HetSeq: Distributed GPU Training on Heterogeneous Infrastructure

Yifan Ding, Nicholas Botzer, Tim Weninger
Department of Computer Science & Engineering
University of Notre Dame
Notre Dame, IN, USA
{yding4,nbotzer,tweninge}@nd.edu

Abstract

Modern deep learning systems like PyTorch and Tensorflow are able to train enormous models with billions (or trillions) of parameters on a distributed infrastructure. These systems require that the internal nodes have the same memory capacity and compute performance. Unfortunately, most organizations, especially universities, have a piecemeal approach to purchasing computer systems resulting in a heterogeneous infrastructure, which cannot be used to compute large models. The present work describes HetSeq, a software package adapted from the popular PyTorch package that provides the capability to train large neural network models on heterogeneous infrastructure. Experiments with language translation, text and image classification shows that HetSeq scales over heterogeneous systems. Additional information, support documents, source code are publicly available at https://github.com/yifding/hetseq.

Introduction

The AI community has witnessed rapid growth in the number of neural network models. Many of these models have matched or even exceeded human performance in a wide range of areas including image classification (Simonyan and Zisserman 2014; He et al. 2016), natural language processing (Vaswani et al. 2017; Devlin et al. 2018; Peters et al. 2018; Radford et al. 2019; Yang et al. 2019), go (Silver et al. 2017, 2018) and multiplayer online battle arena (MOBA) games like DOTA2 (Berner et al. 2019), and StarCraft II (Vinyals et al. 2019).

These increasingly powerful neural network models operate over extremely large training data and require millions (or even billions) of model parameters (Pudipeddi et al. 2020). This requires enormous computational resources that are only available to a handful of large organizations. Alongside the investment in hardware including CPU, GPU and high performance file systems, economies of scale present in only the largest organizations allow for the management and rapid development of new technologies like cloud computing (*e.g.*, AWS, Azure), tensor processing units (TPUs), half-precision computation (*e.g.*, Volta or Turing architectures of NVIDIA GPUs), and network architectures (*e.g.*, infiniband) for high performance communication between thousands of these hardware systems.

Smaller organizations like startups and universities, on the other hand, have relatively limited resources and typically purchase computing systems in an ad hoc manner – whenever funds allow. University computing resources are therefore much more heterogeneous in their composition compared to large technology companies and government labs. Furthermore, mismatches in memory capacity, network interface, and GPU/CPU capacity limit the development of large models. Given the system-homogeneity assumptions of deep learning software platforms, even the process of training an existing model on new data is difficult or impossible because the resource limits of a single node in a heterogeneous system limits the entire pipeline. With these limitations, training even moderately sized models could take weeks or even months to complete.

If not corrected, universities and other small organizations risk losing relevance in the race to develop newer and better machine learning models.

In the present work, we endeavour to level the playing field by adapting existing software platforms to train large neural network models on heterogeneous systems. We release our software package called HetSeq at https://github.com/yifding/hetseq, which is built on PyTorch and includes the common GPU environment and NVIDIA Collective Communications Library (NCCL) without any extra libraries or packages. In contrast, most existing distributed GPU training platforms (Sergeev and Del Balso 2018; Paszke et al. 2019) require extra packages like Docker and Open MPI, which may not be deployable over shared file system without administrative privileges.

We evaluate HetSeq using NVIDIA GPU nodes with various number of cores, memory capacity, and CPU architecture, while running in competition with the myriad of other machine learning projects at the University of Notre Dame. We perform experiments on translation, language modeling, and image classification tasks. We show vast improvements in training scalability using HetSeq without sacrificing model performance. Finally, the released project code covers detailed steps to install and execute. Examples of language translation, language modeling, and image classification tasks are included. Furthermore, developers and researchers can easily extend HetSeq to many other models with little effort.

Preliminaries

Many deep neural network (DNN) models are built upon popular platforms like TensorFlow (Abadi et al. 2016), PyTorch (Paszke et al. 2019) and Apache TVM (Chen et al. 2018a). A standard DNN model with backpropagation includes:

1. Model
2. Dataset and Dataloader
3. Optimizer and Learning Rate Scheduler
4. Checkpointing

A model can be described as a directed graph, where nodes represent parameters and edges represent the dependencies of the pipeline. A dataset is defined as the input of the model and the dataloader executes the data loading process from disk into memory. The optimizer plays a key role by updating the model parameters according to calculated gradients and learning rate generated by the scheduler. Finally, checkpointing is often used to store and load training snapshots.

In the typical case, after the model is defined and its parameters are initialized or loaded from a previous checkpoint, the indices of the dataset are loaded into memory. Because modern datasets are usually too large to fit into GPU memory all at once, the dataloader constructs smaller subsets of the dataset called batches to pass to the model one by one. Upon receiving each batch of data, the model performs a forward pass of the data over the model parameters and computes a loss function. Based on the results of the loss function, learning gradients are obtained by performing backpropagation. For each parameter, a pre-defined optimizer takes its gradient, the corresponding learning rate from learning rate scheduler, and other required factors to update the parameter. For a single batch of training data, this whole process including the forward pass, backpropagation, and the parameter update is called one step. One epoch is complete when all the batches have been processed over the entire training data.

Finally, once the training steps/epochs reach a certain threshold or other defined training criteria are reached (based on the objective function, learning rate, etc), we store a checkpoint. The checkpoint stores the model parameters as well as other necessary training status (like optimizer status, learning rate status, etc.)

Parallel Processing. The model training process is costly, especially using traditional CPU processing. Fortunately, most deep learning platforms support highly parallel GPU processing as well. There are three popular models for GPU parallelism: model parallelism (Shoeybi et al. 2019; Shazeer et al. 2018), pipeline parallelism (Harlap et al. 2018; Huang et al. 2019) and data parallelism (Tarditi, Puri, and Oglesby 2006; Dean et al. 2012). Model parallelism refers to splitting models into several parts where different parts are distributed to different devices (GPUs). In a backpropagation schema, intermediate output from the previous device is transferred to the next device in the forward step while the gradients of next device are transferred to the previous device during backpropagation. Model parallelism is essential especially for very large models which cannot fit the limited memory of a single GPU. However, heavy intermediate output and gradient communication may cause high latency. Pipeline parallelism is similar to model parallelism, instead of splitting the model into multiple steps, pipeline parallelism splits a single step into multiple parts. Data parallelism approaches split the training data into different parts to be distributed into different devices. Each device has its own model, data batch and optimizer thus performing forward, backward, parameter update individually. Recently, researchers from Microsoft have developed a combined platform using aspects from data parallelism, model parallelism, and pipeline parallelism to successfully train a trillion-parameter language model (Rajbhandari et al. 2019; Pudipeddi et al. 2020).

Heterogeneous Infrastructure. These state-of-the-art distributed parallel frameworks work well when the infrastructures are homogeneous, having the same memory capacity, GPU capacity and CPU throughput with high inter-node communication speed. Unfortunately, the business models of many smaller organizations necessitate the need for a more piecemeal approach to their system purchases. As a result, their computing infrastructure is heterogeneous, with many different types of systems purchased individually and without coordination. In this heterogeneous setting, the deep learning system cannot assume that each individual system, CPU, and GPU will have identical memory and throughput. Instead, communication across nodes with different networking infrastructure can be costly. Furthermore, the principles of model parallelism and pipeline parallelism cannot be easily applied because differences in memory capacity and throughput performance cause severe conflicts when reconciling the distributed computation. As a result not all models and training settings are compatible with certain systems and perform poorly on heterogeneous infrastructure.

The HetSeq package, described in the present work, is a distributed deep learning package that provides the capability to train large models on heterogeneous distributed infrastructure. HetSeq is adapted from the fairseq subpackage within PyTorch and uses principles of data parallelism to avoid heavy data communication so that each GPU process can execute expensive forward and backward passes and parameter updates locally. Other inter-node communication of the training loss, gradients, and parameters is managed carefully.

HetSeq: Distributed GPU Training on Heterogeneous Systems

HetSeq is designed to perform distributed data parallel GPU training across heterogeneous systems with large models and training datasets. Specifically, we modified modules to enable fast communication and avoid structural hazards like exceeding the memory capacity. In the HetSeq system implementation, each GPU contains an individual process and inter-process communication (IPC) is utilized. HetSeq is adapted from PyTorch [1], especially the DistributedDataParallel (DDP) mechanism[2], and fairseq [3].

[1]https://github.com/pytorch/pytorch

[2]https://pytorch.org/docs/stable/notes/ddp.html

[3]https://github.com/pytorch/fairseq

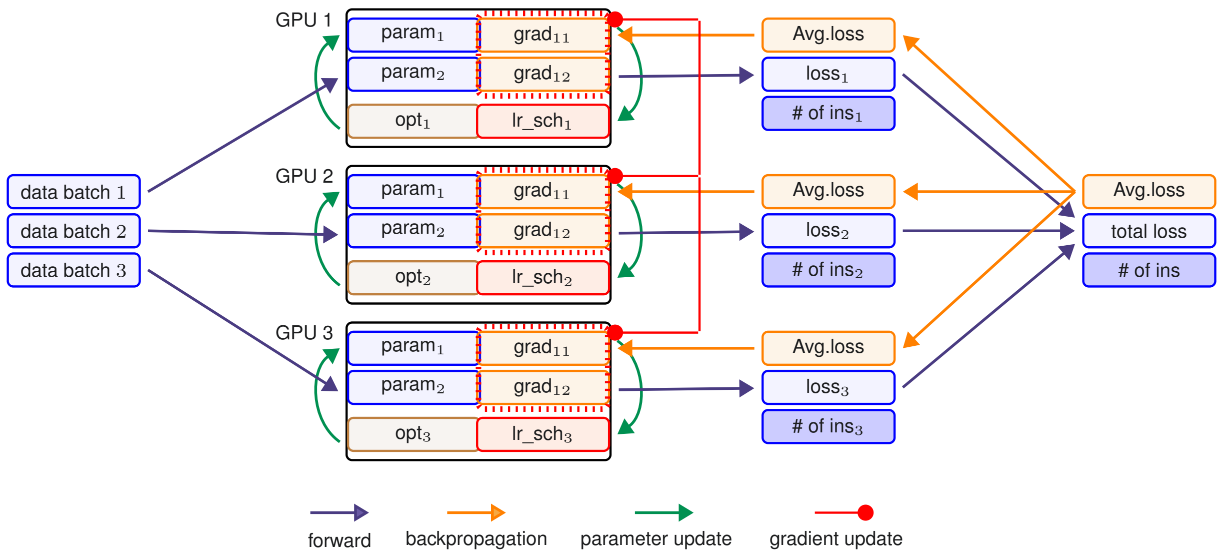

Figure 1: Distributed Data Parallel pipeline employed in HetSeq. The training can be divided into four major steps: the forward step, backpropagation, gradient update, and parameter update. Each GPU has individual models, optimizer (opt) and learning rate scheduler (lr_sch). Model, optimizer, and learning rate scheduler are all initialized with the same states while different GPUs process different segments of training data. In the forward pass, data batches assigned to different GPUs are loaded, the forward pass is executed and individual loss functions are computed. Different loss functions from different GPUs are aggregated to compute the average loss. In the backpropagation step, the average loss is distributed back to each GPU to calculate individual gradients. Gradient updates are then applied to communicate the gradients back to each GPU. Finally, the optimizer and learning rate scheduler of each GPU performs parameter updates individually.

As shown in Figure 1, different GPUs perform forward pass, backpropagation, and updating parameters in parallel. Individual GPU processes also communicate parameters, gradients, and loss functions. In the remainder of this section we introduce main aspects of HetSeq one by one and describe how they handle the complications that arise in the heterogeneous setting.

Model Initialization

We first initial the `ProcessGroup` by executing `init_process_group` function[4]. After the `ProcessGroup` is initialized, HetSeq initializes the model. A model is defined as a child class of `torch.nn.module`, which takes an input tensor (*i.e.*, images or sentences), and outputs a real number from the loss function. This model initialization is performed once on the master node and broadcast to all other GPUs so that they share the same initial state.

Dataset and Dataloader

The format of the input data varies widely depending on the application. Data access should support multiprocessing and multithreading, and the data access medium should support shuffling of the training instances in a way that can be easily stored and reproducible.

Unlike the typical training process on a single GPU, distributed training on multiple nodes with multiple GPUs has many challenges. When the dataset is small, then the system should just load it from disk into memory, and in each training step a chunk of the dataset is passed to the GPU. However, this is not feasible for even medium-sized datasets. In these typical cases, the dataloader is a bottleneck in training large models. Our solution is to separate the dataset into shards so that the dataset can be loaded in parallel. In addition to the dataset size and dataloader speed, the index sampler must be able to accommodate multiple dependent tensors with different data types. Simply put, our goal is to find a universal input and output mechanism that can quickly handle arbitrary, dependent tensors stored over multiple shards.

For this HetSeq uses HDF5 wrapped by h5py[5] packages as our main strategy to deal with the dataloading challenges. HDF5 supports self-describing and heterogeneous data at scale. Another benefit is that it can group multiple relative tensors together in a hierarchical manner so that it can be loaded faster with multiple processes and threads. We define our dataset class as a child module of `torch.utils.data.Dataset`. `__len__` and `__getitem__` functions must be implemented in the dataset definition, and the file `open` function must be defined inside the `__len__` and `__getitem__` functions instead of the `__init__` function to support multithreading loading in the PyTorch dataloader. In order to handle data shards, we add another class to accumulate the lengths of

[4]https://pytorch.org/docs/stable/distributed.html

[5]https://github.com/h5py/h5py

each file. The index of each training instance is mapped to an offset location at a corresponding shard.

Forward Pass

Having obtained the lengths and indexes of all shards, we generate sampler indices by forming batches that satisfy some criteria like maximum number of instances in a batch (*i.e.*, batch size) or maximum number of tokens in a batch. In a distributed training forward pass, each GPU has separate sampling indices according to its GPU index. Each GPU has an individual dataloader to load corresponding data by looking up sampling indices to form a batch. Once a data batch is ready, each GPU can immediately complete the forward pass and compute its individual loss function.

Optimizer and Learning Rate Scheduler

Each GPU has its own optimizer as well as a learning rate scheduler. Both are initialized with the same parameters. Different from the backpropagation process in a homogeneous system, heterogeneous GPUs, with different memory capacities, may require different batch sizes or a different number of tokens. So the loss functions of individual GPUs will likely have different weights.

During the last step of an epoch, GPUs may contain partially-filled batches and empty batches. For example, if there are 5 training instances and the batch size is set to 2 globally, we want to perform distributed training on 4 GPUs named A, B, C, and D, then the corresponding batch sizes should be 2, 2, 1, and 0 respectively, where batch C is half-filled (1/2) and batch D is empty (0/2). If we take the average or sum of the loss directly, then we will not compute the average loss in a way equivalent to the non-distributed setting. Instead, we augment the output by associating it with weights like batch size or number of tokens. After all GPUs send their output, we use a weighted sum to obtain the average. This task is performed by the master process. When complete, the master broadcasts the average loss to all the other processes.

Backpropagation

As soon as a GPU receives the average loss from the master, it can perform backpropagation to obtain the gradient for its individual model. However, because individual models stored on each GPU consider different data (in parallel), their parameters are likely to diverge. It is important to ensure that each parameter has the same partial derivative across GPUs. The `DistributedDataParallel` (DDP) class of PyTorch supports partial backpropagation and gradient synchronizations across GPUs; however, if one GPU has empty batch, we provide the GPU a dummy batch by copying its very first data batch and setting the gradient to 0 before backpropagation. After the backpropagation pass is complete, the calculated gradient is broadcast to each GPU. Then the GPU's optimizer retrieves the learning rate from the learning rate scheduler and updates the parameters in the model.

Checkpointing

In HetSeq, the master process is responsible for loading and storing checkpoints. In addition to include model parameters in checkpoints, we also need to consider: (1) the number of completed epochs, (2) the number of completed steps, (3) the optimizer status (including the learning rate scheduler status), (4) the random number seed, and several other settings.

Additional Considerations

Delayed Update (Gradient Accumulation) Compared to the forward pass, backpropagation is a relatively expensive process. It needs to compute the gradient for the whole model and update each parameter. Because of the difference in complexity, PyTorch recently implemented delayed updating, which aggregates the loss function computed from multiple forward passes before performing the backpropagation pass (Ott et al. 2018; Youkawa et al. 2018). However, when using delayed update the batch size is essentially scaled by the number of forward passes. Changing the batch size and data splitting is further complicated in the heterogeneous infrastructure because batch sizes may be different according to each node's capacity. Because batch sizes dramatically influence training performance, we need to consider the scaling effect of delayed update when computing the average loss function. In addition, carefully managing delayed update settings is important for reproducing model results.

Cython Compared to C/C++, Python is much slower. Performance differences are exacerbated when looping over large datasets and shuffling billions of training instances. HetSeq uses Cython and C++ bindings whenever possible for quicker runtime.

Prefetch and Cache Even though HetSeq achieves multi-processing and multi-threading on heterogeneous infrastructure, data loading is still a bottleneck before each forward pass, especially on larger batch sizes. We use prefetching and caching to reduce data loading latency. With prefetch, instead of loading every batch just before the forward pass, we fetch the next batch while training on the current batch. When memory capacity allows we can prefetch multiple batches. For caching, we utilize the least recently used (LRU) policy to store data in the memory. Although the LRU cache saves some disk access, the use of prefetch results in considerable performance gains because of the large size of most datasets.

Extending the HetSeq Package

The HetSeq package contains three major modules illustrated on the left in Figure 2: train.py, task.py, and controller.py to coordinate the main components illustrated on the right. The train.py module initializes the distributed system and its various components. The task.py module defines the model, dataset, data loader, and optimizer functions; it also executes the forward pass and backpropagation functions. The controller.py module acts as the main training controller. It executes the actual model, optimizer, and learning rate scheduler; loads and saves the checkpoint; communicates the loss; and updates the parameters.

To extend the HetSeq package to other tasks, the five components highlighted in red in Figure 2 need to be defined by using existing plug-ins or by defining customized

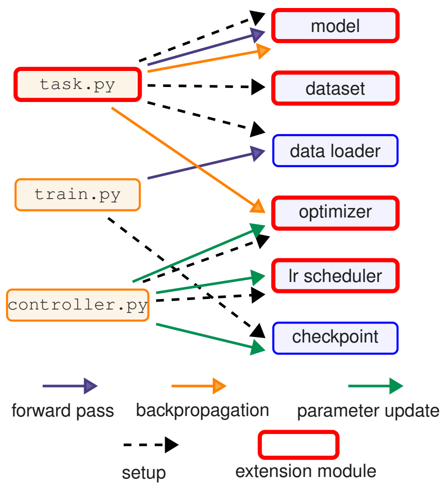

Figure 2: Module and component architecture of the HetSeq package. Python modules on the left are responsible for various components listed on the right. Modules and components highlighted in red can be extended to perform other machine learning tasks over heterogeneous infrastructure.

CPU			GPU			
Type	**Cores**	**Mem**	**Type**	**#**	**Cores**	**Mem**
Xenon	16/24	96/128GB	Xp	4	3840	12GB
Xenon	24	128GB	1080Ti	4	3584	11GB
Xenon	24	128GB	P100	4	3584	16GB

Table 1: Infrastructure used in experiments.

components. We also provide documentation on how to install, use, and extend HetSeq which is publicly available at https://hetseq.readthedocs.io.

Experiments

Here we test the performance of HetSeq on three popular deep neural network models: (1) the transformer translation model, (2) the BERT language model, and (3) an image classification model for MNIST handwritten digit database. We used a variety of different heterogeneous distributed setups. Described in Tab. 1, each GPU node has 4 GPUs consisting of a various number of cores and main memory. Experiments evaluate the training speed, scalability, and model performance across various configurations, specified in Tab. 2. Simply put, experiments with homogeneous regime use the same GPU nodes while heterogeneous settings use different combinations of different GPU nodes. For different heterogeneous settings, we keep the number of epochs constant while changing the number of steps on each GPU. For example, a single epoch with 16 steps on one GPU is equivalent to one step per GPU over 16 GPUs. All the other settings are set to the same for the same task.

Nodes	Translation Config.	BERT & MNIST Config.
1	1080Ti×1	P100×1
2 (hom)	P100×2	Xp×2
2 (het)	P100×1 + Xp×1	P100×1 + Xp×1
4 (hom)	P100×4	Xp×4
4 (het)	P100×1 + Xp×1 + 1080Ti×2	P100×2 + Xp×2
8 (het)	P100×1 + Xp×4 + 1080Ti×3	P100×4 + Xp×4

Table 2: Experiment Configurations

Transformer Translation Model

We evaluate HetSeq using the base Transformer model (Vaswani et al. 2017) on the WMT 2014 English-to-German translation task (En-to-De). The model has 6 encoder layers and 6 decoder layers. The size of the word embedding (*i.e.*, hidden state) is 512 and the number of heads is 8. Dropout rate is set to 0.1 and we use the label-smoothed cross entropy loss function with $\epsilon = 0.1$. In total, this model has about 65 million parameters. We use the Adam optimizer with $\beta_1 = 0.9$, $\beta_2 = 0.98$ and $\epsilon = 10^{-9}$.

Results. Table 3 shows the results of the transformer experiments on 2014 English-to-German dataset. We record the total training time and BLEU4 score (for the average and 1-, 2-, 3-, and 4-grams), which is a standard evaluation metric for language translation, for each experiment.

The heterogeneous configurations show that speedup scales at about one-half the linear rate, which can certainly be improved with further development. As the number of nodes increases from 1 to 2 (*i.e.*, 4 to 8 GPUs), the training time is sped up by a factor of 1.42. As more nodes are added, the performance improvement (*i.e.*, expansion) decreases to 0.6 over 8 heterogeneous nodes, but a nearly 5x speed up is achieved. This allows for the transformer to be trained in only 10 hours.

Critically, the performance of the heterogeneous configurations are rather similar to the performance of the homogeneous configurations. These results indicate that the performance of HetSeq does not deteriorate significantly in the presence of heterogeneous infrastructure - at least as compared to homogeneous infrastructure.

We also find that the BLEU4 score does vary for each task. Different experiments are conducted with the same optimizer and learning rate scheduling set ups but different batch sizes. Although the 2 node configuration resulted the best performance over the same number of steps, the model performance was relatively consistent across configurations.

In summary, we show that HetSeq can speed up training on the transformer model on heterogeneous infrastructure without sacrificing model performance.

BERT Language Model

The BERT language model (Devlin et al. 2018) masks some of the words in a sentence and tries to infer identities of the masked words using information from the unmasked words. We evaluate HetSeq using the base BERT language model trained on the Wikipedia corpus. We train base BERT using the Adam optimizer with $\beta_1 = 0.9$, $\beta_2 = 0.999$ and $\epsilon = 10^{-8}$. We use the linear decay learning rate scheduler

	nodes	GPUs	epochs	max tokens	steps per GPU	avg. step time	training time	BLEU4 score	expansion	speedup
Translator	1	4	128	4,096	260,000	0.62 s	49.47 hr	25.09, 56.8/30.8/18.9/12.0	1.00	1.00
	2 (hom)	8	128	8,192	130,000	0.90 s	34.23 hr	25.16, 57.1/30.9/18.9/12.0	0.73	1.45
	2 (het)	8	128	8,192	130,000	0.90 s	34.81 hr	25.57, 57.2/31.2/19.3/12.4	0.71	1.42
	4 (hom)	16	128	16,384	65,000	0.94 s	18.12 hr	24.98, 56.7/30.5/18.7/12.0	0.68	2.73
	4 (het)	16	128	16,384	65,000	0.94 s	18.74 hr	25.19, 57.1/30.9/18.9/12.1	0.66	2.64
	8 (het)	32	128	32,768	32,500	0.98 s	10.3 hr	18.74, 52.2/24.4/13.2/7.5	0.60	4.80

	nodes	GPUs	epochs	batch-size	steps per GPU	avg. step time	training time	training loss	expansion	speedup
BERT	1	4	5	128	267,139	2.60 s	7.19 d	0.026	1.00	1.00
	2 (hom)	8	5	256	133,570	2.69 s	4.19 d	0.028	0.86	1.72
	2 (het)	8	5	256	133,570	2.74 s	4.26 d	0.028	0.85	1.69
	4 (hom)	16	5	512	66,785	2.79 s	2.23 d	0.031	0.81	3.22
	4 (het)	16	5	512	66,785	2.81 s	2.19 d	0.031	0.82	3.28
	8 (het)	32	5	1024	33,393	3.13 s	1.21 d	0.055	0.74	5.94

	nodes	GPUs	epochs	batch-size	steps per GPU	avg. step time	training time	test loss	test accuracy	expansion	speedup
MNIST	1	4	20	256	47,00	0.00612 s	87.6s	0.0005	0.9918	1.00	1.00
	2 (hom)	8	20	512	2,360	0.01364 s	87.0s	0.0004	0.9911	0.50	1.01
	2 (het)	8	20	512	2,360	0.01406 s	87.7s	0.0004	0.9912	0.50	1.00
	4 (hom)	16	20	1024	1,180	0.01490 s	72.1s	0.0004	0.9912	0.30	1.21
	4 (het)	16	20	1,024	1,180	0.01473 s	72.6s	0.0005	0.9916	0.30	1.21
	8 (het)	32	20	2,048	600	0.01745 s	80.6s	0.0005	0.9909	0.14	1.09

Table 3: We evaluate HetSeq on Translation, BERT, and MNIST. Homogeneous (hom) and heterogeneous (het) experiments over 1-, 2-, 4-, and 8-node node configurations. HetSeq on heterogeneous configurations scales on the translation task at about the same rates as the homogeneous configurations. On 8 heterogeneous nodes, HetSeq achieves almost a 5x speedup on the Translation task and a 6x speedup on the BERT task without a significant loss in model performance. HetSeq does not show performance gains on the MNIST task because of the small number of training samples.

with maximum learning rate $= 0.0001$. In total, the model has over 1 billion parameters. We use the first 10K steps as the warm up and 1 million steps total over all GPUs.

Results. Runtime and training loss are described in Table 3. We show significant speedup as the number of GPUs and nodes increases. Despite fewer steps on each GPU, the training loss is maintained as the number of nodes increases.

Again we find that heterogeneous and homogeneous configurations achieve roughly the same speedup. However, it is important to note that in both the language model and the transformer experiments these runtime results are not apples-to-apples comparisons. For example, the P100s included in the heterogeneous configuration are faster than the Xps used in the homogeneous experiments. We do not intend the current work to be a full system analysis, but rather provide these runtime benchmarks as as evidence of HetSeq's scalability.

In summary, HetSeq is able to reduce the training time for the BERT language model on a single node from seven days to about one day on heterogeneous infrastructure.

MNIST Image Classification Model

To show the extensibility of HetSeq to other models, we also implement the image classification model from PyTorch and evaluate it within HetSeq. This model contains two layers of a convolutional neural network follows by a flat fully connected layer to perform image classification with a cross entropy loss (Simard et al. 2003). We use the Adam optimizer with $\beta_1 = 0.9$, $\beta_2 = 0.999$ and $\epsilon = 10^{-8}$. The starting learning rate is set to 1.01 for all the experiments.

Results. Runtime and accuracy on test set are described in Table 3. Compared to the translation and language models, the image classifier does not show good scaling with HetSeq due to the very small model and datasets. Specifically, MNIST has only $60,000$ training examples which we load directly into memory. Furthermore, the model requires fewer than $5,000$ training steps on single GPU, which is only a small fraction of the actual training time. Nevertheless, this task is useful because it shows how to extend HetSeq to other kinds of models. We expect that training over much larger datasets will more clearly reveal the benefits of HetSeq.

Discussion

The present work describes HetSeq, a publicly available deep learning platform adapted from PyTorch that enables distributed GPU training on heterogeneous infrastructure. HetSeq works by duplicating and distributing the model architecture to each GPU, which is its own process having its own optimizer, learning rate scheduler, data loader, etc. Each GPU communicates the loss and gradients while performs parameter update individually. Experiments on the transformer and BERT language model show that HetSeq can achieve reasonable speedup even over heterogeneous infrastructure.

HetSeq can be extended to incorporate other deep learning models in natural language processing, computer vision, and elsewhere. Future plans include adapting ongoing research in distributed optimization (You et al. 2019) to further improve training performance on heterogeneous infrastructure.

Acknowledgements

We thank Satyaki Sikdar for his help preparing this paper. This work is funded by the US Army Research Office (W911NF-17-1-0448) and the US Defense Advanced Research Projects Agency (DARPA W911NF-17-C-0094).

References

Abadi, M.; Barham, P.; Chen, J.; Chen, Z.; Davis, A.; Dean, J.; Devin, M.; Ghemawat, S.; Irving, G.; Isard, M.; et al. 2016. Tensorflow: A system for large-scale machine learning. In *OSDI*, 265–283.

Berner, C.; Brockman, G.; Chan, B.; Cheung, V.; Dębiak, P.; Dennison, C.; Farhi, D.; Fischer, Q.; Hashme, S.; Hesse, C.; et al. 2019. Dota 2 with Large Scale Deep Reinforcement Learning. *arXiv preprint arXiv:1912.06680*.

Chen, T.; Moreau, T.; Jiang, Z.; Zheng, L.; Yan, E.; Shen, H.; Cowan, M.; Wang, L.; Hu, Y.; Ceze, L.; et al. 2018a. {TVM}: An automated end-to-end optimizing compiler for deep learning. In *OSDI*, 578–594.

Chen, X.; Chen, D. Z.; Han, Y.; and Hu, X. S. 2018b. moDNN: Memory Optimal Deep Neural Network Training on Graphics Processing Units. *IEEE TPDS* 30(3): 646–661.

Dean, J.; Corrado, G.; Monga, R.; Chen, K.; Devin, M.; Mao, M.;Ranzato, M; Senior, A.; Tucker, P.; Yang, K.;2012. Large scale distributed deep networks. In *NeurIPS*, 1223–1231.

Devlin, J.; Chang, M.-W.; Lee, K.; and Toutanova, K. 2018. Bert: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805*.

Harlap, A.; Narayanan, D.; Phanishayee, A.; Seshadri, V.; Devanur, N.; Ganger, G.; and Gibbons, P. 2018. Pipedream: Fast and efficient pipeline parallel dnn training. *arXiv preprint arXiv:1806.03377*.

He, K.; Zhang, X.; Ren, S.; and Sun, J. 2016. Deep residual learning for image recognition. In *CVPR*, 770–778.

Huang, Y.; Cheng, Y.; Bapna, A.; Firat, O.; Chen, D.; Chen, M.; Lee, H.; Ngiam, J.; Le, Q. V.; Wu, Y.; et al. 2019. Gpipe: Efficient training of giant neural networks using pipeline parallelism. In *NeurIPS*, 103–112.

Ott, M.; Edunov, S.; Grangier, D.; and Auli, M. 2018. Scaling neural machine translation. *arXiv preprint arXiv:1806.00187*.

Paszke, A.; Gross, S.; Massa, F.; Lerer, A.; Bradbury, J.; Chanan, G.; Killeen, T.; Lin, Z.; Gimelshein, N.; Antiga, L.; et al. 2019. PyTorch: An imperative style, high-performance deep learning library. In *NeurIPS*, 8024–8035.

Peters, M. E.; Neumann, M.; Iyyer, M.; Gardner, M.; Clark, C.; Lee, K.; and Zettlemoyer, L. 2018. Deep contextualized word representations. *arXiv preprint arXiv:1802.05365*.

Pudipeddi, B.; Mesmakhosroshahi, M.; Xi, J.; and Bharadwaj, S. 2020. Training Large Neural Networks with Constant Memory using a New Execution Algorithm. *arXiv preprint arXiv:2002.05645*.

Radford, A.; Wu, J.; Child, R.; Luan, D.; Amodei, D.; and Sutskever, I. 2019. Language models are unsupervised multitask learners. OpenAI Blog 1(8): 9.

Rajbhandari, S.; Rasley, J.; Ruwase, O.; and He, Y. 2019. ZeRO: Memory Optimization Towards Training A Trillion Parameter Models. *arXiv preprint arXiv:1910.02054*.

Rhu, M.; Gimelshein, N.; Clemons, J.; Zulfiqar, A.; and Keckler, S. W. 2016. vDNN: Virtualized deep neural networks for scalable, memory-efficient neural network design. In *IEEE MICRO*, 1–13.

Sergeev, A.; and Del Balso, M. 2018. Horovod: fast and easy distributed deep learning in TensorFlow. *arXiv preprint arXiv:1802.05799*.

Shazeer, N.; Cheng, Y.; Parmar, N.; Tran, D.; Vaswani, A.; Koanantakool, P.; Hawkins, P.; Lee, H.; Hong, M.; Young, C.; et al. 2018. Mesh-tensorflow: Deep learning for supercomputers. In *NeurIPS*, 10414–10423.

Shoeybi, M.; Patwary, M.; Puri, R.; LeGresley, P.; Casper, J.; and Catanzaro, B. 2019. Megatron-lm: Training multi-billion parameter language models using gpu model parallelism. *arXiv preprint arXiv:1909.08053*.

Silver, D.; Schrittwieser J.; Julian S.; Simonyan K.; Antonoglou I.; Huang A.; Guez A.; Hubert T.; Baker L.; Lai M.; Bolton A.; et al. 2017. Mastering the game of go without human knowledge *Nature*, 550(7676), 354–359.

Silver, D.; Hubert, T.; Schrittwieser J.; Antonoglou I.; Lai M.; Guez A.; Lanctot M.; Sifre, L.; Kumaran D.; Graepel T.; et al. 2018. A general reinforcement learning algorithm that masters chess, shogi, and Go through self-play *Science*, 362(6419), 1140–1144.

Simard, P.Y.; Steinkraus, D.; and Platt, J.C. 2003. Best practices for convolutional neural networks applied to visual document analysis. In *ICDAR*, volume 3.

Simonyan, K.; and Zisserman, A. 2014. Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556*.

Tarditi, D.; Puri, S.; and Oglesby, J. 2006. Accelerator: using data parallelism to program GPUs for general-purpose uses. *ACM SIGPLAN Notices* 41(11): 325–335.

Vaswani, A.; Shazeer, N.; Parmar, N.; Uszkoreit, J.; Jones, L.; Gomez, A. N.; Kaiser, Ł.; and Polosukhin, I. 2017. Attention is all you need. In *NeurIPS*, 5998–6008.

Vinyals, O.; Babuschkin, I.; Chung, J.; Mathieu, M.; Jaderberg, M.; Czarnecki, W. M.; Dudzik, A.; Huang, A.; Georgiev, P.; Powell, R.; et al. 2019. Alphastar: Mastering the real-time strategy game starcraft ii. DeepMind blog 2.

Yang, Z.; Dai, Z.; Yang, Y.; Carbonell, J.; Salakhutdinov, R. R.; and Le, Q. V. 2019. Xlnet: Generalized autoregressive pretraining for language understanding. In *NeurIPS*, 5754–5764.

You, Y.; Li, J.; Reddi, S.; Hseu, J.; Kumar, S.; Bhojanapalli, S.; Song, X.; Demmel, J.; Keutzer, K.; and Hsieh, C.-J. 2019. Large batch optimization for deep learning: Training bert in 76 minutes. In *ICLR*.

Youkawa, T.; Mori, H.; Miyauchi, Y.; Yamada, K.; Izumi, S.; Yoshimoto, M.; and Kawaguchi, H. 2018. Delayed Weight Update for Faster Convergence in Data-parallel Deep Learning. In *IEEE GlobalSIP*, 663–667.