The Impact of Importance-Aware Dataset Partitioning on Data-Parallel Training of Deep Neural Networks

Sina Sheikholeslami[1(✉)], Amir H. Payberah[1], Tianze Wang[1], Jim Dowling[1,2], and Vladimir Vlassov[1]

[1] KTH Royal Institute of Technology, Stockholm, Sweden
{sinash,payberah,tianzew,jdowling,vladv}@kth.se
[2] Hopsworks AB, Stockholm, Sweden
jim@hopsworks.ai

Abstract. Deep neural networks used for computer vision tasks are typically trained on datasets consisting of thousands of images, called examples. Recent studies have shown that examples in a dataset are not of equal importance for model training and can be categorized based on quantifiable measures reflecting a notion of "hardness" or "importance". In this work, we conduct an empirical study of the impact of importance-aware partitioning of the dataset examples across workers on the performance of data-parallel training of deep neural networks. Our experiments with CIFAR-10 and CIFAR-100 image datasets show that data-parallel training with importance-aware partitioning can perform better than vanilla data-parallel training, which is oblivious to the importance of examples. More specifically, the proper choice of the importance measure, partitioning heuristic, and the number of intervals for dataset repartitioning can improve the best accuracy of the model trained for a fixed number of epochs. We conclude that the parameters related to importance-aware data-parallel training, including the importance measure, number of warmup training epochs, and others defined in the paper, may be considered as hyperparameters of data-parallel model training.

Keywords: Data-parallel training · Example importance · Distributed deep learning

The authors would like to acknowledge funding from Vinnova for the Digital Cellulose Competence Center (DCC), Diary number 2016-05193. This work has also been partially supported by the ExtremeEarth project funded by European Union's Horizon 2020 Research and Innovation Programme under Grant Agreement No. 825258. The computations for some of the experiments were enabled by resources provided by the National Academic Infrastructure for Supercomputing in Sweden (NAISS) and the Swedish National Infrastructure for Computing (SNIC) at C3SE, partially funded by the Swedish Research Council through grant agreement no. 2022-06725 and no. 2018-05973. Artifacts are available in https://doi.org/10.5281/zenodo.7855247 and https://github.com/ssheikholeslami/importance-aware-data-parallel-training.

M. Patiño-Martínez and J. Paulo (Eds.): DAIS 2023, LNCS 13909, pp. 74–89, 2023.
https://doi.org/10.1007/978-3-031-35260-7_5

1 Introduction

Data-parallel training (DPT) is the current best practice for training deep neural networks (DNNs) on large datasets over several computing nodes (a.k.a. *workers*) [11]. In DPT, the DNN (model) is replicated among the workers, and the training dataset is partitioned and distributed uniformly among them. DPT is an iterative process where in each iteration, each worker trains its model replica on its dataset partition for one epoch. After each iteration, the parameters or gradients of the worker models are aggregated and updated. Then, all workers continue the training using the same updated model replicas. This "vanilla" DPT scheme is shown in Fig. 1.

The dataset partitions in vanilla DPT are constructed by *random partitioning*, i.e., randomly assigning training examples to each partition. However, it is known that not all examples within a training dataset are of equal *importance* for training DNNs [2,3,6,13] meaning that different examples contribute differently to the training process and the performance of the trained model (e.g., its prediction accuracy). Prior works have used example importance to improve DNN training schemes, mainly aiming at reducing the total training time or increasing the performance of the trained models. For example, in *dataset subset search* [3], the goal is to find subset(s) of a given training dataset that can be used to train equally good or more performant models compared to the models trained on the initial dataset. Example importance has also been used for developing more effective sampling algorithms for stochastic gradient descent (SGD) [6], or in active learning for choosing the best examples to label [2].

Contributions. All the above-mentioned solutions are mainly designed for non-distributed model training. In this paper, we study different heuristics to assign examples, based on their importance, to workers in a distributed environment and in DPT. In particular, the contributions of this work are as follows.

- We introduce *importance-aware* DPT, which replaces the random partitioning of the dataset across workers in vanilla DPT, with heuristics that partition the dataset based on some pre-determined notion of example importance, e.g., the average loss value of each example over a number of training epochs.
- We study the effects of the hyperparameters of importance-aware DPT, including different (i) example importance measures and metrics, (ii) partitioning heuristics, and (iii) partitioning intervals, on the quality of the training scheme. Our experiments for image classification tasks on CIFAR-10 and CIFAR-100 datasets demonstrate that importance-aware DPT can outperform vanilla DPT in terms of the best test accuracy achieved by models.

The remainder of this paper is structured as follows. In Sect. 2, we provide the necessary background, including an overview of DPT and a review of some related work. In Sect. 3, we present importance-aware DPT and discuss how it differs from vanilla DPT, which is importance-oblivious. In Sect. 4, we discuss our prototype implementation of importance-aware DPT in PyTorch. In Sect. 5, we present the results of our experimental evaluation of importance-aware DPT.

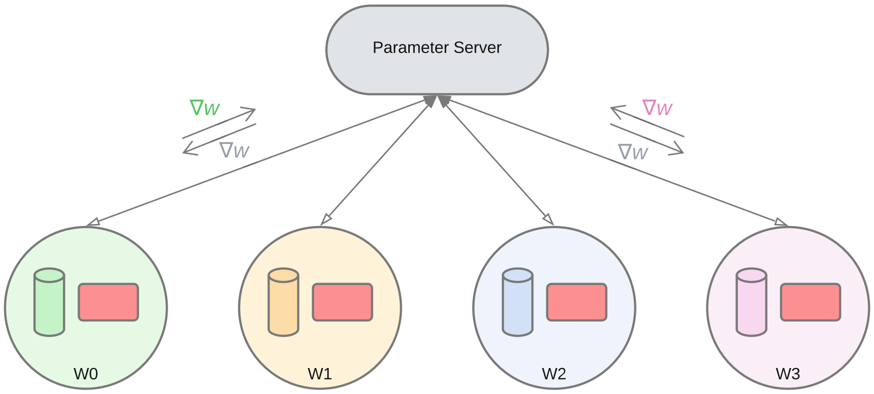


Fig. 1. The vanilla DPT scheme with four workers and one parameter server. At each epoch, each worker gets a random partition of the dataset, and all the workers are assigned the same model replica. After one epoch of training, the workers send their local gradients or model parameters to the parameter server. The parameter server performs either gradient aggregation or model aggregation and sends back the new gradients or parameters to the workers.

Finally, in Sect. 6, we give our conclusions and discuss the current limitations of our importance-aware DPT prototype and further research directions.

2 Background and Related Work

Our work presented in this paper lies in the intersection of data-parallel DNN training and prior work that studies the difference of examples within a dataset in terms of their importance for model training. In this section, we give a brief overview of the DPT of DNNs and some related work on example importance.

2.1 DNN Data-Parallel Training (DPT)

Given a training dataset D consisting of training examples $e \in D$, the aim of training the model M is to optimize model parameters with regards to a cost function, e.g., Mean Squared Error or Binary Cross-Entropy, using an iterative optimization algorithm, e.g., Stochastic Gradient Descent. A training dataset is typically made up of examples of a specific type, such as images, structured data, or sentences. During each *epoch* of training, batches of examples are passed through the model, and model parameters are optimized using the iterative optimization algorithm. To scale out the training process, one can use multiple processing nodes, a.k.a. *workers*, and partition the DNN (for model-parallelism) or the dataset (for DPT) and assign them to the workers to enable parallel training. For our purposes, we define a worker $w \in W$ as a process within a processing node that is allocated exactly one GPU, i.e., each worker corresponds to exactly one GPU in our cluster of processing nodes.

In a typical most common DPT scheme, which we refer to as *vanilla DPT*, the DNN is replicated across the workers. At the beginning of each epoch, the dataset is partitioned uniformly at random into disjoint subsets $p \in P$, such that worker w_i is allocated the partition p_i (dataset partitioning step). More formally, $P = \bigcup_{i=0}^{n-1} p_i$ such that $p_i \cap p_j = \emptyset$ for $i \neq j$; and $p_i \neq \emptyset$ for each i. For simplicity, we assume that the number of examples in the dataset, or $|D|$, is divisible by the number of workers, $n = |W|$; but the approach and results can easily be extended to cases where the assumption does not hold.

During an epoch, each worker independently trains its own replica of the DNN model (local training step) on its own partition p_i. At the end of an epoch, a model synchronization step occurs, e.g., using a parameter server, and the workers get a new identical replica of the model. This process is repeated for a specified budget (e.g., a pre-determined number of epochs) or until a model convergence criterion or performance metric is satisfied. We are interested to see if using a partitioning function, based on notions of example importance, may lead to better results compared to vanilla DPT's random partitioning in terms of the target performance metrics. We define the importance of an example, denoted by Imp, as a mapping of an example to a scalar value:

$$Imp : e \rightarrow \mathbb{R} \tag{1}$$

In practice, to implement Imp, a certain property of the example or the result of its interactions with the model (e.g., the loss generated by the example after a forward pass) is used in combination with an aggregation method (e.g., average, or variance of the losses over a number of epochs).

A partitioning function $PartitioningFunction$ maps the examples to workers to create the set of partitions P, where each worker w_i gets the partition p_i. We are interested in using the output of Imp to construct the $PartitioningFunction$. Example definitions for a $PartitioningFunction$ are explained in Sect. 3.3.

2.2 Prior Work on Example Importance

The diversity of examples in training datasets has attracted increasing attention in recent years and has been exploited to improve the state-of-the-art in domains such as dataset subset search [3,12,13] and sampling for SGD [2,6,13,14].

Chitta et al. [3] propose an ensemble active learning approach for dataset subset selection using ensemble uncertainty estimation. They also show that training classifiers on the subsets obtained in this way leads to more accurate models compared to training on the full dataset. Isola et al. [5] investigate the *memorability* of different examples based on the probability of each image being recognized (perceived as a repetition by the viewer) after a single view and train a predictor for image memorability based on image features. Memorability is also a familiar phenomenon to humans, as we can all think of images or visual memories that have stuck more in our minds compared to other images. Arpit et al. [1] define example difficulty as the average misclassification rate over a number of experiments.

Chang et al. [2] propose to prefer *uncertain* examples for SGD sampling, e.g., the examples that are neither consistently predicted correctly with high confidence nor incorrectly. They use two measures for "example uncertainty": (i) the variance of prediction probabilities and (ii) the estimated closeness between the prediction probabilities and the decision threshold. Yin et al. [14] observe that high similarity between concurrently processed gradients may lead to the speedup saturation and degradation of generalization performance for larger batch sizes and suggest that diversity-inducing training mechanisms can reduce training time and enable using larger batch sizes without the said side effects in distributed training.

Vodrahalli et al. [13] propose an importance measure for SGD sampling based on the gradient magnitude of the loss of each example at the end of training and use this measure to select a subset of the dataset for retraining. This measure can also be used to study the diversity of examples in datasets. Katharopoulos and Fleuret [6] propose an SGD sampling method that favors the more *informative* examples, which they describe as the examples that lead to the biggest changes in model parameters. Toneva et al. [12] propose *forgettability* as an importance measure for dataset examples. A forgettable example is an example that gets classified incorrectly at least once, after its first correct classification, over the course of training. They also suggest that the forgetting dynamics can be used to remove many examples from the base training dataset without hurting the generalization performance of the trained model.

Finally, in the domain of natural language processing, Swayamdipta et al. [10] have investigated the difference in example importance. They introduce *data maps* and calculate two measures for each example: the confidence of the model in the true class and the variability of the confidence across different epochs in a single training run. They then categorize the examples into three categories: *easy-to-learn*, *ambiguous*, and *hard-to-learn*.

3 Importance-Aware DPT

Importance-aware DPT consists of three stages of model training, as shown in Fig. 2. In the first stage, which we refer to as *warmup training*, we train the DNN using vanilla DPT for a number of "warmup" epochs (E_{warmup}). Blocks (1) and (2) in Fig. 2 show the first stage. In the second stage, we calculate the *importance* of each example according to a predefined importance measure, e.g., the average loss value of each example over E_{warmup} training epochs. In the third stage (blocks (3)–(5) in Fig. 2), we continue training using *importance-aware* DPT in several *intervals*. Each interval consists of three steps: (i) dataset partitioning, i.e., assigning examples to partitions based on a *heuristic* and allocating one partition to each worker, (ii) model training, i.e., training the DNN using those fixed partitions for $E_{interval}$ epochs, and (iii) example importance calculation, in which we recalculate and update the importance value of each example for the next interval. In the rest of this section, we discuss importance-aware DPT in more detail.

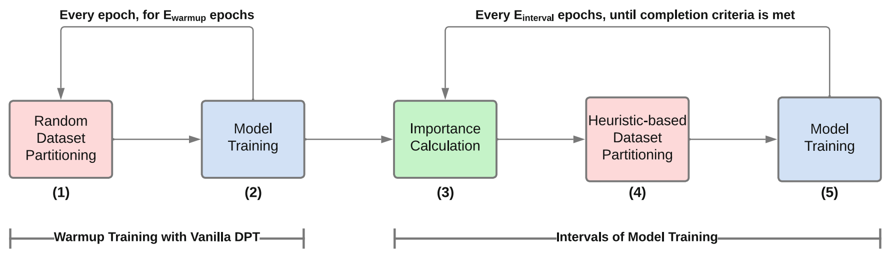

Every epoch, for E_warmup epochs
Every E_interval epochs, until completion criteria is met
Random Dataset Partitioning
Model Training
Importance Calculation
Heuristic-based Dataset Partitioning
Model Training
(1)
(2)
(3)
(4)
(5)
Warmup Training with Vanilla DPT
Intervals of Model Training

Fig. 2. An overview of Importance-aware data-parallel training. The model is first trained with Vanilla DPT for E_{warmup} epochs, after which the random dataset partitioning is replaced with heuristic-based dataset partitioning, and the dataset is partitioned at the beginning of each interval of training rather than at the beginning of each epoch.

3.1 Warmup Training

In the first stage, warmup training, the model is trained with vanilla DPT for E_{warmup} epochs, in which the dataset is randomly partitioned among the workers at the beginning of each epoch. We collect the value(s) needed for calculating the importance of examples during this stage. In this work, we use the loss value (the result of backpropagation forward pass) of each example in each epoch to calculate its importance value, which is the average loss over a number of epochs. It is worth noting that we will discard the loss values from the first E_{ignore} epochs in warmup training (e.g., the first three epochs), as the losses generated in the first few epochs are influenced by the random initialization of the neural network to a high degree.

3.2 Importance Calculation

The second stage is a pause in model training, in which we calculate the importance of examples using values collected during warmup training. To demonstrate how this works, consider we calculate the importance of each example using "average loss across epochs". To do this, during warmup training, we collect the loss values (the result of the forward pass) of each example across E_{warmup} epochs. At the end of warmup training, we will have a matrix such as in Fig. 3. In this matrix, each row corresponds to a single example, and each column corresponds to an epoch. Hence, an element $a_{i,j}$ in the matrix is the loss value of example i in epoch j. Calculating the importance of each example would then require a simple aggregation or computation over each row, e.g., a row-wise average. At the end of this stage, we have one or more scalar values attributed to each example, indicating its importance, which we use for sorting or categorizing the examples in the next stage (dataset partitioning).

$$\begin{bmatrix} 2.4630 & 1.6089 & ... & 0.8972 \\ ... & ... & ... & ... \\ ... & ... & ... & ... \\ ... & ... & ... & ... \\ 0.9879 & 3.1874 & ... & 1.7276 \end{bmatrix}$$

Fig. 3. example-epoch-loss matrix that is used to calculate the importance score of each example.

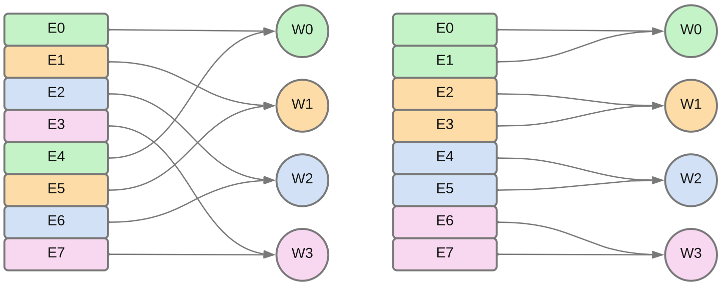


Fig. 4. Depiction of `Stripes` (left) and `Blocks` (right) partitioning heuristics for a setting with eight examples (indexed in order of importance) and four workers.

3.3 Dataset Partitioning Heuristics

Now that we have a mapping between examples and their importance values, we can use various heuristics to proceed with dataset partitioning for importance-aware DPT. Remind that in vanilla DPT, the examples are partitioned randomly across the parallel workers at the beginning of each epoch. We have defined two such heuristics, namely `Stripes` and `Blocks`, and compared them with random partitioning (i.e., vanilla DPT).

Stripes Heuristic. The `Stripes` partitioning heuristic is a cyclic assignment of examples to workers. The intuition behind using this heuristic is to preserve the same distribution of examples with regard to their importance values, in each partition. To this end, we sort the examples of the dataset D by their importance value and create a list called *Sorted Examples* (SE). Then, the partition P_i that is allocated to worker w_i is determined as:

$$P_i = \{e \in D \mid sorted\_index(e) \equiv i \pmod n\} \tag{2}$$

where $sorted\_index(e)$ returns the index of example e in the sorted list SE, n is the number of workers, and $i = 0, ..., n-1$. The `Stripes` heuristic is depicted on the left side of Fig. 4.

Blocks Heuristic. This partitioning heuristic assigns a continuous block of examples to each worker so that we will end up with different importance distributions across the workers. Assuming n workers, the `Blocks` heuristic allocates the first $\frac{|D|}{n}$ examples ranked in the SE list to the first worker, the second $\frac{|D|}{n}$ of SE to the second worker, and so on. Thus, the partition P_i that is allocated to worker w_i using the `Blocks` heuristic is determined as follows:

$$P_i = \{e \in D \mid i \times \frac{|D|}{n} \leq sorted\_index(e) < (i+1) \times \frac{|D|}{n}\} \tag{3}$$

where $sorted\_index(e)$ returns the index of example e in the sorted list SE and $i = 0, ..., n-1$. The `Blocks` heuristic is depicted on the right side of Fig. 4.

3.4 Intervals of Model Training

After warmup training, calculating example importance, and partitioning the dataset based on the importance values, we continue model training using fixed partitions in intervals, each comprising of $E_{interval}$ epochs. At the beginning of each training interval, we repartition the dataset using the importance values calculated during the previous interval. This means that dataset repartitioning only occurs at the beginning of each interval rather than at the beginning of every epoch (as in vanilla DPT).

4 Implementation in PyTorch

This section presents the implementation details of importance-aware data-parallel training in PyTorch v1.10.1 [8,9]. The implementation is mainly based on several classes and methods that (i) track and calculate the importance of examples as explained in Sects. 3.1 and 3.2, (ii) partition the dataset across workers based on importance-aware heuristics defined in Sect. 3.3, and (iii) resume and continue the model training for fixed intervals of $E_{interval}$ epochs as described in Sect. 3.4.

4.1 Importance Calculation

Our proof-of-concept implementation of importance-aware DPT provides importance calculation for each example based on its average forward pass loss across a number of epochs. Loss function implementations in PyTorch, by default, do a batch-wise reduction on the losses and return a scalar aggregate value (e.g., the average loss of examples in the mini-batch when using `CrossEntropyLoss`[1]). To get individual (per example) loss values, we construct an additional loss function of the same type and set its `reduction` parameter to `None`. This way, this loss function returns a tensor instead of a scalar.

[1] As described in https://pytorch.org/docs/stable/generated/torch.nn.CrossEntropyLoss.html.

Hence, each step of the training consists of two forward passes: the first one uses the customized loss function and writes values to a local worker copy of a loss-epochs matrix similar to the one depicted in Fig. 3, and the second forward pass uses the default loss function implementation which is used with the backward pass. Each worker maintains its own copy of the loss-epochs matrix, and before each dataset partitioning step, the workers wait at a barrier (by calling `torch.distributed.barrier()`) for the main process to merge the local copies and aggregate, i.e., to compute the row-wise average which is the average loss of each example across the epochs. The output of this step is a sorted list of tuples `(example, importance value)` - the *Sorted Examples* list introduced in Sect. 3.3, that is used with the importance-aware partitioning heuristics.

4.2 Dataset Partitioning Heuristics

In PyTorch, the `DistributedDataSampler` class implements the logic for assigning examples to workers. By default, this class contains an implementation of random sampling, so we extend this class and add a sampler, called `ConstantSampler`, to arbitrarily assign the examples to workers. In this way, we decouple the implementation for assigning examples to workers, from the implementation of importance-aware partitioning heuristics. Hence, the same `ConstantSampler` can be used with different partitioning heuristics.

A dataset partitioning heuristics provides a mapping between examples and workers. We implement this mapping in PyTorch by creating a dictionary (`dict`) with worker indices as keys and a list of example indices as the value of each key. Depending on the heuristic, filling in this dictionary would then require iterating over the list of examples or workers. The result of this step, which is a `dict` that maps examples to workers, is used to construct a `ConstantSampler` instance that assigns the dataset examples across the workers. Once the `ConstantSampler` instance is constructed, the main process also reaches the barrier, so all the worker processes exit the barrier they had entered before merging their local matrices (as described in the previous section).

4.3 Modified Training Loop for Importance-Aware Training

Model training in PyTorch typically consists of a few blocks of code for setting up the training (e.g., downloading the dataset, constructing the train/test/validation folds and data samplers, and creating the model), followed by a single loop for iterative training of the model. To implement different stages of importance-aware DPT, we first break down the default training loop into two separate loops: one for warmup training (Sect. 3.1) and the other for intervals of importance-aware training (Sect. 3.4). The first loop is similar to a typical PyTorch training loop but is extended with code to track and calculate the importance of examples. The second loop is nested: an outer loop maintains the intervals, while the inner loop contains the code for the actual dataset partitioning step, the example importance calculation step, and the model training step.

5 Evaluation

In this section, we describe our experimental setup and scenarios and discuss the results of the experiments. When talking about "model performance" we mainly refer to *best test accuracy* of a model trained for 100 epochs. Our hardware setup consists of a single machine with 4 GeForce RTX 2070 SUPER graphic cards, so we train on 4 workers.

5.1 Experimental Setup

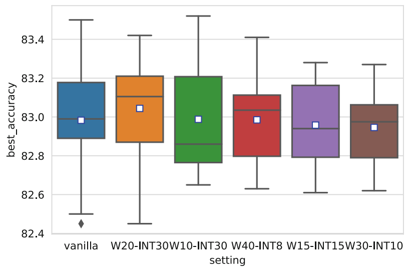
To be able to empirically evaluate the effects of importance-aware dataset partitioning on the performance of DPT systems, we use two well-known DNN architectures for image classification: ResNet-18 and ResNet-34 [4] and train them on CIFAR-10 and CIFAR-100 datasets [7]. We use official PyTorch implementations of the models[2] and initialize them with random weights. In total, our experiments consist of 1830 training runs across 183 workloads (different combinations of datasets, models, partitioning heuristics, importance metrics, E_{warmup}, and $E_{interval}$). Three of these 183 workloads use vanilla DPT (ResNet-18 on CIFAR-10, ResNet-34 on CIFAR-10, and ResNet-34 on CIFAR-100), and we use them as baselines for comparison. For all runs that use importance-aware DPT, we set E_{ignore} to 5. We use the same hyperparameters for all runs of vanilla DPT and importance-aware DPT, i.e., SGD with a 0.9 Nesterov momentum and a learning rate starting at 0.1 and weight decay (L2 penalty) of 0.0005.

Considerations for Randomness: The training process of DNNs is a stochastic one and is affected by many factors, e.g., choice of hyperparameters, stochasticity in the optimization algorithms, and the stochastic behavior of the tools, frameworks, and hardware used for training [15]. To better control for this stochasticity, each of the 183 workloads is repeated ten times using ten predetermined global random seeds. In Tables 1, 2, 3, 4 and 5, we report the average best test accuracy and standard deviation of ten runs for each workload. Also, the box plot of the performance of the top five settings of each table, alongside the performance of the corresponding baseline (vanilla DPT), is shown in Fig. 5.

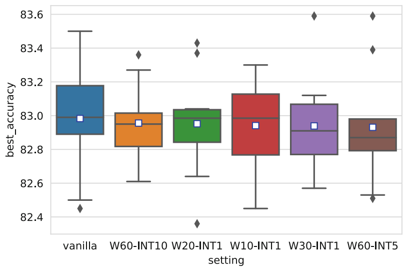
5.2 Different Dataset Complexities

We consider workloads of (ResNet-34, `Stripes`, Variance) with each of the CIFAR-10 and CIFAR-100 datasets. The results of the runs can be seen in Tables 4 and 5, and in Fig. 5 subfigures (4)–(5). CIFAR-10 and CIFAR-100 contain the same number of examples in train (50000 examples) and test (10000 examples) subsets, but they differ in the number of classes. CIFAR-10 has ten classes (5000 training examples per class), and CIFAR-100 has 100 classes (500 training examples per class). Hence, CIFAR-100 has a higher complexity than CIFAR-10 in terms of classes.

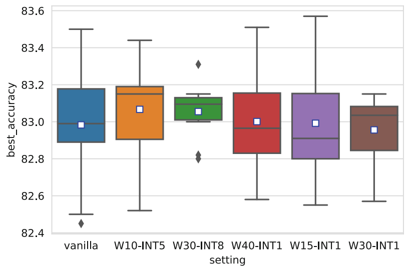
[2] See https://pytorch.org/vision/main/models.html.



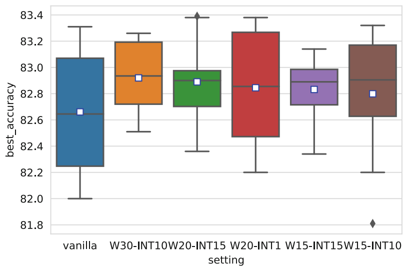
(1) CIFAR-10, ResNet-18, `Stripes`, Variance



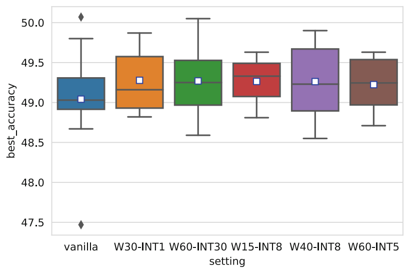
(2) CIFAR-10, ResNet-18, `Stripes`, Average



(3) CIFAR-10, ResNet-18, `Blocks`, Variance



(4) CIFAR-10, ResNet-34, `Stripes`, Variance



(5) CIFAR-100, ResNet-34, `Stripes`, Variance

Fig. 5. Box plots comparing the performance of the top 5 settings of E_{warmup} (W) and $E_{interval}$ (INT) for different combinations of (Dataset, Model, Partitioning Heuristic, Importance Metric). The leftmost box plot in each subfigure is the performance of vanilla DPT (baseline), and the other five box plots are ordered in decreasing average best test accuracy. The white square on each box plot denotes the average best test accuracy for a setting. Each subfigure (1)–(5) corresponds to a table with the same number, which contains the average best test accuracies and standard deviations over ten runs for each of the combinations of W and INT.

Table 1. Average best test accuracies (over ten runs) and standard deviations for different combinations of E_{warmup} (W) and $E_{interval}$ (I), when training ResNet-18 on CIFAR-10 with `Stripes` policy and loss variance as the importance metric. The baseline (using vanilla DPT) is 82.983 ± 0.327.

W	I					
	1	5	8	10	15	30
10	82.766 ± 0.185	82.848 ± 0.278	82.742 ± 0.152	82.862 ± 0.237	82.836 ± 0.387	82.988 ± 0.299
15	82.743 ± 0.373	82.752 ± 0.157	82.891 ± 0.302	82.888 ± 0.296	82.958 ± 0.247	82.873 ± 0.262
20	82.776 ± 0.243	82.832 ± 0.262	82.749 ± 0.309	82.722 ± 0.221	82.878 ± 0.283	83.044 ± 0.311
30	82.846 ± 0.202	82.858 ± 0.376	82.837 ± 0.263	82.946 ± 0.204	82.843 ± 0.307	82.773 ± 0.266
40	82.946 ± 0.246	82.773 ± 0.208	82.985 ± 0.238	82.869 ± 0.364	82.815 ± 0.296	82.827 ± 0.161
60	82.813 ± 0.283	82.898 ± 0.300	82.882 ± 0.152	82.764 ± 0.293	82.830 ± 0.249	82.705 ± 0.415

Table 2. Average best test accuracies (over ten runs) and standard deviations for different combinations of E_{warmup} (W) and $E_{interval}$ (I), when training ResNet-18 on CIFAR-10 with `Stripes` policy and average loss as the importance metric. The baseline (using vanilla DPT) is 82.983 ± 0.327.

W	I					
	1	5	8	10	15	30
10	82.941 ± 0.262	82.880 ± 0.339	82.859 ± 0.312	82.815 ± 0.290	82.836 ± 0.226	82.891 ± 0.195
15	82.885 ± 0.231	82.816 ± 0.287	82.841 ± 0.316	82.778 ± 0.259	82.866 ± 0.260	82.773 ± 0.247
20	82.952 ± 0.314	82.913 ± 0.247	82.903 ± 0.240	82.889 ± 0.265	82.841 ± 0.278	82.919 ± 0.210
30	82.939 ± 0.294	82.854 ± 0.185	82.853 ± 0.236	82.889 ± 0.227	82.743 ± 0.335	82.929 ± 0.279
40	82.864 ± 0.138	82.903 ± 0.152	82.883 ± 0.225	82.766 ± 0.220	82.905 ± 0.244	82.851 ± 0.236
60	82.908 ± 0.337	82.931 ± 0.339	82.818 ± 0.245	82.956 ± 0.228	82.806 ± 0.195	82.758 ± 0.237

The results show that there are several combinations of $(E_{warmup}, E_{interval})$ for training settings that can train better models than vanilla DPT. Thus, the gains of importance-aware DPT seem to hold across different datasets, given that we can find and select good hyperparameters for the training setting (e.g., E_{warmup} and $E_{interval}$).

5.3 Different Models

We consider workloads of (CIFAR-10, `Stripes`, Variance) with each of the ResNet-18 (18 layers, 8 residual blocks) and ResNet-34 (34 layers, 16 residual blocks) models [4]. The results of the runs can be seen in Tables 1 and 4, and in Fig. 5 subfigures (1) and (4). There are combinations of $(E_{warmup}, E_{interval})$ corresponding to each model that train better models than their corresponding baselines, but ResNet-34 shows to gain more from importance-aware DPT than ResNet-18.

Table 3. Average best test accuracies (over ten runs) and standard deviations for different combinations of E_{warmup} (W) and $E_{interval}$ (I), when training ResNet-18 on CIFAR-10 with `Blocks` policy and loss variance as the importance metric. The baseline (using vanilla DPT) is 82.983 ± 0.327.

W	I					
	1	5	8	10	15	30
10	82.921 ± 0.352	83.067 ± 0.270	82.778 ± 0.426	82.743 ± 0.218	82.662 ± 0.240	82.706 ± 0.165
15	82.992 ± 0.321	82.899 ± 0.308	82.890 ± 0.253	82.805 ± 0.165	82.664 ± 0.178	82.109 ± 0.338
20	82.845 ± 0.292	82.939 ± 0.376	82.850 ± 0.429	82.716 ± 0.205	82.747 ± 0.289	82.523 ± 0.165
30	82.956 ± 0.189	82.942 ± 0.309	83.055 ± 0.153	82.954 ± 0.382	82.815 ± 0.247	82.583 ± 0.206
40	83.001 ± 0.270	82.861 ± 0.336	82.786 ± 0.247	82.925 ± 0.18	82.865 ± 0.177	82.894 ± 0.254
60	82.918 ± 0.348	82.873 ± 0.283	82.848 ± 0.271	82.886 ± 0.273	82.884 ± 0.228	82.462 ± 0.222

Table 4. Average best test accuracies (over ten runs) and standard deviations for different combinations of E_{warmup} (W) and $E_{interval}$ (I), when training ResNet-34 on CIFAR-10 with `Stripes` policy and loss variance as the importance metric. The baseline (using vanilla DPT) is 82.661 ± 0.478.

W	I					
	1	5	8	10	15	30
10	82.650 ± 0.547	82.653 ± 0.399	82.590 ± 0.395	82.621 ± 0.243	82.751 ± 0.461	82.753 ± 0.632
15	82.537 ± 0.332	82.424 ± 0.510	82.745 ± 0.401	82.799 ± 0.481	82.832 ± 0.239	82.433 ± 1.020
20	82.845 ± 0.441	82.659 ± 0.637	82.787 ± 0.407	82.606 ± 0.541	82.890 ± 0.321	82.492 ± 0.300
30	82.671 ± 0.434	82.539 ± 0.307	82.719 ± 0.509	82.920 ± 0.287	82.594 ± 0.434	82.720 ± 0.589
40	82.669 ± 0.426	82.773 ± 0.403	82.422 ± 0.728	82.530 ± 0.305	82.649 ± 0.339	82.562 ± 0.353
60	82.789 ± 0.336	82.615 ± 0.342	82.683 ± 0.397	82.768 ± 0.525	82.678 ± 0.451	82.622 ± 0.661

5.4 Different Partitioning Heuristics

We consider workloads of (CIFAR-10, ResNet-18, Variance) with each of the `Stripes` and `Blocks` heuristics. The results of the runs can be seen in Tables 1 and 3, and in Fig. 5 subfigures (1) and (3).

The results show that for both heuristics, there are combinations of (E_{warmup}, $E_{interval}$) that can train better models than vanilla DPT. It is particularly interesting that training using the `Blocks` heuristic shows comparable performance to training with both `Stripes` heuristic and vanilla DPT.

5.5 Different Importance Metrics

With the loss values generated by each example in forward passes across several epochs as our importance measure, we evaluate the effects of the choice of two different metrics: *average loss* and *loss variance*. We consider workloads of (CIFAR-10, ResNet-18, `Stripes`) with each of the above metrics. The results of the runs can be seen in Tables 1 and 2, and in Fig. 5 subfigures (1)–(2). Loss variance as an importance metric performs marginally better than the average loss.

Table 5. Average best test accuracies (over ten runs) and standard deviations for different combinations of E_{warmup} (W) and $E_{interval}$ (I), when training ResNet-34 on CIFAR-100 with `Stripes` policy and loss variance as the importance metric. The baseline (using vanilla DPT) is 49.042 ± 0.698.

W	I					
	1	5	8	10	15	30
10	49.169 ± 0.335	49.064 ± 0.312	49.167 ± 0.432	48.758 ± 0.597	49.04 ± 0.503	49.033 ± 0.450
15	49.156 ± 0.332	48.959 ± 0.437	49.264 ± 0.292	49.186 ± 0.498	49.073 ± 0.573	49.079 ± 0.351
20	48.978 ± 0.550	49.144 ± 0.637	49.024 ± 0.365	49.149 ± 0.297	48.944 ± 0.436	48.977 ± 0.380
30	49.278 ± 0.399	48.906 ± 0.792	49.102 ± 0.393	48.897 ± 0.432	49.152 ± 0.446	48.966 ± 0.389
40	49.129 ± 0.549	48.978 ± 0.527	49.262 ± 0.489	49.155 ± 0.387	48.998 ± 0.450	49.024 ± 0.284
60	49.083 ± 0.348	49.224 ± 0.338	49.027 ± 0.453	49.194 ± 0.396	49.107 ± 0.461	49.270 ± 0.429

Table 6. Overhead statistics (in seconds) of importance-aware DPT when training ResNet-18 on CIFAR-10 with the different 36 combinations of E_{warmup} and $E_{interval}$.

Quantity	Min	Average	Max
Importance tracking overhead (each epoch)	0.979	1.052	1.407
Heuristic overhead (each interval)	2.456	2.643	5.213
Total training time	715	721.556	758

5.6 Added Overheads

The overheads of importance-aware DPT compared to vanilla DPT include (1) tracking importance data for each example at every epoch (a.k.a., importance tracking overhead) and (2) calculating the importance of examples and repartitioning the dataset based on heuristics at the beginning of each interval (a.k.a., heuristic overhead). In Table 6, we report the statistics on these overheads (in seconds) when we train ResNet-18 on CIFAR-10 for 100 epochs using four workers and the different 36 combinations of E_{warmup} and $E_{interval}$ (as reported in Tables 1, 2, 3, 4 and 5). The importance tracking overhead is independent of E_{warmup} and $E_{interval}$, as it happens at every epoch, and on average accounts for 14.57% of the total wallclock time. However, we should note that this is a prototype implementation of importance-aware DPT, and many optimizations can be made to significantly reduce the overheads (e.g., getting the individual example losses and the mini-batch losses in the same forward pass or using MPI operations for calculating the importance of examples). By only requiring repartitioning at every $E_{interval}$, importance-aware DPT has the potential to significantly reduce the network and I/O overhead that vanilla DPT requires for fetching examples at each epoch, especially in large training settings consisting of hundreds of thousands or millions of examples.

6 Conclusion

In this paper, we proposed importance-aware DPT, a data-parallel training approach for deep neural networks, that partitions the dataset examples across the workers based on a notion of the importance of each example. Our empirical evaluation across a number of well-known image classification workloads suggests that by setting relevant values for the hyperparameters of this approach, most notably E_{warmup} and $E_{interval}$, we can find better models (in terms of best test accuracy) compared to when training with vanilla DPT. Future work can concentrate on, e.g., using hyperparameter tuning methods for finding the best values for the hyperparameters of importance-aware DPT and evaluating the effects of different importance metrics and measures.

References

1. Arpit, D., et al.: A closer look at memorization in deep networks. In: International Conference on Machine Learning, pp. 233–242. PMLR (2017)
2. Chang, H.S., Learned-Miller, E., McCallum, A.: Active bias: training more accurate neural networks by emphasizing high variance samples. In: Advances in Neural Information Processing Systems, vol. 30 (2017)
3. Chitta, K., Alvarez, J.M., Haussmann, E., Farabet, C.: Training data distribution search with ensemble active learning. arXiv preprint arXiv:1905.12737 (2019)
4. He, K., Zhang, X., Ren, S., Sun, J.: Deep residual learning for image recognition. In: Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition, pp. 770–778 (2016)
5. Isola, P., Xiao, J., Parikh, D., Torralba, A., Oliva, A.: What makes a photograph memorable? IEEE Trans. Pattern Anal. Mach. Intell. **36**(7), 1469–1482 (2013)
6. Katharopoulos, A., Fleuret, F.: Not all samples are created equal: deep learning with importance sampling. In: International Conference on Machine Learning, pp. 2525–2534. PMLR (2018)
7. Krizhevsky, A.: Learning multiple layers of features from tiny images. Technical report, University of Toronto (2009)
8. Li, S., et al.: PyTorch distributed: experiences on accelerating data parallel training. Proc. VLDB Endow. **13**(12), 3005–3018 (2020)
9. Paszke, A., et al.: PyTorch: an imperative style, high-performance deep learning library. In: Advances in Neural Information Processing Systems, vol. 32 (2019)
10. Swayamdipta, S., et al.: Dataset cartography: mapping and diagnosing datasets with training dynamics. In: Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing (EMNLP), pp. 9275–9293 (2020)
11. Tang, Z., Shi, S., Chu, X., Wang, W., Li, B.: Communication-efficient distributed deep learning: a comprehensive survey. arXiv preprint arXiv:2003.06307 (2020)
12. Toneva, M., Sordoni, A., des Combes, R.T., Trischler, A., Bengio, Y., Gordon, G.J.: An empirical study of example forgetting during deep neural network learning. In: ICLR (2019)
13. Vodrahalli, K., Li, K., Malik, J.: Are all training examples created equal? An empirical study. arXiv preprint arXiv:1811.12569 (2018)

14. Yin, D., Pananjady, A., Lam, M., Papailiopoulos, D., Ramchandran, K., Bartlett, P.: Gradient diversity: a key ingredient for scalable distributed learning. In: International Conference on Artificial Intelligence and Statistics, pp. 1998–2007. PMLR (2018)
15. Zhuang, D., Zhang, X., Song, S., Hooker, S.: Randomness in neural network training: characterizing the impact of tooling. In: Marculescu, D., Chi, Y., Wu, C. (eds.) Proceedings of Machine Learning and Systems, vol. 4, pp. 316–336 (2022)