CS 744 Project: Efficient Distributed Transfer Learning using Pipelined Model Parallelsim

Daniel McNeela Mohil Patel Surabhi Gupta

Abstract

With an increase in deep neural network model sizes to many billions of parameters, it has become impractical and infeasible to perform re-training of such large models from scratch. Instead, a large body of work focuses on fine-tuning pre-trained models as new inputs arise and to specialize models to various tasks. In our project, we have explored one way to perform transfer-learning of three models simultaneously from a single pre-trained model in an efficient, distributed manner. We devise a way to perform this training using model pipeline parallelism while bypassing the weight staleness issues that are common in pipelined systems. Our results show significant reductions in both memory utilization and training time as compared to the baseline.

1 Introduction and Motivation

With the advent of very fast compute hardware and infrastructure [6,7], the pace of advances in machine learning and AI has skyrocketed. With this, the community has seen a rapid increase in the use of deep neural networks in all applications ranging from image classification, virtual assistants, chatbots, fake news detection, robotics, and healthcare to even music production and self-driving cars. It is no surprise that these models are comprised of millions of parameters and features. For instance, the recently released Chat-GPT has a whopping 175 billion parameters and was trained on nearly a trillion words!

With such large models, it is often not desirable or even feasible to re-train them very frequently. Instead, such models are often fine-tuned to perform specific tasks or re-calibrate them as new data arise. For this, a short training phase is performed on an existing model that has been pre-trained on large, application-related datasets to obtain general domain knowledge. This general idea of building upon pre-existing knowledge is called transfer learning [13].

In our class project, we explored one such transfer learning scenario where the goal is to fine-tune a pre-existing model into three separate models.

2 Background and Related work

2.1 Distributed Model Training

There are two key approaches to performing distributed deep neural network training: data parallelism and model parallelism. In this section, we describe each approach briefly.

2.1.1 Data Parallelism:

This approach is typically used when the training data is too large to fit inside the memory of a single GPU. The main idea here is to split the training data across multiple GPUs and perform training separately on each GPU. The weights are periodically synchronized and updated across all GPUs to reflect the training of the entire dataset together [9, 10].

2.1.2 Model Parallelism:

This approach is typically used when the model is too large to fit into the memory of a single GPU. Hence, the model is split across layers and each block of layers is sent to a different GPU. Pipelining is then used to train the model across all GPUs [4, 5, 12].

One of the major challenges with pipelined systems like PipeDream [12] is the issue of *weight staleness*, which can result in longer convergence times or even convergence failure. Weight staleness happens when the weights used for the forward pass of a batch are different from the ones available for the backward pass. This problem occurs because, in order to fill the pipeline, the next batch must be processed before the current batch is completed. This means that the weights used for the forward pass of a batch are already updated by the time the backward pass needs to be performed - this is what leads to the convergence issues described earlier. PipeDream addresses this issue by storing old weights, but this increases memory usage and reduces the benefits of model parallelism.

2.2 Transfer Learning

The main idea of transfer learning is that, instead of training a model from scratch with large amounts of data, models can learn to solve new problems with very few samples by leveraging previously trained models [2, 3]. Transfer learning formalizes a two-phase learning framework: a pre-training phase to capture knowledge from one or more source tasks and a fine-tuning stage to transfer the captured knowledge to target tasks.

There are essentially 4 broad categories to transfer learning models:

- Model fine-tuning: Here, a pre-trained model in one domain(source) is used as an initialization step for the training in the target domain. This initial model is fine-tuned to cater to the target domain.
- Feature-based methods: Here, the idea is to find a common feature space and transform the target domain features into the source domain features and then leverage the model trained for the source domain.
- Instance-based methods: Here, the source samples are reweighted so that the source domain data and target domain data share the same distributions.
- Model-based methods: The idea is to reuse pre-trained knowledge by distilling a larger trained model into a smaller model.

3 Idea

The setting in our project is to fine-tune a pre-trained model into 3 separate models. One basic approach is to copy the entire model into 3 different machines and have them perform the transfer learning for each model separately and independently.

Our key observation stems from the fact that for transfer learning, a big portion of the model is kept fixed - this means that this fixed part only undergoes forward computation, and no parameters are updated. Hence, there is no real need to perform this forward computation independently on 3 different machines as this leads to repeated work!

Our approach to avoid this repeated work is to provide a mechanism for the three machines to 'collaboratively' perform the forward computation and then independently train the trainable part of the model. This collaboration is achieved via pipelined model parallelism.

Figure 1 shows the system design. We split the base, pre-trained model into 4 parts - where each part comprises multiple layers. The first three parts are fixed and the 4th part is trained into three different versions. Thus, each machine is assigned one part from the fixed section and 1 version of the 4th part.

Each batch of data then moves between the machines in a pipelined fashion (each machine performing forward computation on it's part of the model). Once the last machine computes the forward tensors, it broadcasts the result to the others and they then proceed to compute the Forward + Backward computation on the trainable part (shown in green boxes in the figure). We perform 1-Forward and 1-backward scheduling to interleave the forward and training phases in the steady state of the pipeline.

3.1 Potential Benefits

Compared with the baseline (where each machine performs transfer learning of 1 model independently), this approach leads to both lower memory cost and lower training time. This is because of the following reasons:

1. *Memory:* Each machine holds two parts of the model (1 fixed part and 1 training part). In contrast, in the baseline setup, each machine would have all 4 parts of the model. Thus, this approach can, theoretically, reduce memory usage to half. In addition to this, the baseline setup would also have a copy of the dataset

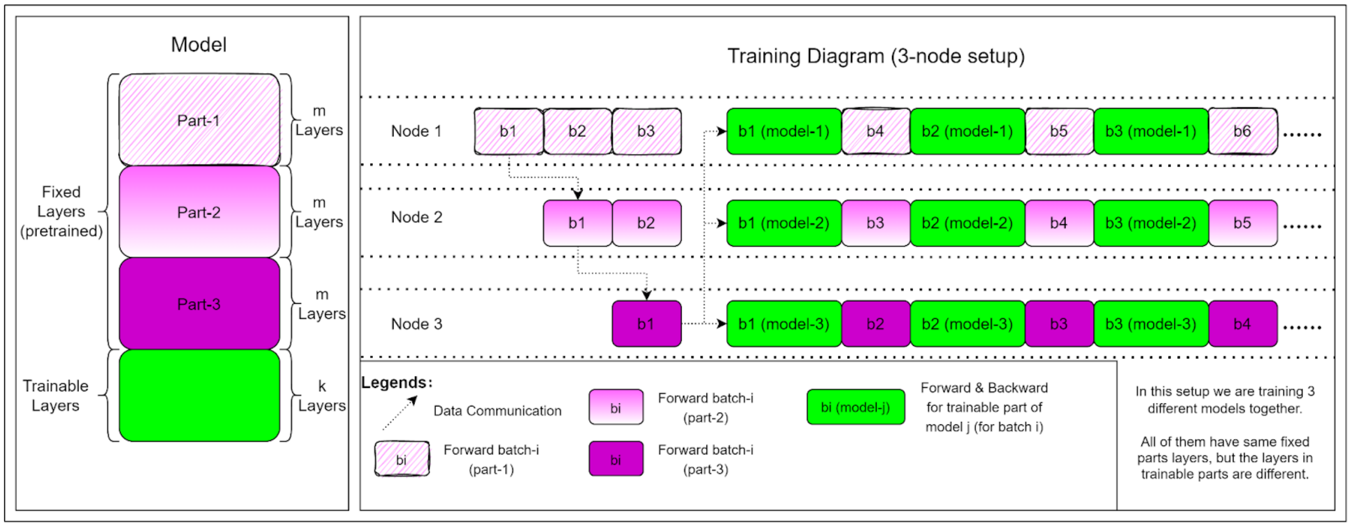


Figure 1: System Diagram

in all the machines. On the other hand, in the pipelined architecture, the input data will be present only in the first machine and only the forward computed tensors will be sent to the other machines!

2. *Train Time:* Pipelining the forward computation and the overlap between computation and communication leads to reduction in the overall training time. To understand this, consider the following: when batch 1 is undergoing forward computation on the second part of the model (on machine 2), machine 1 can start the forward computation on part 1 for batch 2. This parallel computation could lead to lowered training time!

   Another way to think about this is the following: in the baseline, each machine performs 5 units of computation for each batch (3 forward passes for the fixed model, 1 forward and 1 backward pass for the trainable part); in contrast, the pipelined setup only performs 3 units of computation (1 forward pass for the fixed model, 1 forward and 1 backward for the trainable part). Thus, we expect that the train time on each machine would roughly be 3/5 of the baseline setup!

3. *Staleness:* In pipedream [12], the pipelined setup led to weight staleness issue. This happened because the system there performed both forward and backward computations in the pipelined fashion. In contrast, our setting and approach does not lead to any staleness issues. This is because we do not perform the training computation as part of the pipeline (in the sense that this part happens independently across all machines) and only the forward computation is pipelined. This ensures that the backward phase for any batch will always get the correct version of the weights without the need for any more management on the algorithm front!

4 Implementation Details

We have used the PyTorch Distributed asynchronous communication library to implement the aforementioned pipelining infrastructure.

Our code has two major parts: implementing an async communication library between different machines and using this library to implement model training.

The library implements async communication using PyTorch's async isend and irecv operations to send forward tensors between nodes. Each machine maintains two circular queues: one for sending data and one for receiving it. The queue size is configurable, and when the queue is full, we block the operation until space is available. Increasing the queue size can lead to better computation and communication overlap. We also use PyTorch's async broadcast operation for broadcasting tensors to trainable parts (Figure 1, green component).

We use the communication library to implement model training. We have implemented a 1-Forward, 1-Backward schedule to keep the pipeline moving smoothly. The training data is loaded on node-1, and the forward tensors plus training labels are passed using the communication library to the other nodes for processing.

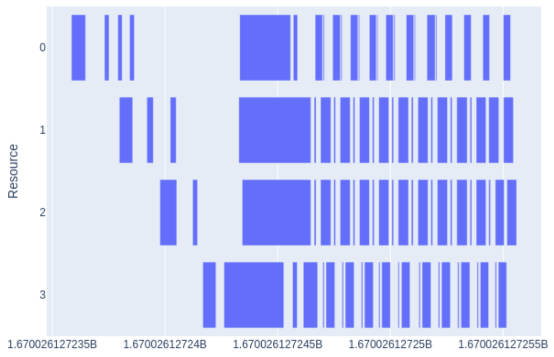


Figure 2: Pipeline Diagram

Figure 2 shows the pipeline diagram in one of the runs with a small number of batches. Each row (resource) represents a machine and the colored portion represents the time spent in computing the forward and training computations. Note that the training computation includes a forward pass, backward pass, and gradient updates - and is seen in the thicker sections of the plot.

We can see that the pipeline has big bubbles before reaching a steady state. This happens after the training phase for batch 1 has concluded. The uncolored sections show the bubbles that correspond to the time spent in communication. We can see the communication and computation overlapped in this plot!

5 Evaluation Methodology

To validate our hypothesis presented in section 3, we performed experiments to compare our pipelined design with the baseline. All this experimentation was done on a CloudLab cluster of 3 machines. For the baseline, we created a copy of the model on each of these machines and used dstat to measure the amount of memory and CPU used during the transfer learning computation. The same 3 machines were then re-used to perform the pipelined setup and we collected the memory and usage statistics as the experiment ran.

We used CloudLab [1] to run our experiments. To do so, we instanced multiple virtual machines (VMs) on the platform, each with 5 Intel Xeon Silver 4114 cores, 16 GB of RAM, and 40 GB of disk space.

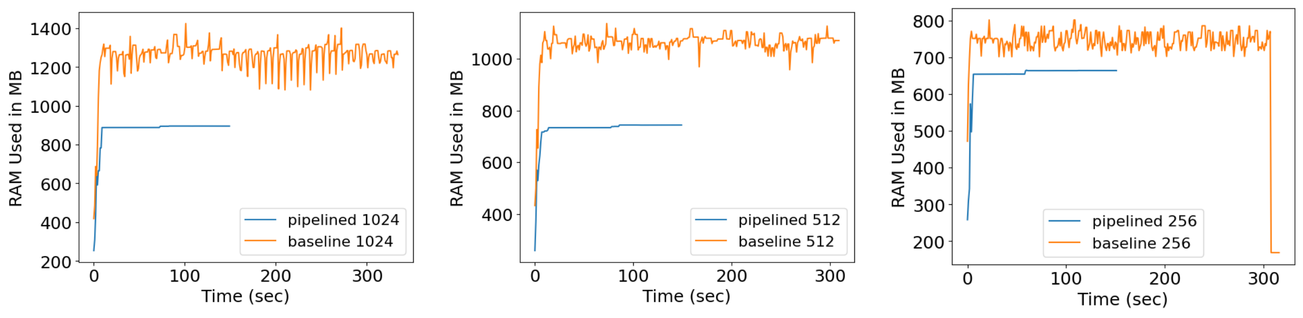
We used the pre-trained resnet50 model from torch vision [11] for our experimentations and the dataset, CIFAR10 [8], was also taken from the same torchvision library. The entire dataset consists of 50,000 images corresponding to ten different classes.

6 Experiments And Interpretations

6.1 Memory Usage

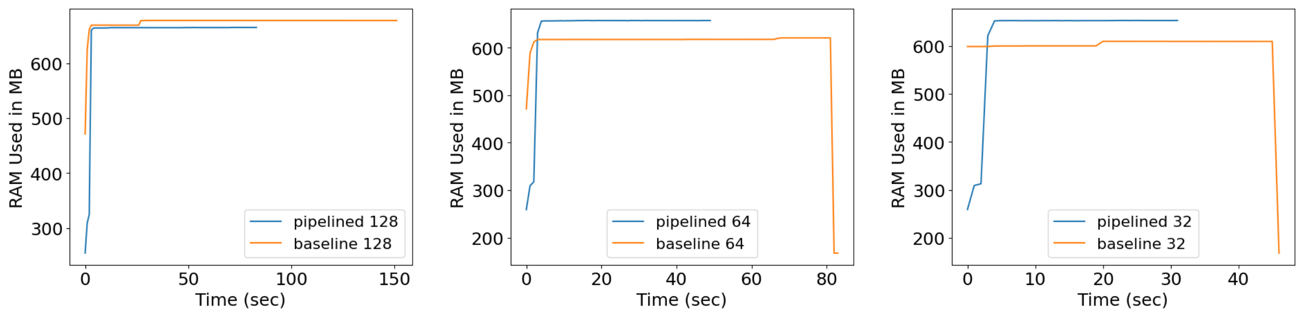
Figures 3 and 4 show the variation in the amount of memory used as the experiment progressed. The Y axis on each of these plots show the memory used in MB and the X axis plots the time in seconds. Each of these plots are for different experiments in which we varied the batch size from 1024 to 32. Note that the time taken to run both the baseline and pipelined versions increases as we increase the batch size. This huge variation is not really a fundamental phenomenon, but just an artifact of the way our experiments were set up - we fixed the number of batches across all experiments. Hence, a higher batch size simply means that the training is happening over a larger number of images.

Figure 3(a) helps validate our previous hypothesis! We see that the pipelined transfer learning took roughly half the time as compared to the baseline version (reflected in the X-axis). Also, the pipelined version uses significantly less memory than the baseline (reflected from the Y-axis). Figures 3(b), 3(c), 4(a), 4(b), 4(c) show the same plot for batch sizes 512, 256, 128, 64 and 32, respectively. We observe that the benefits of pipelining are more prominent with larger batch sizes. We think that this is because the overheads of network transfer and synchronization overshadow any benefits obtained when using smaller batch sizes such as 32 and 64.



(a) Batch Size 1024 (b) Batch Size 512 (c)Batch Size 256

Figure 3: Memory usage with Time



(a) Batch Size 128 (b) Batch Size 64 Batch Size 32

Figure 4: Memory Usage with Time

6.2 CPU Utilization

Figures 5 and 6 show the CPU Utilization plots for our implementation. The X-axis represents the time in seconds, and the Y-axis represents the CPU Utilization percentage. Similar to the memory usage plots, we varied the batch size from 1024 to 32 to generate different plots. The time taken for each batch size shows the same results as memory usage plots, i.e., the pipelined implementation takes roughly half the time of the baseline implementation.

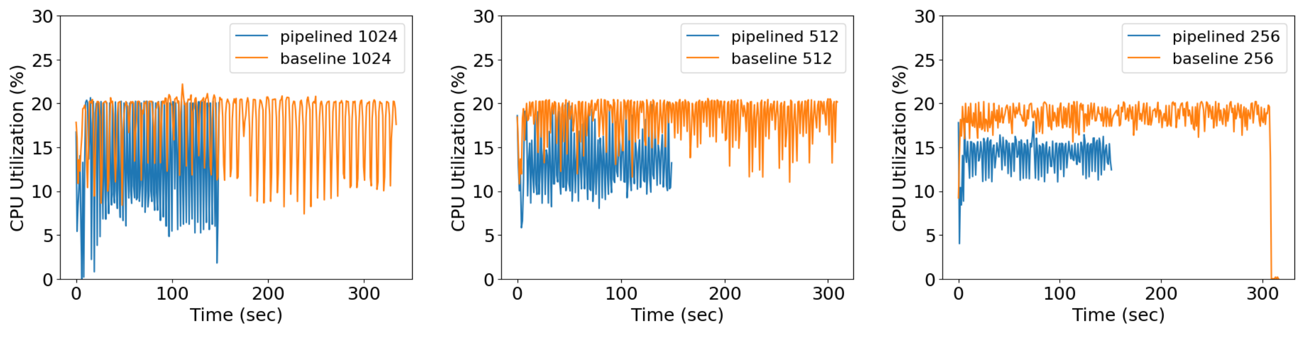
Regarding CPU Utilization, as we can see in Figures 5 and 6, the pipelined implementation shows slightly lower CPU utilization compared to the baseline. This behavior is due to the pipeline stalls or bubbles that can occur in the pipelined implementation, as shown in Figure 2. We can also see in Figure 5(a) that both the baseline and pipelined implementations show significant variations in CPU utilization, predominantly with larger batch sizes. This behavior could be due to higher memory latency or cache misses seen at larger batch sizes. Overall, the pipelined implementation shows good performance in terms of CPU utilization, despite the occasional stalls or bubbles.

7 Future Work

We think that the following directions could be taken to expand our preliminary work:

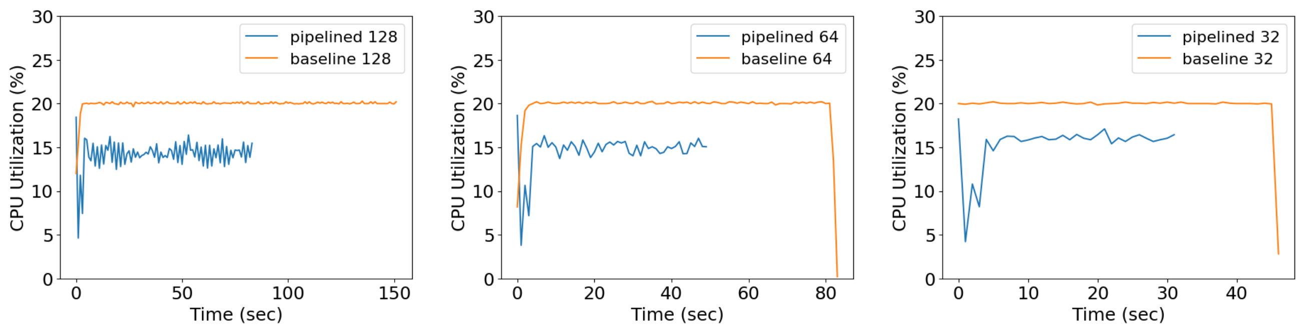
1. *Optimal Model Split*: In our current implementation, we have done a naive split of the model across the three machines - each machine gets roughly the same number of layers. We think that benchmarking the complexity of each layer and splitting them across machines to perform better load balancing could potentially improve memory usage, training times, and overall performance.

2. *Effect of queue sizes:* Another interesting direction is to study the effect of varying the queue sizes described in section 4. Our hypothesis is that larger queue sizes would allow for



(a) Batch Size 1024 (b) Batch Size 512 (c)Batch Size 256

Figure 5: CPU utilisation with Time



(a) Batch Size 128 (b) Batch Size 64 Batch Size 32

Figure 6: CPU utilisation with Time

more asynchronous communication opportunities. On the other hand, using a queue size of, say, 1 would mimic a synchronous pipeline without any overlap of computation and communication. However, a larger queue size might also lead to more overheads due to more open connections and memory required. Experimenting with various queue sizes to study this trade-off could be another interesting future direction!

3. *Better label propagation:* In our approach, we only need the input data to be present at node 0; however, the labels for training need to be present with all machines. Currently, we propagate the labels along with each forward tensor and then re-propagate them back to all machines with a broadcast. This adds some overhead that can be avoided by implementing a better label propagation mechanism that does not require each label to be transmitted twice!

8 Conclusion

In this project, we adapted pipelined machine learning training to a transfer learning setting where we fine-tuned pre-trained machine learning models by training only the last few layers. To speed up the forward tensor computation, we used the idea of pipelined model parallelism to split the model across different machines. And to avoid the weight staleness issue commonly seen in pipelined training, we trained 3 models together instead of training only 1 model. Additionally, we used 1-Forward, 1-Backward scheduling to keep the pipeline moving.

Our results show that the system finishes computation in roughly half the time of the baseline due to the benefits of pipelining. We also see an improvement in memory usage due to the model splitting. While we do see a slight reduction in CPU Utilization due to pipeline stalls, overall the pipelined implementation shows better timing and memory usage and provides significant benefits to training the models.

9 Contributions

During the ideation phase of the project, the teams worked together to develop the idea and plan the implementation. For the implementation, Daniel was responsible for setting up the pre-trained ResNet model, splitting it into multiple parts, and adding additional trainable layers. He also prepared the dataloader and training dataset. Mohil implemented the communication library, which allowed for efficient async communication between different nodes using circular queues. Surabhi worked on implementing a pipelined fashion ML model training using the communication library, and the ML model splits. She implemented the training logic with 1-Forward and 1-Backward scheduling, and added code to flow the data through the pipeline. Everyone worked collaboratively, helping each other debug the code and ensuring the system worked correctly and without deadlocks. For the experimentation phase, Mohil set up the CloudLab instances and prepared the baseline code for comparison. He also installed the code on the instances to collect data. Daniel collected and plotted the results for the baseline experiments, while Surabhi collected and plotted the results for the pipelined implementation.

References

[1] Dmitry Duplyakin, Robert Ricci, Aleksander Maricq, Gary Wong, Jonathon Duerig, Eric Eide, Leigh Stoller, Mike Hibler, David Johnson, Kirk Webb, Aditya Akella, Kuangching Wang, Glenn Ricart, Larry Landweber, Chip Elliott, Michael Zink, Emmanuel Cecchet, Snigdhaswin Kar, and Prabodh Mishra. The design and operation of CloudLab. In *Proceedings of the USENIX Annual Technical Conference (ATC)*, pages 1–14, July 2019.

[2] Saar Eliad, Ido Hakimi, Alon De Jagger, Mark Silberstein, and Assaf Schuster. Fine-tuning giant neural networks on commodity hardware with automatic pipeline model parallelism. In *2021 USENIX Annual Technical Conference (USENIX ATC 21)*, pages 381–396. USENIX Association, July 2021.

[3] Saar Eliad, Ido Hakimi, Alon De Jagger, Mark Silberstein, and Assaf Schuster. Fine-tuning giant neural networks on commodity hardware with automatic pipeline model parallelism. In *2021 USENIX Annual Technical Conference (USENIX ATC 21)*, pages 381–396. USENIX Association, July 2021.

[4] Jinkun Geng, Dan Li, and Shuai Wang. Elasticpipe: An efficient and dynamic model-parallel solution to dnn training. In *Proceedings of the 10th Workshop on Scientific Cloud Computing*, pages 5–9, 2019.

[5] Lei Guan, Wotao Yin, Dongsheng Li, and Xicheng Lu. Xpipe: Efficient pipeline model parallelism for multi-gpu dnn training. *arXiv preprint arXiv:1911.04610*, 2019.

[6] Qihang Huang, Zhiyi Huang, Paul Werstein, and Martin Purvis. Gpu as a general purpose computing resource. In *2008 Ninth International Conference on Parallel and Distributed Computing, Applications and Technologies*, pages 151–158, 2008.

[7] Norman P. Jouppi, Cliff Young, and et.al. Patil. In-datacenter performance analysis of a tensor processing unit. *SIGARCH Comput. Archit. News*, 45(2):1–12, jun 2017.

[8] Alex Krizhevsky, Geoffrey Hinton, et al. Learning multiple layers of features from tiny images. 2009.

[9] Mu Li, David G. Andersen, Jun Woo Park, Alexander J. Smola, Amr Ahmed, Vanja Josifovski, James Long, Eugene J. Shekita, and Bor-Yiing Su. Scaling distributed machine learning with the parameter server. In *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation*, OSDI'14, page 583–598, USA, 2014. USENIX Association.

[10] Shen Li, Yanli Zhao, Rohan Varma, Omkar Salpekar, Pieter Noordhuis, Teng Li, Adam Paszke, Jeff Smith, Brian Vaughan, Pritam Damania, and Soumith Chintala. Pytorch distributed: Experiences on accelerating data

parallel training. *Proc. VLDB Endow.*, 13(12):3005–3018, sep 2020.

[11] Sébastien Marcel and Yann Rodriguez. Torchvision the machine-vision package of torch. In *Proceedings of the 18th ACM International Conference on Multimedia*, MM '10, page 1485–1488, New York, NY, USA, 2010. Association for Computing Machinery.

[12] Deepak Narayanan, Aaron Harlap, Amar Phanishayee, Vivek Seshadri, Nikhil R. Devanur, Gregory R. Ganger, Phillip B. Gibbons, and Matei Zaharia. Pipedream: Generalized pipeline parallelism for dnn training. SOSP '19, page 1–15, New York, NY, USA, 2019. Association for Computing Machinery.

[13] Minghui Qiu, Peng Li, Chengyu Wang, Haojie Pan, Ang Wang, Cen Chen, Xianyan Jia, Yaliang Li, Jun Huang, Deng Cai, and Wei Lin. Easytransfer: A simple and scalable deep transfer learning platform for nlp applications. In *Proceedings of the 30th ACM International Conference on Information & Knowledge Management*, CIKM '21, page 4075–4084, New York, NY, USA, 2021. Association for Computing Machinery.