Point-Voxel CNN for Efficient 3D Deep Learning in TensorFlow

Zach Ghera [1]

Abstract

There is increasing utilization of 3D data due to the rise of applications such as autonomous vehicles and AR/VR. 3D deep learning is relatively underdeveloped compared to the current state of 2D deep learning (i.e. computer vision) due to its exponentially larger memory and time requirements. Previous 3D deep learning models either use a point-based or voxel-based representation. The former suffers from inefficient computation due to random memory access while the latter suffers from cubically increasing computation and memory costs as the resolution increases. Point-Voxel CNNs were introduced to address this problem by combining the superior spatial locality and data-regularity of voxel-based models and the high-resolution, memory efficient point-based models to obtain a solution that is both memory and computationally efficient. We reimplement PVCNN in TensorFlow for the indoor scene segmentation task on the S3DIS dataset to enable other TensorFlow developers to run, extend, or deploy PVCNN. Inference evaluation showed a reduction of 4.5% and 37% latency and GPU memory, respectively, compared the original implementation. Training of the TensorFlow model yielded approximately a 35% mean IoU accuracy and 4x loss reduction despite being unable to finish one epoch.

1. Substantive Review and Critique

1.1. 3DShapeNets

1.1.1. Review

3DShapeNets[1] (Wu et al., 2015) is the first “3D deep learning model.” More specifically, 3DShapeNets is the first model to apply convolutional neural networks (CNNs) to a 3D representation. The other major component of 3DShapeNets is the creation of the ModelNet dataset, a collection of 3D CAD models to facilitate training and testing 3DShapeNets and future 3D deep learning models.

[1]I received permission from Dr. Inouye on 9/16 to use 3DShapeNets as one of my three papers to critique.

3DShapeNets can perform a variety of tasks. The model can perform object classification with both 3D (e.g. CAD model) and 2.5D (e.g. images from RGB-D camera) depth map/mesh inputs. To increase classification accuracy with 2.5D mesh inputs, the network can determine the next camera view that, when combined with the initial view, will have the greatest decrease in recognition uncertainty. Additionally, 3DShapeNets can construct potential 3D shape completions from a single (2.5D) depth image. Previous methods for shape reconstruction involve either “assembly-based” (Wu et al., 2015) or smooth interpolation and/or extrapolation approaches. These rules-based methods are limited to a specific set of shapes and require handmade templates. Deep learning approaches for classifying and constructing shapes as well as “next-best-view” problems existed prior to 3DShapeNets, but only involving 2D inputs.

3DShapeNets represent 3D shapes with a 3D voxel grid where each 3D point is represented as a binary tensor. Each point in the grid indicates whether that voxel lies within the observed object, outside of the object, or is unknown. This 3D voxel representation is fed as input to a Convolutional Deep Belief Network (CDBN). 3DShapeNets’ CDBN adapts the standard Deep Belief Network (Hinton et al., 2006) network (DBN) to perform 3D convolution on the 3D voxel grid as opposed to a fully connected layer for a 2D image. In addition to using a 3D kernel, the CDBN differs from typical CNN models by excluding pooling operations, as reducing the dimensionality increases uncertainty for shape reconstruction tasks.

1.1.2. Critique

One of the strengths of the 3DShapeNets paper is the high-level explanations of topics. This was demonstrated with the informative and straightforward figures that supported discussion related to processing information in 2.5D and 3D space. One specific example of this is Figure 4, where the authors visualize a potential scenario and set of possible candidates for the “next-best-view prediction” (Wu et al., 2015). Additionally, the authors provided a clear connection between each of the sub-problems that 3DShapeNets solve: object classification, next-best-view, and object completion.

The primary weakness of the 3DShapeNets paper is incomplete nature of low-level details in the paper. The remainder of this section includes specific observations and examples for the 3DShapeNets authors that illustrate this claim.

In Sec 3, the authors define the energy of a convolutional layer in the 3DShapeNets CDBN. What this energy value represents in the network when compared to typical CNNs is not clear. Is this the pre-non-linear activation value for a single output voxel? Or does this represent some other quantity?

After defining the energy of a convolutional layer, the authors detail the architecture of a CDBN. While the authors clearly state that the top layer "forms an associative memory DBN," it is left unclear what the "multinomial label variables and Bernoulli feature variables" represent with respect to the model as a whole (Wu et al., 2015).

In Sec 6.1, the authors detail an experiment to quantify the performance of 3DShapeNets for a 3D shape classification and retrieval task. While this experiment provided useful insights to the abilities of 3DShapeNets, this section felt out of place as 3D shape classification and retrieval tasks were not described in a previous section.

1.2. PointNet

1.2.1. Review

PointNet (Qi et al., 2017a) is the first deep neural network architecture designed to process 3D point cloud data as an unordered set of multi-dimensional (3D coordinate plus any additional feature channel) points. Pointnet was designed to perform a variety of applications including shape classification, part segmentation, and semantic segmentation.

Previous notable state of the art methods that perform classification/segmentation tasks on 3D point cloud data using deep learning techniques require transforming the point cloud data to large data formats such as 3D voxels (e.g. 3DShapeNets (Wu et al., 2015)) or a collection of images before they are input to a known CNN architecture. This is necessary as typical CNNs architectures assume input tensors with spatially ordered elements.

PointNet is shown to have equal or better accuracy compared the previous state of the art methods with drastic improvements in time and space complexity owed to its sparse representation of 3D data. More specifically, PointNet is 8 and 141x more efficient computationally than Subvolume (Qi et al., 2016) and MVCNN (Su et al., 2015) networks as well as 17x more space efficient (less network parameters) than MVCNN.

PointNet identifies and addresses three challenges of using raw 3D point cloud data as input to deep neural networks: permutation invariance, point relationships, and transformation invariance.

Permutation Invariance Point clouds are a set of unordered multi-dimensional points. Thus, a network must be invariant to all possible permutations of the input set.

PointNet addresses unordred input using symmetric functions. More specifically, PointNet found that the max pooling operation is a simple but effective way to obtain a global feature vector from the transformed (Sec 1.2.1), unordered input data.

Point Relationships Neighboring points often compose sub-structures that make up features in the overall scene. Classification tasks solely rely on global features whereas segmentation tasks must leverage local and global point features.

The global feature vector composed from the output of the max pooling layer can be directly fed to a fully-connected network for classification tasks. For segmentation tasks the global feature vector is concatenated with each local feature vector and fed into another fully-connected network to determine new per point features that rely on global and local information.

Transformation Invariance Objects or scenes may undergo affine transformations such as rotation or translation. The model must be invariant to these transformations for both classification and segmentation tasks.

PointNet addresses this with a transformation sub-network whose goal is to learn a transformation matrix that aligns the input data in a known format for feature extraction. This idea was motivated by Spatial Transformer Networks (Jaderberg et al., 2015).

1.2.2. Critique

One of the primary strengths of this paper is clear but detailed explanations of complex topics. For key aspects of the design, the authors provided theorems and rigorous proofs that supported their approach as well as a summary of key takeaways from a high level view. For example, Sec 4.3 explains how the the network is robust to input noise due to its ability determine a set of "critical points" that characterize objects. One exception to this claim, however, was the explanation of the transformation networks (T-nets) in Supplementary section C. Although they discussed the network architecture in the second paragraph, any additional details or a figure would aid in efforts to reproduce T-nets.

The remainder of this sections includes identified gaps and missing details.

The paper explored different alternatives for symmetry functions. However, the authors do not elaborate on any experiments conducted with other layer types for transforming inputs to higher dimensional space other than fully-connected layers (e.g. 1x1 convolutional layers with multiple feature maps).

In the Introduction, the authors write "Effectively the network learns a set of optimization functions/criteria that select interesting or informative points of the point cloud and encode the reason for their selection"(Qi et al., 2017a). They explained how the network can extract a "critical point set" given by the K points in the bottleneck dimension. However, there was no explanation as to how the network "encodes a reason for selection." This should be elaborated upon.

In the "Joint Alignment Network" subsection of Sec 4.2 the authors write, "An orthogonal transformation will not lose information in the input, thus is desired"(Qi et al., 2017a). The authors fail to substantiate or qualify this claim and the implication to non-orthogonal transformations.

1.3. Point-Voxel CNN for Efficient 3D Deep Learning

1.3.1. Review

Point-Voxel CNNs (Liu et al., 2019) present a solution for accurate and efficient 3D deep learning. This is achieved by combining the good spatial locality and data regularity of voxel-based models and the high-resolution, memory efficient point-based models to obtain a solution that is both memory and computationally efficient. Point-Voxel CNNs can perform a variety of tasks including part segmentation, semantic scene segmenation, and 3D object detection.

The primary drawback with voxel-based 3D deep learning models is the cost for high-resolution voxels. That is, convolution computation and memory cost both increase cubically with higher resolution voxel grids. As a result, voxel-based models must sacrifice resolution in order to fit in memory on a GPU. For example, the authors mention that on a GPU with 12GB of memory, the "largest affordable resolution will lead to 42% information loss" (Liu et al., 2019). However, volumetric convolutions with voxels facilitate high spatial locality which lead to more efficient hardware usage.

Conversely, point-based models achieve very low memory usage as they use a sparse representation of 3D point clouds. However, because this representation uses unordered sets, neighboring points are not contiguous. As a result, identification and indexing of neighbors for a specific point (for convolution) leads to random memory access. It can be shown that the fraction of time spent on random memory access accounts for "55% to 88%" (Liu et al., 2019) of all computations.

Point-Voxel CNNs introduce a new convolution method, "Point-Voxel Convolution (PVConv)," (Liu et al., 2019) to that combines the strengths of voxel-based and point-based models. PVConv layers contain two groups of computation and then fuses the results. One branch clusters neighboring points to extract local relationships between points. This aggregation is performed using volumetric convolutions with a low-resolution voxel-grid representation of some points to take advantage of spatial locality for an operation that does not need high-resolution. The other branch determines individual point features using a fully connected network. The outputs from each branch are fused to obtain features for individual and groups of points in high resolution.

1.3.2. Critique

The primary strength of this paper is the clear explanations of complex topics and analysis of their approach. The authors demonstrated this with the detailed analysis of the strengths and weaknesses of the voxel-based and point-based approaches. Sec 4 on the Point-Voxel Convolution was also very straightforward while providing key figures, equations, and references required to gain a good understanding of their approach. The authors also provided a plethora of experiments and results to demonstrate the computational and memory gains of Point-Voxel CNNs compared to the previous state of the art voxel and point based networks.

The primary weakness of this paper was their lack of detail for their implementation. While Sec 4 provided a clear description of each of the steps for a PVConv operation, there was no section dedicated to the model architecture. Additionally, within the explanation of the PVConv layer, there was no information about the network architecture or hyperparameters used in the MLP for the "Point-Based Feature Transformation" (Liu et al., 2019) branch.

2. Description of Implementation, Evaluation and Discussion

2.1. Implementation

2.1.1. Implementation Goal

The goal for this project was to reimplement the Point-Voxel CNN (Liu et al., 2019) that drastically improves upon the previous point and voxel-based state-of-the-art 3D deep learning models. Moreover, Point-Voxel CNNs (PVCNNs) have been extended in more recent models for applications such as autonomous driving, (Tang et al., 2020). The authors implement PVCNN in PyTorch, one of most popular machine learning frameworks. The other main ML framework is TensorFlow. While TensorFlow and Pytorch both allow users to easily create ML models, they both have strengths and weaknesses (Dubovikov, 2017). As a result, implementations of models with a large impact in the ML community are often implemented in both libraries. Therefore, the goal for this project was to reimplement PVCNN in TensorFlow to enable other TensorFlow developers to run, extend, or deploy PVCNN. In addition to the communitywide benefits, this reimplementation effort also served as a great way to learn how to implement, debug, and train a deep learning model end-to-end.

In the official PyTorch implementation of PVCNN[2] (Liu et al., 2019), the authors created multiple models to perform object part segmentation, indoor scene segmentation, and 3D object detection on the ShapeNet (Chang et al., 2015), S3DIS (Armeni et al., 2016), and KITTI (Geiger et al., 2013) data sets, respectively. Given the complexity of the implementation, limited time, and limited experience with TensorFlow as well as machine learning in general, the scope of the reimplementation effort was limited to implementing PVCNN to perform indoor scene segmentation on the S3DIS data set.

The sub-sections of Section 2.1 detail the major components of the TensorFlow implementation of PVCNN, which will be referred to as PVCNN-TF for the remainder of the paper.

2.1.2. Data Pre-Processing and Pipeline

Much of the pre-processing and data augmentation for the S3DIS (Armeni et al., 2016) data set was re-used from the original implementation. For example, the pre-processing module, `data/s3dis/prepare_data.py`, normalizes the x,y,z coordinate and color (RGB) values of each point in the point cloud. The input pipeline, on the other hand, was implemented from scratch using TensorFlow's `tf.data.Datasets`. The data pipeline performs three main tasks: loading and merging data from files, perform final data pre-processing, and configuring the `tf.data.Dataset`.

The pre-processing module writes the augmented input data and corresponding class labels for each indoor scene to two HD5F files. These two files represent two different subsets of the total point cloud the indoor scene. The first step of the input pipeline is to gather a subset of the HD5F file paths from the pre-processing output directory. This subset is defined in the experiment configuration based on which indoor scene areas should comprise the training vs testing/validation split. The file paths and data format information are used to create a `tf.data.Dataset` for each file using TensorFlow's I/O library. Finally, all of the individual datasets from each indoor scene are merged into one large dataset. Additionally, the total number of samples in the desired data split is determined while reading in each of the HD5F files to determine the number of batches in an epoch.

The majority of data pre-processing is handled in `data/s3dis/prepare_data.py`. However, there are a few final pre-processing steps that are included in the PVCNN-TF input data pipeline. The key step that needed to be replicated from the original data loader is the random sampling a configurable number of points from the original point cloud. Sampling with replacement was trivial to implement using samples from Uniform distribution to index the original point cloud. However, there was no built in operator in TensorFlow to sample without replacement. Research revealed that others have requested this feature in TensorFlow and have resorted to a work-around using a variant of the Gumbel-max trick (Willson, 2018). At a high-level this equates to getting the indices of the top K unnormalized log probabilities (logits) after adding i.i.d Gumbel(0,1) noise to them. To equally sample from all points in the point cloud, the logits vector is simply a zero vector of size equal to the number of points in the original point cloud. The PVCNN-TF pipeline performs two additional operations from the original implementation. First, another layer of filtering is performed to ensure that there are no samples with corresponding class labels outside of the appropriate range of 0 to 12 (13 classes). Lastly the labels are transformed to a one-hot representation so that these labels can be directly passed to the loss function in the training step (See Section2.1.4).

Finally, the `tf.data.Dataset` instance was configured to use a desired batch size, shuffle samples, and utilize prefetching. Prefetching is a optimization provided by `tf.data.Datasets` to read the data for the next step while executing the current training/evaluation step. Originally, the pipeline was also configured to cache the data set to save time by performing the pre-processing steps only for the first epoch. However, because the points are randomly sampled in the input pipeline pre-processing, the output points are different for each epoch. As a result, TensorFlow will cache every single batch. This eventually leads to running out of RAM, therefore caching is not feasible for this data set.

A visualization script was included with the implementation to verify that the input data pipeline produced valid results. Matplotlib is used to plot multiple 3D scatter plots on top of each other by masking the points in a scene point cloud for each of the desired class labels. See Figure 1 for an example output of the visualization script. Additionally, the input data pipeline was validated using an interactive tool, lidarview.com(XtSense-GmbH, 2013).

2.1.3. Experiment Configuration

The original PVCNN (Liu et al., 2019) paper implemented and trained multiple models for each of the 3D tasks they used to evaluate their architecture. For example, when evaluating PVCNN on S3DIS for indoor scene segmentation, the authors implemented three versions of the original PVCNN architecture with different number of feature channels, a reimplemented version of PointNet(Qi et al., 2017a), and

[2]https://github.com/mit-han-lab/pvcnn

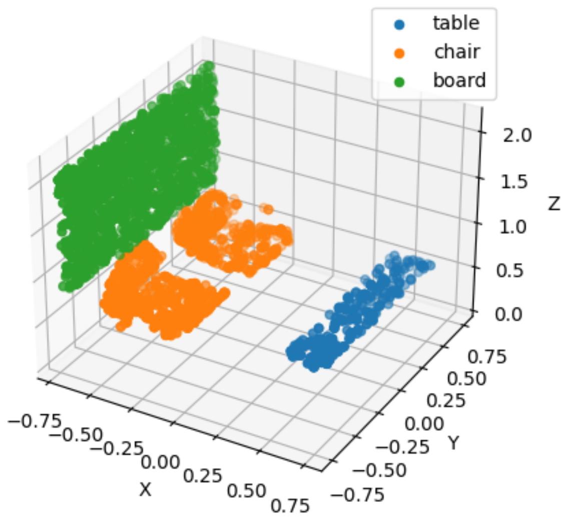

Figure 1. Data pipeline visualization script output. Points corresponding to walls, ceiling, and floor were removed.

an improved version of PVCNN based on PointNet++(Qi et al., 2017b).

The authors used a hierarchical set of configuration files to facilitate running different experiments for different model versions. These configuration files enable an organized definition of the experiment parameters and other objects used in training evaluation (e.g. the model class) while only requiring a single line of code in the final training script. This was done using a simple Python class `Config` that subclasses a python dictionary and adds a few utility functions such as initialization of the objects passed to a `Config` instance with their respective parameters defined in the config files.

PVCNN-TF makes use of the same `Config` class as the original implementation for simplicity.

2.1.4. Evaluation Metrics

PVCNN-TF shares the same evaluation metrics as the original implementation (Liu et al., 2019). Namely, categorical cross entropy is used for the loss function with a softmax output. This is a standard choice for supervised tasks such as point segmentation.

In addition to the loss function, the original paper used two accuracy metrics: mean categorical and mean intersection-over-union (IoU). Mean categorical accuracy measures how often the models prediction of the most likely class matched the correct class label for each point. For this metric, TensorFlow's `tf.keras.metrics.CategoricalAccuracy` metric was used. IoU measures the number of correctly predicted class labels relative to the total number of points with that class label across both the prediction and the ground truth. A custom `tf.keras.metrics.Metric` class was implemented in PVCNN-TF to calculate Mean IoU rather than use TensorFlow's built-in `tf.keras.metrics.MeanIoU` metric. This was done as there were assertions failing in the calculation of the confusion matrix related to class labels falling outside the range of 0 to "number of classes" despite filtering out any invalid labels in the data pipeline (see Section 2.1.2).

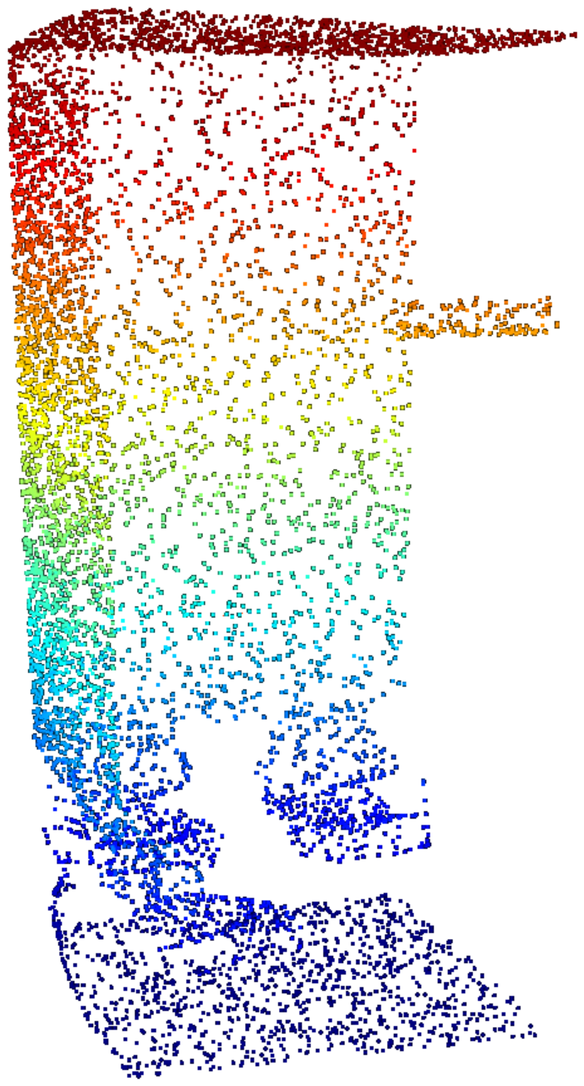

Figure 2. Data pipeline visualization using lidarview.com.

2.1.5. Custom Operations

Their are two custom operations that are a part of the Point-Voxel Convolution (Liu et al., 2019): average voxelization and trilinear devoxelization. As described in the paper, point features are transformed to a voxel representation to aggregate neighboring point information using 3D convolutions. Voxelization is performed by assigning each point to a single voxel in the grid. Finally, the features for each voxel are set to the average of the features of the points that fall in that voxel. Following the volumetric convolutions, devoxlization is performed to map voxel features back to each individual point. Trilinear devoxelization sets each point's features to a weighted interpolating of the features of eight surrounding voxels around that point. The forward pass and gradient calculations for both operations were implemented as custom CUDA kernels in PVCNN as these operations are expensive and hard to perform with standard tensor operations.

Given limited experience with TensorFlow, the following options were considered for reimplementing these operations for PVCNN-TF: 1. implement a custom TensorFlow op with a GPU/Cuda kernel, 2. translating the operations to existing TensorFlow operations (e.g. `reduce_sum`, `scatter_nd`, etc.), 3. translating the operations to Python code that will execute eagerly (i.e. `tf.py_function`), or 4. find and modify an existing voxelization / devoxelization scheme that is already implemented in TensorFlow. An initial attempt of option two for the average voxelization op proved to be very difficult so it was abandoned. Option three was not attempted as research and consultation with experts[3] suggested that this option would likely produce a large bottleneck for training. Option four was ruled out after failing to find adaptable voxelization / devoxelization schemes in TensorFlow. This left only left pursuing option one.

There were two main sets of challenges associates with creating a custom GPU operation in TensorFlow: configuring the development environment and testing / debugging the op implementations.

The first major challenge for implementing a custom GPU operation in TensorFlow was compiling CUDA code on Google Colab. Two Cuda header files needed for compiling .cu files were missing from Colab's vanilla environment and had to be copied from a TensorFlow custom ops docker container (tensorflow, 2018). The other major problem was a bug in TensorFlow 2.7 within a header file needed for defining the shapes for the custom ops, `tensor_types.h`. Downgrading to TensorFlow 2.6.2 resolved this issue. The last set of errors were related to having mismatched CuDNN versions in Colab.

[3]This option was discouraged by both Dr. Inouye as well as the student team lead of the Purdue TensorFlow Model Garden research team.

A few issues were noticed during development and testing of the voxelization and devoxelization operations. First, outputs of the operation would change from one run to the next with the same inputs. This appeared to be resolved by `memset`ing the inputs to 0 before calling the CUDA kernels. The original implementation (Liu et al., 2019) of average voxelization is prone to array index out of bounds error if the voxel resolution is small. PVCNN-TF clips the voxel index based on the resolution to prevent this issue. Finally, the gradients calculated for trilinear devoxelization backwards pass inconsistent values in the first batch compared to the other batches. This phenomena was only observed for small resolutions so it was deemed unlikely to cause an issue during model training. For more details, see the unit tests written for voxelization and devoxelization ops in PVCNN-TF.

2.1.6. Model Architecture

The goal when designing PVCNN-TF was to create a faithful reproduction of the original model(Liu et al., 2019) using a model architecture that is clear and easily extensible. The inputs to the model are a batch of point clouds with six or nine channels (i.e. a floating point tensor with shape [B x C x N] where B is the batch size, C is either six or nine, and N is the number of points in the point cloud block). The first six channels represent the red, green, and blue intensities as well as the x, y, and z coordinates of that point in the point cloud block. The additional three channels represent the x, y, and z coordinates normalized based on the largest x, y, and z coordinates of points in the scene. The PVCNN model for S3DIS contains three components based on the PointNet(Qi et al., 2017a) architecture: the point features branch, the cloud features branch, and a classification head. The point features branch is responsible for extracting features for each point in the point cloud. These features capture information about the local structure neighboring that point as well as individual point characteristics. This branch is composed of a configurable number of blocks. The cloud feature branch extracts information about the overall point cloud. This is done by effectively max pooling the final point features outputted by the point features branch over the point dimension for each channel. The point features extracted from each block of the point feature branch are concatenated with the cloud feature branch output. This combined tensor is fed into a classification head that downsamples the channels to the number of classes and then applies a softmax over that channel dimension. This output of the model in a [B x 'number of classes' x N] tensor that represents a class prediction for each point in the point cloud.

As mentioned in Section 2.1.3, the authors implemented multiple versions of PVCNN for each of the experiments they conducted in the original paper(Liu et al., 2019). To facilitate code reuse across the different PVCNN versions, the authors created a set of re-usable layers (in `modules`) and a common layer factory module (`models\utils.py`). Although the implementation goal was to implement only the S3DIS PVCNN model, PVCNN-TF took inspiration from and tried to improve upon this modular structure. Namely, PVCNN-TF includes a Python package named `layers` that contains all of the `tf.keras.layers.Layer` classes to represent the common layers and building blocks (sets of layers repeated in a particular order) used in the PVCNN-TF S3DIS model.

In the original implementation, the S3DIS PVCNN model directly called the utility factory module and performed multiple reshaping operations between the layers created by the factory functions. This was confusing when reading through the code for the first time as there was no clear distinction on the major sub-components of the model. In contrast, PVCNN-TF's PVCNN S3DIS model, simply instantiates three `tf.keras.layers.Layers` that represent the three major components of PVCNN. Each of these components handle the majority of the input/output re-shaping operations and creation of the common layers through the factory functions. This structure facilitates code re-use, readability, and unit testing of individual components.

The PVCNN-TF model also emphasizes documentation through commenting. One common challenging aspect when reading through the code for any neural network is keeping track of the shapes as the input passes through each of the model layers. PVCNN-TF adds many more comments to indicate the shapes of layer outputs at each stage in the model.

2.1.7. Training and Evaluation

The final aspect of the PVCNN-TF implementation is the training and evaluation module. The training and evaluation script took inspiration from the original implementation(Liu et al., 2019) for command line arguments and initializing the experiment configuration object. However, the remaining portion of the module was written from scratch.

The training and evaluation model uses custom training loops as opposed to the default training loops (i.e. `fit()` and `evaluate()`) to have low-level control during training. This proved useful when debugging issues with training.

One additional feature the PVCNN-TF added to the original implementation is model checkpoint saving. This enables saving the model, optimizer, and other training state variables at regular intervals in case something goes wrong during training. The training module includes two checkpoint managers that save the progress of the model multiple times an epoch as well as one that saves the model whenever the validation loss reaches a new minimum. This is especially useful for training on Google Colab as there are frequent disconnections due to usage limits.

Table 1. Results of indoor scene segmentation of S3DIS for the TensorFlow and original (Pytorch) implementations of PVCNN. Latency and GPU memory were averaged over 1000 and 10 model inferences, respectively.

Model	mAcc	mIoU	Latency	GPU Mem.
PVCNN	86.66	56.12	47.3 ms	1.3 GB
PVCNN-TF	∼4.0	∼35.0	45.2 ms	0.82 GB

2.2. Experiment Results and Discussion

Results for PVCNN-TF were collected based on initial training efforts. As shown in Table 1, PVCNN-TF's inference latency and GPU memory were 4.5% and 37% less than the original implementation (Liu et al., 2019), respectively. The slight improvement in latency appears reasonable and may be attributed to small performance differences between similar operations in TensorFlow and Pytorch. On the other hand, significant reduction in GPU memory usage was unexpected. GPU memory was measured using an experimental feature in TensorFlow, `tf.config.experimental.get_memory_info`. Thus, the accuracy of this measurement is not well understood at this time.

Both mean categorical accuracy and mean IoU accuracy of PVCNN-TF were much lower than the original model (Liu et al., 2019). This was a result of ongoing issues that prevented training the model for multiple epochs. These issues are discussed later in this section. Although it is difficult to compare the accuracy of the TensorFlow model to the original, the approximate 4x reduction in loss and 35% mean IoU accuracy acheived in the first 2500 iterations of the first epoch (see Figure3) suggests that PVCNN-TF was in fact learning up until crashing.

The issue that prevented training for multiple epochs was computation or numerical instabilities that led to undefined values (i.e NaNs) in the model predictions. Debugging was done by switching between execution to eager mode and graph mode for better error messages and inserting `tf.print` statements at different stages of the model and training. Initially, the first model prediction contained NaNs. Research suggested that modifying model parameters such as the batch size and learning rate could alleviate this problem. Lowering the learning rate to a very small value (e.g. 1e-6) enabled training to occur for multiple iterations. It was

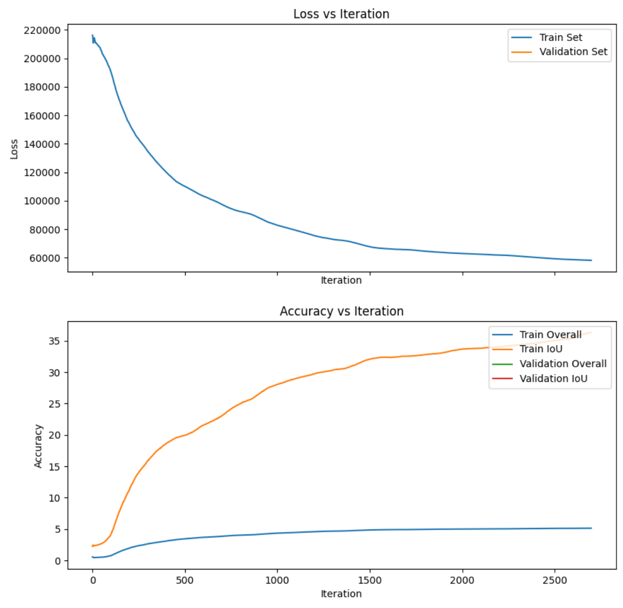

Figure 3. Loss and accuracy vs training iteration for the first epoch.

observed that the loss remained constant after less than 25% through the first epoch. Based on these observations, the learning rate should be very small initially (i.e. 1e-6) and gradually increase to a typical starting value (e.g. 1e-3) before decaying (See Figure 4). This approach has been used by other models such as Transformers (Vaswani et al., 2017). PVCNN-TF replaced the original cosine decay learning rate scheduler (LRS) with a "warm-up" LRS. Additionally, the voxelization layer in the original implementation had a potential for a divide by zero error in the event of the norm of the mean normalized coordinates being equal to zero. The authors included a variable, `epsilon` to add to the denominator in this layer but it was set to 0. This was corrected in PVCNN-TF by setting `epsilon` to a small positive value.

Despite these changes, there were still NaNs in the loss. Additional debug logging at this stage pointed to the gradient calculations as the source of the NaNs. At this point, various strategies were applied to resolve the NaNs in the gradient calculations. These included clipping the gradients using the global norm of the gradients (Pascanu et al., 2013), adding additional batch normalization layers, changing the batch size, and different initialization of the layer weights. None of these strategies resolved the issue.

One final strategy was to replace the NaNs in the gradients with small values (e.g. one) in the hopes that a small nudge to the parameter values would push the model off of the unstable point. While this appeared to work at first, the source of the NaNs changed to layer outputs in the forward pass. A similar approach to the NaN replacement in the gradients was used on the layer outputs. Instead of replacing NaNs with a small value, NaNs were replaced with the norm of the tensor. This norm was calculated after replacing NaNs in the original tensor with zeros. The goal of this was to ensure numerical stability. However, adding this transformation after every layer resulted in an out of GPU memory error.

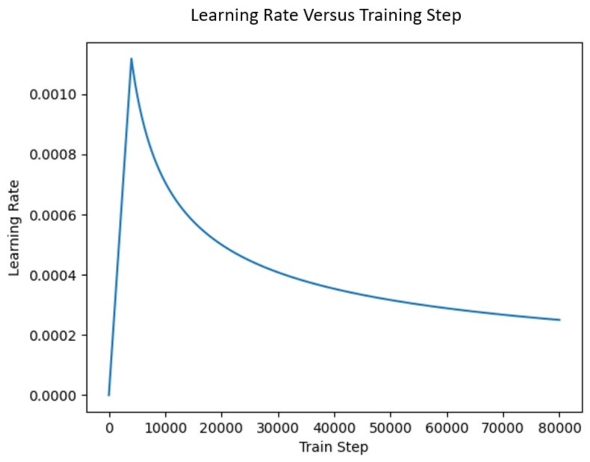

Figure 4. Learning rate scheduler based on the Transformers network (Vaswani et al., 2017).

Advice was sought out from machine learning experts[4] for these issues and debugging attempts. With time, a few suggestions could be explored. It is possible that the custom operation may have a small numerical instability in the forward or backwards calculations that may lead to NaNs. While there were no NaNs observed in these layers on the forward pass, additional investigation is needed to inspect the backward pass. One option may be to place `tf.print` statements in the gradient registration function for the custom ops. Debugging attempts focused on looking for the source of the first NaN. One possibility could be values that go to infinity (i.e. `Inf`) that may be used in an expression, thus generating NaNs on the output. Thus, it may be beneficial to add assertions to detect layer weights / outputs that exceed some upper threshold such that multiplying those values by a value greater than one could result in `Inf`(s).

Additional research online validated that issues with NaNs in TensorFlow are common and their root cause are often unclear especially when the models are complex and large in size. The lack of clarity in error messages is one known weakness of TensorFlow in comparison to PyTorch.

[4]Dr. Inouye and Dr. Aly El Gamal.

References

Armeni, I., Sener, O., Zamir, A. R., Jiang, H., Brilakis, I., Fischer, M., and Savarese, S. 3d semantic parsing of large-scale indoor spaces. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pp. 1534–1543, 2016. URL https://openaccess.thecvf.com/content_cvpr_2016/html/Armeni_3D_Semantic_Parsing_CVPR_2016_paper.html.

Chang, A. X., Funkhouser, T., Guibas, L., Hanrahan, P., Huang, Q., Li, Z., Savarese, S., Savva, M., Song, S., Su, H., et al. Shapenet: An information-rich 3d model repository. *arXiv preprint arXiv:1512.03012*, 2015. URL https://arxiv.org/abs/1512.03012.

Dubovikov, K. Pytorch vs tensorflow — spotting the difference, 2017. URL https://towardsdatascience.com/pytorch-vs-tensorflow-spotting-the-difference-25c75777377b. Accessed: 2021-10-12.

Geiger, A., Lenz, P., Stiller, C., and Urtasun, R. Vision meets robotics: The kitti dataset. *International Journal of Robotics Research (IJRR)*, 2013. URL https://journals.sagepub.com/doi/full/10.1177/0278364913491297/.

Hinton, G. E., Osindero, S., and Teh, Y.-W. A fast learning algorithm for deep belief nets. *Neural computation*, 18 (7):1527–1554, 2006. URL https://direct.mit.edu/neco/article/18/7/1527/7065/A-Fast-Learning-Algorithm-for-Deep-Belief-Nets.

Jaderberg, M., Simonyan, K., Zisserman, A., et al. Spatial transformer networks. *Advances in neural information processing systems*, 28:2017–2025, 2015. URL https://proceedings.neurips.cc/paper/2015/hash/33ceb07bf4eeb3da587e268d663aba1a-Abstract.html.

Liu, Z., Tang, H., Lin, Y., and Han, S. Point-voxel cnn for efficient 3d deep learning. In *Advances in Neural Information Processing Systems*, 2019. URL https://proceedings.neurips.cc/paper/2019/hash/5737034557ef5b8c02c0e46513b98f90-Abstract.html.

Pascanu, R., Mikolov, T., and Bengio, Y. On the difficulty of training recurrent neural networks. In *International conference on machine learning*, pp. 1310–1318. PMLR, 2013.

Qi, C. R., Su, H., Nießner, M., Dai, A., Yan, M., and Guibas, L. J. Volumetric and multi-view cnns for object classification on 3d data. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pp. 5648–5656, 2016. URL https://openaccess.thecvf.com/content_cvpr_2016/html/Qi_Volumetric_and_Multi-View_CVPR_2016_paper.html.

Qi, C. R., Su, H., Mo, K., and Guibas, L. J. Pointnet: Deep learning on point sets for 3d classification and segmentation. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pp. 652–660, 2017a. URL https://openaccess.thecvf.com/content_cvpr_2017/html/Qi_PointNet_Deep_Learning_CVPR_2017_paper.html.

Qi, C. R., Yi, L., Su, H., and Guibas, L. J. Pointnet++: Deep hierarchical feature learning on point sets in a metric space. *arXiv preprint arXiv:1706.02413*, 2017b. URL https://arxiv.org/abs/1706.02413.

Su, H., Maji, S., Kalogerakis, E., and Learned-Miller, E. Multi-view convolutional neural networks for 3d shape recognition. In *Proceedings of the IEEE international conference on computer vision*, pp. 945–953, 2015. URL https://www.cv-foundation.org/openaccess/content_iccv_2015/html/Su_Multi-View_Convolutional_Neural_ICCV_2015_paper.html.

Tang, H., Liu, Z., Zhao, S., Lin, Y., Lin, J., Wang, H., and Han, S. Searching efficient 3d architectures with sparse point-voxel convolution. In *European Conference on Computer Vision*, pp. 685–702. Springer, 2020. URL https://link.springer.com/chapter/10.1007/978-3-030-58604-1_41.

tensorflow. Guide for building custom op for tensorflow. https://github.com/tensorflow/custom-op, 2018.

Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., Kaiser, Ł., and Polosukhin, I. Attention is all you need. In *Advances in neural information processing systems*, pp. 5998–6008, 2017.

Willson, M. Sampling from a categorical distribution without replacement. https://github.com/tensorflow/tensorflow/issues/9260#issuecomment-408950922, 2018.

Wu, Z., Song, S., Khosla, A., Yu, F., Zhang, L., Tang, X., and Xiao, J. 3d shapenets: A deep representation for volumetric shapes. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pp. 1912–1920, 2015. URL https://www.cv-foundation.org/openaccess/content_cvpr_2015/html/Wu_3D_ShapeNets_A_2015_CVPR_paper.html.

XtSense-GmbH. Online lidar point cloud viewer. http://lidarview.com/, 2013.