Efficient Training of Deep Learning Models for 3D Object Detection

Hemil Desai
UID: 405508015
`hemil10@ucla.edu`

Kriti Gupta
UID: 305491572
`kritigupta@ucla.edu`

Rohan Chetan Kapoor
UID: 305525510
`rohanck6399@ucla.edu`

Tanvi Bhandarkar
UID: 705525339
`tanvi17@ucla.edu`

Abstract

Self-driving technology has seen massive progress in the last decade. The amount of teams working on data collection and curation for self-driving cars has also increased tremendously, resulting in a variety of datasets ranging from object detection, semantic segmentation, lane markings, drivable area detection, multi-object tracking, etc. In particular, object detection is an interesting area of research geared towards detecting objects in the path of an autonomous vehicle but the efficient training of these models has always remained a challenge. In this project, we used the Lyft Level 5 3D object detection dataset to train end-to-end deep learning models for 3D Object Detection and explore optimizations within the training process. We integrate DeepSpeed and explore other techniques to enable efficient training. The central point of our project is to utilize multiple different optimizations ranging from AMP, FP-16 and offloading optimizers to CPU to removing data pipeline bottlenecks. We observe that these optimizations yield considerable speedups in both time and memory. We use PyTorch as the framework of choice for the project and run our experiments on Google Cloud Platform pre-emptive instances with one Nvidia v100 GPU.

1 Introduction

Self-driving technology presents a rare opportunity to improve the quality of life in many of our communities. Avoidable collisions, single-occupant commuters, and vehicle emissions are choking cities, while infrastructure strains under rapid urban growth. Autonomous vehicles are expected to redefine transportation and unlock a myriad of societal, environmental, and economic benefits. Level 5 comprises of completely autonomous vehicles with no human intervention. Perception is one of the cornerstone issue behind the slow development of self-driving vehicles. It can be defined as identification and classification of objects around the vehicle. Using information from car's sensors such as image and LiDAR data, it is possible to be able to accurately recognize objects in 3D space. For autonomous vehicles to work, it is very important for the perception component to detect the real world objects with both high accuracy and fast inference. In this project, we use the Lyft Level 5 3D object detection dataset to train end-to-end deep learning models to predict bounding boxes for the objects. Our work builds on the kaggle competition [1] hosted by Lyft a couple of years ago. Since the data contains millions of points and is of quite high resolution, efficient training of the model becomes very important. Thus, we reproduce and improve training efficiency for the MMDetection3D model. We have successfully integrated DeepSpeed [2] into the training process and tried out various optimizations. A comprehensive comparative analysis of the performance tradeoffs for these improvements is included in the report. PyTorch is the framework of choice for the project and we have run our experiments on Google Cloud Platform.

2 Data

2.1 Overview

The success of autonomous vehicles heavily relies on the algorithm's ability to handle complex environments, where accurate and robust perception is the foundation. To achieve this, autonomous vehicles are equipped with various sensors, including a camera, radar, and a lidar. In this, lidar is considered as the most critical one as it could provide accurate depth information and eventually performs better than image-based methods. Hence, lidar data became our choice of input.

We used the Lyft dataset in which each snapshot consists of two forms of information: image data and LiDAR data. The dataset contains data that is collected from a full sensor suite. Hence, for each snapshot of a scene, we are provided with references to a family of data that is collected from these sensors. Each of the host cars has seven cameras and one LiDAR sensor on the roof, and 2 smaller sensors underneath the headlights of the vehicle. The data consisted of 20 second long scenes. Each scene was made up of a number of samples. Samples in turn consisted of data from a number of sensors that has been captured at the same time. Each scene consisted of about 125 samples. The training set had annotations consisting of bounding volumes around all objects of interest in each sample. The objects of interest fell into nine classes. The classes were pretty imbalanced, there were more than half a million cars and fewer than 200 emergency vehicles. The training and testing data sets together contained more than 350,000 images and nearly 60,000 point clouds. Altogether the data sets used 120 gigabytes of disk space. We worked on the lidar data only. Since we were unable to download the entire dataset to our local machines we decided to use Google Cloud Platform (GCP) to import our data into an instance.

Lidar data has been proven to be a better alternative achieving higher accuracy than camera based approaches. The challenge with using lidar data is that it produces data in the form of point clouds which have millions of points thus increasing the computational cost and processing time. Each LiDAR sensor will shoot lasers 360 degrees to detect objects and get 3D spatial geometric information. These LiDAR sensors produce a point cloud each with 216,000 points at 10Hz. The data was captured across 13 files which served as inputs to our model. The primary measures captured by the LiDAR are x,y,z coordinates of an object along with its length, width, height and yaw. The metrics from the LiDAR sensors are also listed below with definitions for easy interpretation.

- centre_x, centre_y, and centre_z correspond to the coordinates of an object's location on the XYZ plane.
- yaw is the angle of the volume around the z-axis, making 'yaw' the direction the front of the vehicle/bounding box is pointing at while on the ground.
- length, width, and height represent the bounding volume in which the object lies.

2.2 Data Pipeline

Figure. 1 gives an overview of the data pipeline used to load the data for the model. We describe the main steps below.

Loading Data
The source of the scene data is from the 7 cameras and 3 LiDARs placed on the car. These images were recorded at different angles at a similar time. Information carried by point cloud captures surrounding objects in the form of x, y, z from the location of the sensor. We load Points in LiDAR coordinates from the file and update them by utilising previous sweeps. Then we add 3D bounding box and 3D label annotations.

Data Processing
We transform inputs to superimpose the data from all the 3 sensors to form a consolidated point cloud. To achieve this, we apply global rotation and scaling transformations to points and randomly flip them in horizontal and vertical direction. We then filter points and objects in a range.

Data Formatting
We format voxels and convert them to tensors and data containers. Finally, we collect points, 3D boxes and 3D labels from the loader.

Test Data Augmentation
We perform Test-time augmentation with multiple scaling, rotation and flipping transformations.

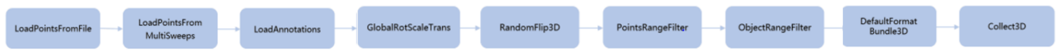

Figure 1: Data Pipeline

3 Model

In the real world the autonomous vehicles need to detect multi-class objects simultaneously, while the mainstream 3D detection frameworks often focus on single-category detection, such as car or pedestrian. In this way, to efficiently distinguish between heterogeneous categories plays an indispensable role in the success of multi-class 3D object detection. For this, we have investigated a models and architectures for their applicability and performance for 3D object detection and training on the Lyft Level 5 dataset.

Among models like VoteNet [3], PointPillars [4] we opted for Shape Signature Networks(SSN) [5] as it outperforms others in the aspect of a more salutary grouping of complex objects with similar sizes and shapes. This multi-modality quality thus demonstrates promising performance in discerning 3D objects.

3.1 Shape Signature Networks

The recent release of SSN model instantly gained attention in the field of 3D object detection and tracking. This prompt popularity is ascribed to their proposition of shape-aware heads grouping and shape signatures for better multi-class detection.

The shape-aware grouping heads bring the objects with similar shapes together, so as to share weights based on the object size. Heavier heads with more convolutions and large strides are designed for large objects while small heads for smaller objects. For instance, bus and truck need a heavier head than a car or bike.

The shape signatures aim to keep the shape information consistent, if not the same, within the identical category and separate the shape distributions across different categories, thus benefitting the feature capability of multi-category discrimination.

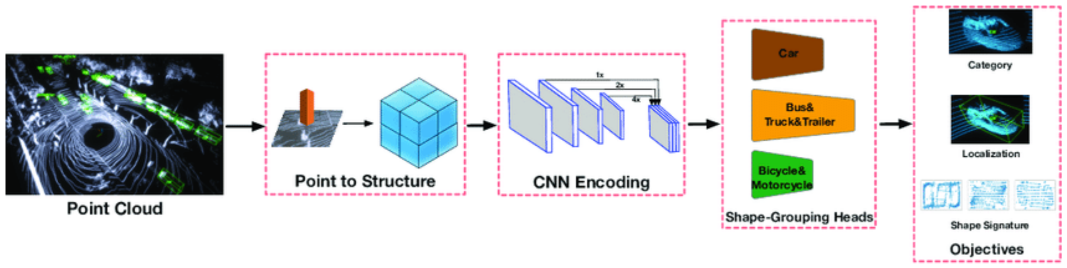

Figure 2: The SSN model Pipeline

3.2 Network Architecture

As shown by the red dashed-bounding box in Figure 2, the SSN network consists of four components:

- **Point-to-Structure:** It converts the raw point cloud points into the structured representations. Here, we have used Voxelization layer to transform points to voxels [6].

Type	voxel size	point cloud range	max #points	max voxels
Voxelization	[0.25, 0.25, 8]	[-100, -100, -5, 100, 100, 3]	20	(60000, 60000)

Table 1: Voxel Layer

- **Pyramid Feature Encoding**: The SSN takes the inspiration of feature encoding from the FPN [7] model. It implements the top-down convolution network to extract the features from the multiple spatial resolutions and fuses them through upsampling and concatenation.

Layer	Type	voxel size	point cloud range
scatter	DynamicScatter	[0.25, 0.25, 8]	[-100, -100, -5, 100, 100, 3]

Table 2: Voxel Encoder

Layer	Type	in features	out features
VFELayer	Linear	10	32
VFELayer	NaiveSyncBatchNorm1d	32	-

Table 3: VFE layer

Layer	Type	in channel	out channel	kernel	stride	padding	Activation
1	ConvTranspose2d	64	128	1	1	1	
	NaiveSyncBatchNorm2d	128					ReLU

Table 4: SECONDFPN

- **Shape-aware grouping heads**: As discussed above, it groups the objects with similar size and scale.

Layer	Type	in channel	out channel	kernel	stride	padding	Activation
1	Conv2d	64	128	3	2	1	-
	NaiveSyncBatchNorm2d	128	-	-	-	-	ReLU
2	Conv2d	128	128	3	1	1	-
	NaiveSyncBatchNorm2d	128	-	-	-	-	ReLU

Table 5: backbone network

- **Shape Encoding Objectives**: Here we perform the tasks such as multi-class classification, localization regression and shape vector regression on the shape vectors.

Layer	Type	in channel	out channel	kernel	stride	padding	Activation
1	Conv2d	384	128	3	1	1	
	BatchNorm2d	128					ReLU
conv-cls	Conv2d	64	36	1	1	1	
conv-reg	Conv2d	64	28	1	1	1	
conv-dir-cls	Conv2d	64	8	1	1	1	

Table 6: bounding box head

4 Experiments and Results

We performed several experiments to improve the efficiency of training the above mentioned Deep Learning models on the Lyft dataset [1]. We used the MMDetection3d library [8] for implementation of the model. Throughout our experiments, we used a Google Cloud Platform Compute Instance with 15GB RAM and one Nvidia v100 GPU. The instance was pre-emptible, meaning that it could be halted anytime within a 24 hour duration. To circumvent the halting problem of pre-emptible instances, we enabled regular checkpointing during training. Also, the pre-emptible instances provided us with cost savings of around 67%. We first describe the experiment setup in detail, and then move on to the experiments. Our code and implementation can be found in our Github repository [1]. The configuration files for models is also present in the repository.

4.1 Setup

Checkpoint We enable regular checkpointing to save the model weights. This allows us to keep track of the progress of model training. It also allows us to resume model training from the latest checkpoint if and when we need to restart the pre-emptible instance. In order to save as much progress as possible, we checkpoint the model at every 1000 steps (batches). Since the training in most cases involves around 226,800 steps, persisting every iteration would lead to exhaustion of the persistent disk space. As a result, we only keep the k latest checkpoints, with k=5 in most of our experiments.

Optimizer and Learning Rate We use the AdamW [9] optimizer with a learning rate of 1e-3 and a weight decay of 0.01. We use the WarmupLR schedule which basically increases the learning rate from 0 to maximum linearly upto the warmup steps and then continues at the maximum value. The maximum value is just the learning rate of 1e-3 we set for AdamW. We use 1000 steps for warmup.

Batch Size and Epochs Since we only have 1 GPU, at a time we are only able to process forward and backward pass for a maximum of 2 point clouds with sweeps. In order to reduce gradient update and optimizer step computations, we use gradient accumulation to accumulate gradients for n steps and then process the gradient and optimizer update. We use $n = 16$ for most of our experiments. We use 24 epochs for training unless specified otherwise.

4.2 Sweeps

One sample from a scene constitutes one data point as described earlier. Since the number of points in one point clouds is constrained, most approaches use sweeps to add more points. Sweeps have been used since the nuScenes dataset [10]. As the Lyft dataset is fairly similar to nuScenes, it makes sense to use sweeps for this task too. A sweep is basically just random sampling of points from frames around the sample frame. Since the scene has many frames per second, it makes sense to add more data points from frames around the sample frame.

Most approaches in [10] use 10 sweeps. However, we notice that using higher sweeps quickly becomes a bottleneck for model training. Hence, we experiment with reducing the sweeps and calculate the performance tradeoff of this choice. Due to compute and time constraints, we are only able to experiment with 0 and 5 sweeps apart from the 10 sweeps in the original. The results are summarized in Table. 7.

Sweeps	Public Leaderboard Score	Private Leaderboard Score	Training Time
0	0.113	0.111	21 hours
5	0.141	0.140	48 hours
10	0.175	0.174	92 hours

Table 7: Performance and Training times for different sweep configurations

As we can see, performance steadily improves with increasing the sweeps, but so does the training time. There is almost a direct correlation with increase in performance as well as an increase in training time. For compute constrained budgets, we propose experimenting and prototyping with lower sweeps and then using the higher number of sweeps for final training. This enables rapid prototyping as well as maximum performance extraction while keeping in mind the compute budget.

Another caveat we identified was that loading and processing the sweeps itself was taking a long time, so ideally we could save time and use the maximum number of sweeps if we pre-processed the sweeps

[1]`https://github.com/hemildesai/deepspeed_mmdetection3d`

and saved the data object as a python pickle file. But since the sweeps add a considerable amount of data points, storing the pickle files would take up huge amounts of disk space. Nevertheless, disk space is much more easily available than compute so it seems promising to pre-process the data beforehand for considerable speedups. Our repository contains a script to accomplish this pre-processing.

4.3 AMP and FP-16 Training

By default, training using PyTorch occurs in IEEE FP32 precision. Each floating point number is represented using 32 bits. While this provides a greater limit of numbers possible, it also takes up more memory. Many people have explored training in IEEE FP16 precision, which represents a floating point number using just 16 bits. Many deep learning models have been successfully trained in this precision with little loss in performance. However, there are certain caveats to consider. Since the limit of numbers possible via 16 bits is considerably small, there are higher chances of overflow and underflow while performing backward and forward passes. In order to prevent underflow, loss scaling is used during the backward pass. This essentially means just multiplying the loss with a scale in order to prevent underflow during gradient calculations, and then dividing the final result by the same scale. Setting a loss scale manually is challenging, hence people have come up with dynamic and automatic loss scaling. One such approach is highlighted in Zhao et al. [11]. We use the dynamic loss scaling provided by Apex [12] and implemented by Deepspeed. We use an initial scale power of 32, a loss scale window of 1000, a hysteresis of 2 and a minimum loss scale of 1.

Building on FP16 training, and to circumvent overflow and underflow, we also try Mixed Precision Training (Micikevicius et al. [13]). In particular, we use Automatic Mixed Precision (AMP) training provided by Apex [12]. This automatically selects which operations to convert to FP16 and which operations to keep to FP32. We use the O1 optimization stage of AMP. More details about the stages are available on their documentation site.

Additionally, we also force some operations to FP32 based on guidance from the MMDetection3d library [8] which we use for the implementation of our model. We summarize the memory, flops and training time for different precisions in Table. 8. Initial experiments showed that there was minimal effect on performance but we were unable to conduct a comprehensive evaluation due to the compute restrictions.

Precision	Memory (MB)	Memory Improvement	FLOPS	Training Time (5 sweeps)	Training Time Improvement
FP32	8760	-	1.53	58 hours	-
AMP O1	6061	**30.5 %**	1.53	49 hours	**15.5 %**
FP16	6027	**30.9 %**	1.59	48 hours	**17.2 %**

Table 8: Comparisons for different precisions

As we can see, both FP16 and AMP provide considerable improvements in both memory and training time for little loss in performance. The advantage of FP16 over AMP is minimal especially when you consider the stability provided by AMP. We recommend AMP as the default choice of mixed precision training for this task and also most tasks in general.

4.4 Zero-offload and Activation Checkpoints

Zero-offload The Zero Redundancy Optimizer (abbreviated ZeRO) (Rajbhandari et al. [14]) family has been a great step towards democratizing training of large deep learning models. In our particular case, since we're dealing with 1 GPU, Zero-Offload Ren et al. [15] is extremely relevant for reducing the GPU memory footprint thus allowing larger batch sizes and faster training. Zero Offload basically offloads the optimizer state and optimizer updates to the CPU. This greatly frees up the GPU memory. For an optimizer like Adam, it should free up about 3/5 of the initial GPU footprint. Moreover, the DeepSpeed implementation of CPUAdam is very efficient. DeepSpeedCPUAdam is 5x faster than Torch's implementation of CPU Adam.

Activation Checkpoints Checkpointing works by trading compute for memory. Instead of storing all activations during the forward pass, you can checkpoint some activations to be computed again

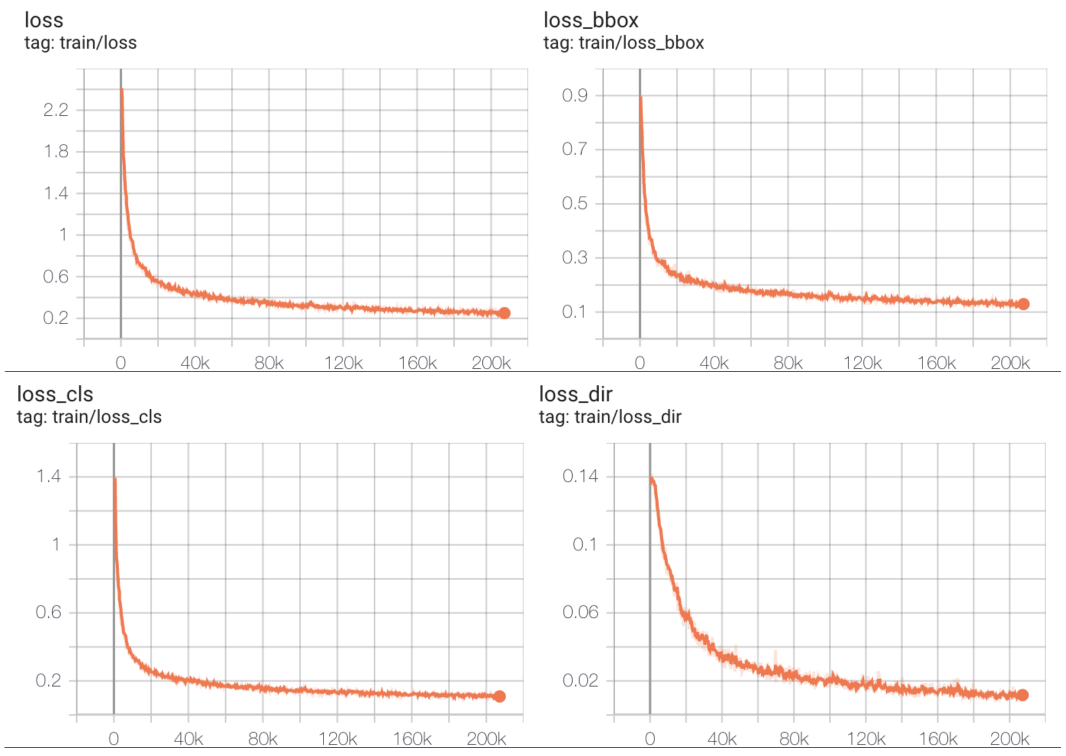

Figure 3: Training Loss over Steps

during the backward pass to gain further memory on the GPU. DeepSpeed checkpointing reduces the memory footprint at the expense of 33% additional computation. It also has the ability to offload checkpoints to the CPU instead of recalculating them. Combined with Zero-Offload, this can greatly reduce the GPU memory footprint.

Initial experiments allowed us to double the batch size at the very least. One constraint of offloading tensors to the CPU is the CPU Memory or RAM. If higher CPU RAM is available, more activations can be offloaded to the CPU for further space on the GPU. With higher RAM, it is also possible to use Zero Offload Stage 3 which also offloads some model paramters to the CPU along with the optimizer state. Trying out all different combinations is beyond the scope of this project due to compute limitations, but we plan to explore it in the future.

4.5 Logging

We utilize the tensorboard logger provided by MMDetection3d [8] and upload the logs to `https://tensorboard.dev` for persistence given the recurring restarts of the compute instance. Loss plots for a full training run over 2 days are shown in Fig. 3. Total Loss is the summation of the three part losses for bounding box, category and direction.

4.6 Visualization

The predictions of the trained model are visualized using MeshLab [16]. The point cloud is generated from the Lidar points of the original sample frame and the bounding boxes predicted by the 3D Detection Model. As seen in Figure. 4, the solid boxes represent the 3D objects and the points represent the original Lidar points from the sample frame used as the data point.

5 Future Work

Due to compute restrictions, we could only run our experiments on 1 GPU. In the future, we plan to incorporate efficient training on multiple GPUs. In particular, we aim to try out the following:

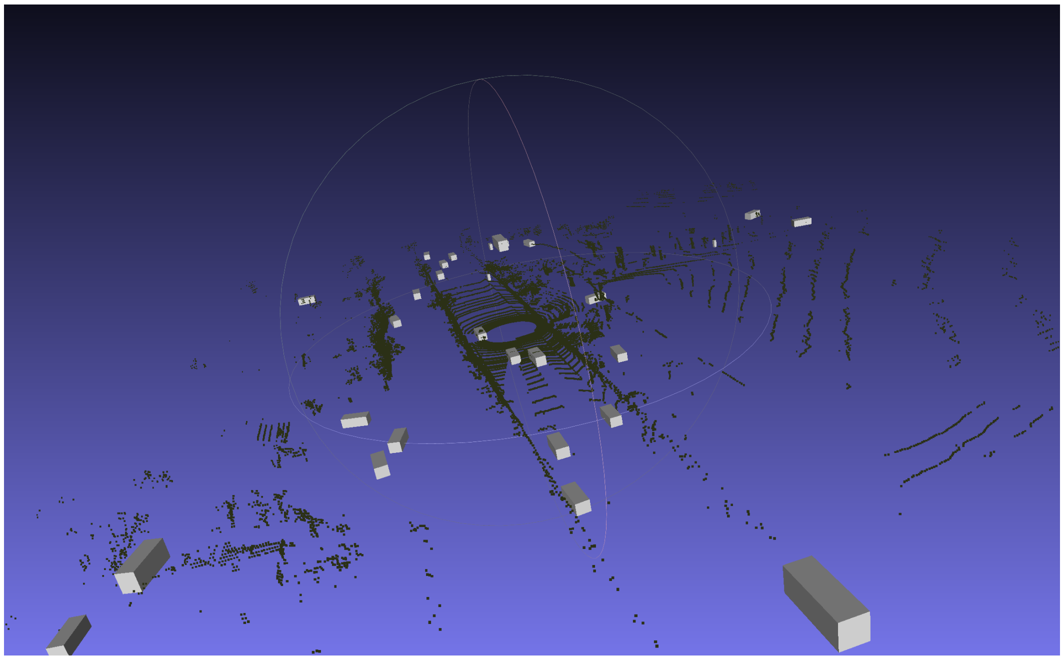

Figure 4: 3D point cloud visualization of predictions

5.1 ZeRO on multiple GPUs

The Zero Redundancy Optimizer (abbreviated ZeRO) (Rajbhandari et al. [14]) is a family of memory optimization technologies for large-scale distributed deep learning. Unlike data parallelism (that is efficient but can only support a limited model size) or model parallelism (that can support larger model sizes but requires significant code refactoring while adding communication overhead that limits efficiency), ZeRO allows fitting larger models in memory without requiring code refactoring while remaining very efficient. ZeRO does so by eliminating the memory redundancy that is inherent in data parallelism while limiting the communication overhead to a minimum. ZeRO removes the memory redundancies across data-parallel processes by partitioning the three model states (optimizer states, gradients, and parameters) across data-parallel processes instead of replicating them. By doing this, it boosts memory efficiency compared to classic data-parallelism while retaining its computational granularity and communication efficiency. For our implementation, we can configure training to use different stages of the Zero Optimizer ranging from 1 to 3 corresponding to the sharding of just optimizer state, optimizer state and gradients and optimizer state, gradients and parameters respectively. Subsequently, we can increase the batch size of 1 per device to ensure that the extra GPU memory is consumed for efficient training. ZeRO enabled would consequently allow the model to train smoothly on 8 GPUs without running out of memory.

5.2 1-bit Adam on multiple GPUs

The goal of this compression is to reduce the overall communication volume so that commodity systems with bandwidth-limited interconnects can be used to train large models. These challenges are addressed in DeepSpeed using a fully optimized 1-bit Adam implementation (Tang et al. [17]) for training on communication-constrained systems. **1-bit Adam offers the same convergence as Adam, incurs up to 5x less communication that enables up to 3.5x higher throughput for BERT-Large pretraining and up to 2.7x higher throughput for SQuAD fine-tuning.** It is worth mentioning that DeepSpeeds' 1-bit Adam optimizer scales so well on a 40 Gigabit Ethernet system that its performance is comparable to Adam's scalability on a 40 Gigabit InfiniBand QDR system. 1-bit Adam can be enabled by initialising three different parameters, viz. freeze_step, cuda_aware, and comm_backend_name in the optimization configuration file. For specifying freeze_step, we would need to empirically find the best step after which 1-bit compression needs to be applied to communication. Upon applying the aforementioned strategies we can add support for the 1-bit Adam feature for our integration of DeepSpeed with MMdetection3d.

6 Conclusion

We have studied and experimented with various optimizations for improving training efficiency of Deep Learning Models for 3D Object Detection. Most of them have been integrated using the DeepSpeed suite of tools. DeepSpeed is a revolutionary library with a great set of tools for every compute budget that can improve the training efficiency of most deep learning tasks. The optimizations via DeepSpeed can easily be generalized and implemented for any Neural Network training regime and we recommend doing so. Apart from these, we have also looked at task specific bottlenecks like data loading using sweeps and how those can be circumvented or made more efficient. Combining generalized optimizations along with task specific ones we can provide vast improvements for training neural networks, as seen in this task on 3D Object Detection.

References

[1] Lyft. Lyft 3d object detection for autonomous driving, 2020. `https://www.kaggle.com/c/3d-object-detection-for-autonomous-vehicles`.

[2] Microsoft. Deepspeed.ai, 2020. `https://deepspeed.ai`.

[3] Kaiming He Leonidas J. Guibas Charles R. Qi, Or Litany. Deep hough voting for 3d object detection in point clouds. *arXiv preprint arXiv:1904.09664v2*, 2019.

[4] Vora S. Caesar H. Zhou L. Yang J. Beijbom O. Lang, A.H. Pointpillars: Fast encoders for object detection from point clouds. *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, 2019.

[5] Tai Wang Yan Xu Jianping Shi Dahua Lin Xinge Zhu, Yuexin Ma. Ssn: Shape signature networks for multi-class object detection from point clouds. *arXiv preprint arXiv:2004.02774v1*, 2020.

[6] Oncel Tuzel Yin Zhou. Voxelnet: End-to-end learning for point cloud based 3d object detection. *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, 2017.

[7] Dolla r P. Girshick R. He K. Hariharan B. Belongie S. Lin, T.Y. Feature pyramid networks for object detection. *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2017.

[8] open mmlab. Mmdetection3d, 2021. `https://mmdetection3d.readthedocs.io/en/latest/`.

[9] Ilya Loshchilov and Frank Hutter. Decoupled weight decay regularization. *arXiv preprint arXiv:1711.05101*, 2017.

[10] Holger Caesar, Varun Bankiti, Alex H Lang, Sourabh Vora, Venice Erin Liong, Qiang Xu, Anush Krishnan, Yu Pan, Giancarlo Baldan, and Oscar Beijbom. nuscenes: A multimodal dataset for autonomous driving. In *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*, pages 11621–11631, 2020.

[11] Ruizhe Zhao, Brian Vogel, and Tanvir Ahmed. Adaptive loss scaling for mixed precision training. *arXiv preprint arXiv:1910.12385*, 2019.

[12] Nvidia. Nvidia apex, 2021. `https://nvidia.github.io/apex/`.

[13] Paulius Micikevicius, Sharan Narang, Jonah Alben, Gregory Diamos, Erich Elsen, David Garcia, Boris Ginsburg, Michael Houston, Oleksii Kuchaiev, Ganesh Venkatesh, et al. Mixed precision training. *arXiv preprint arXiv:1710.03740*, 2017.

[14] Samyam Rajbhandari, Jeff Rasley, Olatunji Ruwase, and Yuxiong He. Zero: Memory optimizations toward training trillion parameter models. In *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–16. IEEE, 2020.

[15] Jie Ren, Samyam Rajbhandari, Reza Yazdani Aminabadi, Olatunji Ruwase, Shuangyan Yang, Minjia Zhang, Dong Li, and Yuxiong He. Zero-offload: Democratizing billion-scale model training. *arXiv preprint arXiv:2101.06840*, 2021.

[16] Meshlab, 2021. `https://www.meshlab.net/`.

[17] Hanlin Tang, Shaoduo Gan, Ammar Ahmad Awan, Samyam Rajbhandari, Conglong Li, Xiangru Lian, Ji Liu, Ce Zhang, and Yuxiong He. 1-bit adam: Communication efficient large-scale training with adam's convergence speed. *arXiv preprint arXiv:2102.02888*, 2021.