by scaling compute

	1 GPU	*4 GPU*	*8 GPU*	*16 GPU*	*32 GPU*
Instance Type	1 Node ml.p3.2xlarge	1 Node ml.p3.8xlarge	1 Node ml.p3.16xlarge	2 Node ml.p3.16xlarge	4 Node ml.p3.16xlarge
Global Batch Size	32	128	256*	512	1024
Mean Train Time	18h 17m	5h 20m	3h 41m*	16h 24m	53m
Std Dev	+/- 6m 30sec	+/- 4m 03sec	+/- 42m 58sec*	+/- 4m 4sec	+/- 1m 15sec
Sample Size	5	3	3*	4	6
Time Saved by Scaling	-	+12h 57m	+1h 39m	+2h 18m	+30m 50sec
$USD Job Cost Difference by Scaling	-	+$8.44	+$25.54	- $25.47	+$20.57

* Experimental run 16, *(600,000 samples running on 1 node of a p3.ml.16xlarge with 8GPUs)* was executed with a per device batch size of 16 rather than 32, to work around a CUDA out of memory error. Consequently, this change decreased the global batch size for this observation to 128, which translated to increased training time and cost for one of the three total runs performed on 8 GPUs.

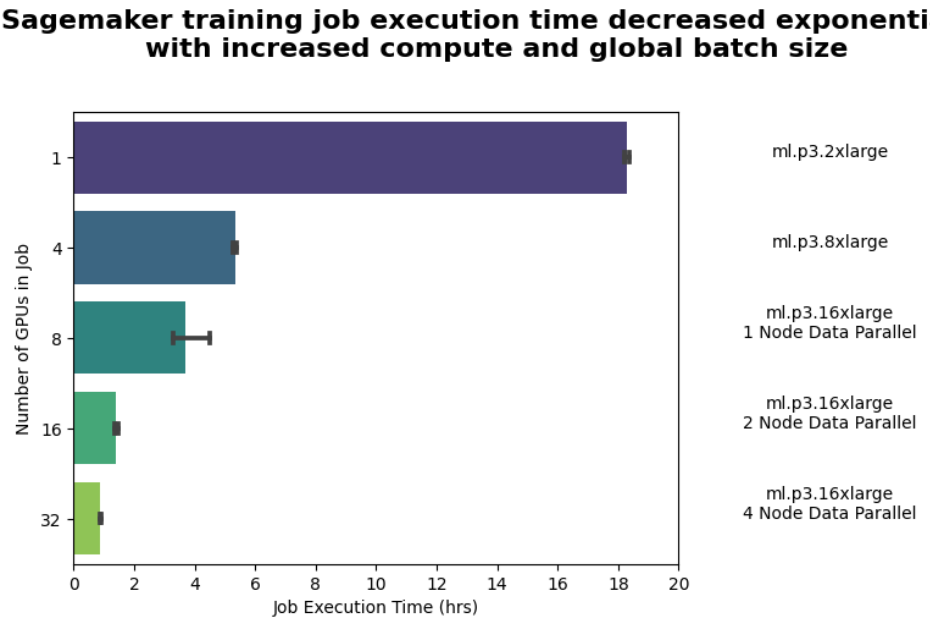

Figure 4: Training job execution time with respect to number of GPUs in the underlying compute instance used and global batch size. Per GPU batch size was 32. Learning rate fixed at 5e-5. Global Batch Sizes: p3.2xlarge (32), p3.8xlarge (128), 1 Node p3.16xlarge (256), 2 Node p3.16xlarge (512), 4 Node p3.16xlarge (1024). See Table 1 for a full outline of deep learning parameters used.

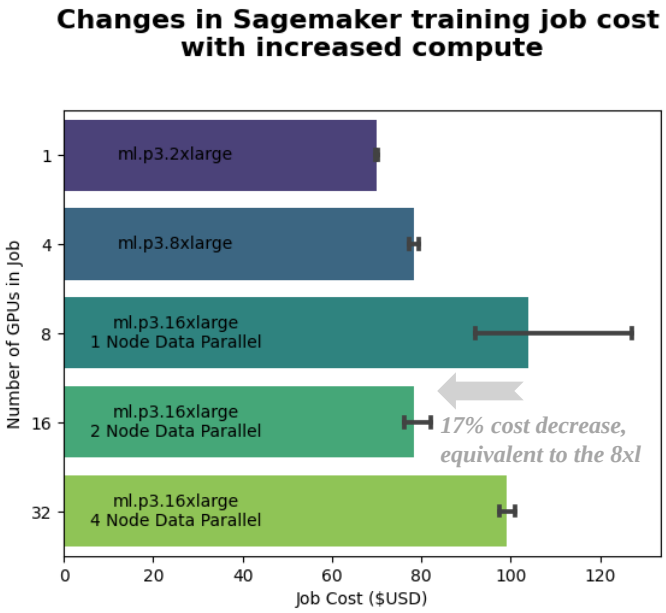

Figure 5: Introducing 2-Node training with p3.16xlarge instances and SageMaker Data Parallelism can decrease job cost, despite the increasing compute. Per GPU batch size was 32. Learning rate fixed at 5e-5. Global Batch Sizes: p3.2xlarge (32), p3.8xlarge (128), 1 Node p3.16xlarge (256), 2 Node p3.16xlarge (512), 4 Node p3.16xlarge (1024). See Table 1 for a full outline of deep learning parameters used.

Model Performance

Validation F1 scores trended down as compute resources increased, for all three dataset sizes, as shown in Figure 6. The decrease was greatest for the 100,000 sample dataset.

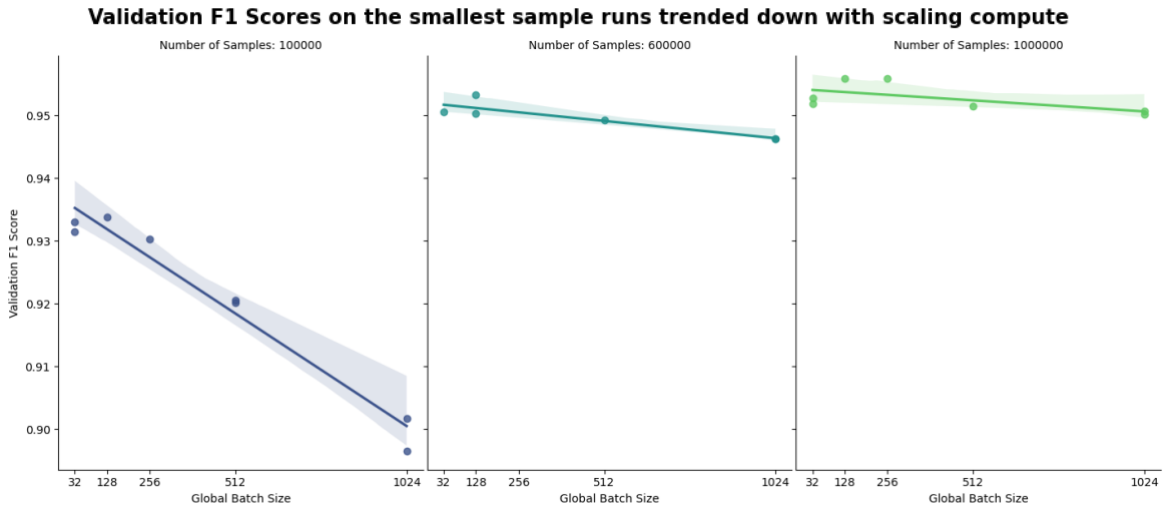

Figure 6: Validation F1 scores trended down with increased compute and global batch size. Per GPU batch size was 32. 95% confidence interval highlighted, all original data points and their regression lines shown. Learning rate fixed at 5e-5. Global Batch Sizes: p3.2xlarge (32), p3.8xlarge (128), 1 Node p3.16xlarge (256), 2 Node p3.16xlarge (512), 4 Node p3.16xlarge (1024).

The cause of the performance decrease was investigated. An important consideration was the selection of the deep learning parameters that were fixed in the experimental design. Generally, as global batch size increases, it is recommended in deep learning practice to proportionally update the learning rate [5]. The learning rate dictates the extent to which model weights are changed with each training step. Hence, as the step size (global batch size) increases, the learning rate should be proportionally adjusted to maintain equivalent model learning, and thus model performance.

In this benchmarking experiment, the learning rate was kept constant (5e-5). The per device (GPU) batch size was also fixed (32). The global batch size was increased *by means of* increasing the number of GPUs in the compute configuration. As a result, this combination of deep learning parameters while scaling compute yielded a decrease in model performance.

It is known that with larger compute configurations, a practitioner *can* fit larger global batch sizes during training, which decreases training time [5]. A key takeaway from this benchmarking study, however is that when scaling compute for model training jobs, **the practitioner should examine how their baseline deep learning hyperparameters will be impacted by scaling.** Similar advice is echoed in other practical literature on implementing distributed training and scaling compute to reduce training time [2]. Examining the changes in deep learning hyperparameters such as global batch size, learning rate, per GPU batch size, and number of GPUs in the compute configuration before and after scaling will suggest how model performance will change with scaling. If care is not taken to anticipate how hyperparameters will change with

scaling, some of the time saving benefits of scaling may be lost to the additional time taken to remediate performance dips [2].

Remediating Performance Drop by Scaling Learning Rate

There is empirical evidence that for the AdamW optimizer (used with HuggingFace by default) the learning rate can be adjusted by $\sqrt{n} \times \propto_1$ where n is the proportional increase in global batch size and $\propto_1$ is the original learning rate. [3][4].

To explore the implications of this adjustment on the present benchmarking study, experimental runs 21and 22 repeated runs 11 and 2 respectively, only using empirically adjusted learning rates. Specifically, run 21 repeats the training job with 100,000 samples and a global batch size of 1024 with the proportionally adjusted learning rate (2.83e-4 = $\sqrt{32} \times$ 5e-5 for a 32x increase in global batch size). Run 22 runs the equivalent run with 1,000,000 samples. Figure 7 shows the result. The validation F1 performance did improve for the case of 100,000 samples, however not in the case of 1,000,000 samples. Albeit, the experiment using 1,000,000 samples did not materially drop in performance with the increased global batch size in the original experiment. Further optimizations with hyperparameter tuning could be further explored to fully remediate the performance drop if desired. However, as noted above, this will consume some of the time saved by scaling the training job.

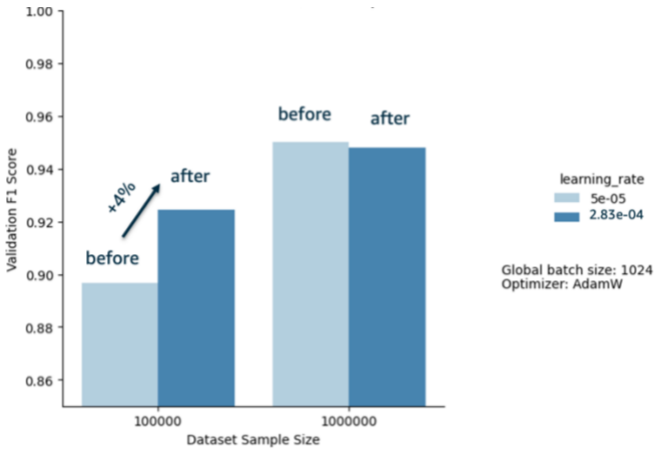

Figure 7: Increasing the learning rate according to the empirical scaling rule ($\sqrt{32} \times 5e-5$ for a 32x in global batch size) increased improved performance at 100,000 samples (F1=0.897 to F1=0.924). However, remediated performance was still 1% below the F1 score obtained on 1 GPU at the original global batch size of 32 (F1=0.931).

Alternative approaches to changing hyperparameters when scaling

To bypass the need to vary both global batch size and learning rate with scaling compute, other parameters can be adjusted. Specifically, the global batch size and learning rate originally yielding acceptable model performance should be fixed at the outset of the scaling project. The parameter to vary instead is the per device (per GPU) batch size. This parameter should be varied as required preserve the global batch size while more GPUs are added to the compute configuration. This configuration for benchmarking is expected to maintain performance.

Future work is necessary to benchmark an absence of performance changes when pursuing this approach. This aws-samples repo can be easily modified to pursue the aforementioned alternative approach in future work with distilbert. Simply reconfigure the experimental design (in get_results.ipynb) to vary the number of epochs and per device batch size in such a way that global batch size is fixed while scaling the number of GPUs in the compute configuration. Equations for calculating epochs are previously described in Table 2, and in the get_results notebook. Similarly, to compare performance at different dataset sizes, ensure the number of steps are fixed at a given global batch size, and the number of epochs are varied to enforce the same number of steps for all dataset sizes. Equations for setting step size are similarly described in the methods section. The resulting benchmarking experiment configuration will have a fixed global batch size, learning rate, and number of steps in all cases, with a varying per device batch size, and a varying number of epochs. Epochs will vary as the dataset sizes being benchmarked change, such that the same number of steps is enforced. Some guessing and checking may be required to ensure the resulting epochs are reasonable whole numbers, and the per device batch sizes required are compatible with device memory in all cases (i.e. avoiding CUDA out of memory errors).

PITFALLS & POINTS OF CONFUSION

To aid others interested in working with HuggingFace and scaled compute to reduce training times with SageMaker, some pitfalls and points of confusion identified during this benchmarking study are expanded below.

- **CUDA out of memory errors**
  - An important factor in scaling compute is managing SageMaker training job compatibility with device memory. A device in this context is a GPU. The HuggingFace training job arguments include a per_device_train_batch_size hyperparameter which refers to the per GPU batch size. See the HuggingFace docs for further details.
  - If you are encountering CUDA out of memory errors, the per_device_batch_size may require an adjustment down, such that the batches can fit in device memory. Additionally, adding save_total_limit = 1 to TrainingArguments() in the SageMaker training script will drastically reduce the model size saved and can help with conserving memory.

- **Ensure version compatibility**
  - When running SageMaker data parallelism with HuggingFace, make sure to use version 4.3.0 of the transformers library or newer in the training script
  - The HuggingFace Deep Learning Container Image URI used in this work was:
    - 763104351884.dkr.ecr.us-east-1.amazonaws.com/huggingface-pytorch-training:1.7.1-transformers4.6.1-gpu-py36-cu110-ubuntu18.04
  - The PyTorch version used was: ‘py36’

- **Transfer Learning vs Fine-Tuning**
  - HuggingFace can be used for fine-tuning (where all model parameters are adjusted by re-training a pretrained model with a custom dataset) or transfer learning (freezing encoder weights and just training the classification head).
  - Transfer learning can be faster than fine tuning as it drastically reduces the number of tunable parameters (for distilBERT, about 67 million tunable parameters are reduced to about 67,000). However, model performance was not as good as with the fine-tuning approach, and fine-tuning was pursued for this study.

- **EBS Volume Size**
  - Originally, all EBS volume sizes passed into the training job were left to the default value of 30 GBs.
  - While the training jobs running on the 16xl instances were successful with the defaults, all other instance types required additional EBS volume in their training jobs (1024 GBs was sufficient to get all jobs to work) to avoid training job failure due to an ArchiveError. The volume size ultimately required by each training jobs is indicated in the experimental design.

RECOMMENDED BEST PRACTICES

- **Troubleshoot training jobs on one instance before scaling**
  - Start by successfully executing a custom training job on a single instance before scaling to multi-node training jobs with SageMaker parallelism
  - Errors are more challenging to debug on multi-node jobs
  - Use a warm start to the learning rate (turned on by default with HuggingFace, and warm start defaults were used presently) to help with convergence

- **Scale vertically before scaling horizontally**
  - You may accomplish your training time goals by scaling vertically and enjoy the simplicity of working on one instance, as noted in a recently published practical guide to scaling compute [2]

- **If need to reduce time and cost further, scaling horizontally can in some cases be cheaper and faster than scaling vertically**
  - As shown in Figure 4 and Figure 5 under the experimental conditions of this benchmark, multi-node parallelism with two ml.p3.16xl instances can be faster and cheaper than one node, and equivalent in cost but 4hours less in time than a single ml.p3.8xl instance

- **Plan in advance for how model performance may be impacted by scaling compute**
  - Anticipate how your global batch size, learning rate, and per device batch size (AKA per GPU batch size) may need to be adjusted to avoid changes in performance
  - See *Alternative approaches to changing hyperparameters* when scaling for details

- **Review additional best practices from the SageMaker Parallelism documentation**
  - Additional best practices on selecting a batch size elaborated on in the SageMaker Parallelism documentation

Links to Additional Reading on HuggingFace, SageMaker, & Distributed Training

- Free getting started with HuggingFace Course, Part 1
- HuggingFace on Amazon SageMaker Big Picture
- Using HuggingFace with Amazon SageMaker Overview
- SageMaker Data Parallel Distributed Training Library Configuration and Pitfalls
- HuggingFace Estimator class documentation
- Preparing a HuggingFace transformers fine tuning script in SageMaker

REFERENCES

[1] J. Simon, “New – Data Parallelism Library in Amazon SageMaker Simplifies Training on Large Datasets,” *AWS News Blog*, 2020. https://aws.amazon.com/blogs/aws/managed-data-parallelism-in-amazon-sagemaker-simplifies-training-on-large-datasets/.

[2] O. Cruchant, *The Efficient Machine Learning Practitioner*. LeanPub.com, 2021.

[3] Y. You, I. Gitman, and B. Ginsburg, “Large Batch Training of Convolutional Networks,” 2017. [Online]. Available: http://arxiv.org/abs/1708.03888.

[4] E. Hoffer, I. Hubara, and D. Soudry, “Train longer, generalize better: closing the generalization gap in large batch training of neural networks,” 2017, doi: 10.1016/j.jcjd.2014.02.001.

[5] S. L. Smith, P. Kindermans, C. Ying, Q. V Le, and G. Brain, “Don’t Decay the Learning Rate Increase the Batch Size,” no. 2017, pp. 1–11, 2018.