Project 5: Classifying Point Clouds with PointNet

CS 6476

Spring 2025

Brief

- Due: Check Canvas for up to date information
- Project materials including report template available on the Canvas Assignments Tab
- Hand-in: through Gradescope
- **IMPORTANT**: When submitting your report, **you must assign a slide to every problem even in the case of problems you do not complete, including extra credit.** The only exception is for "Bells & Whistles (alternate extra credit)," where it is acceptable to not attach a slide only if you do not try any alternate extra credit
- Required files: `<your_gt_username>.zip`, `<your_gt_username>_proj5.pdf`

Overview

In this project, you will design and train a neural network to classify point clouds from lidar scans.

A GPU is not necessary for this project. The networks you will be building are simple enough to train quickly on CPU.

Setup

1. Follow Project 5 README in the ZIP.
2. Run the notebook using `jupyter notebook proj5.ipynb` inside the repository folder.
3. After implementing all functions, ensure that all sanity checks are passing by running `pytest tests` inside the repository folder.
4. If you'd like to run this project on Colab or PACE, please follow the same procedure as previous projects to upload the project and install the environment.
5. Generate the zip folder for the code portion of your submission once you've finished using `python zip_submission.py --gt-username <your_gt_username>`

Introduction

For most of this course, we've focused on understanding 2D *images*. Convolutional neural networks have proven adept at ingesting 2D grids of pixel intensities and outputting classification decisions, segmentations, depth estimates, or more complex outputs like detections and keypoints. But what if we want to use neural networks to make decisions about *3D data*? 3D data is growing more popular as RGBd and lidar sensors become cheaper and more accurate. For robotics applications, including self-driving vehicles, it would be

helpful to make decisions about this 3D data. However, there isn't a canonical representation for 3D data. Many representations have been tried – range view images, voxel grids, polygon meshes, point clouds, and renderings of one of these previous representations from fixed views. In this project you will implement a neural network to classify point clouds cropped from scenes encountered by a self-driving vehicle. The network will be a simplified version of the PointNet architecture which we discussed in lecture. We will compare our PointNet representation to a simple hand-defined voxel grid representation.

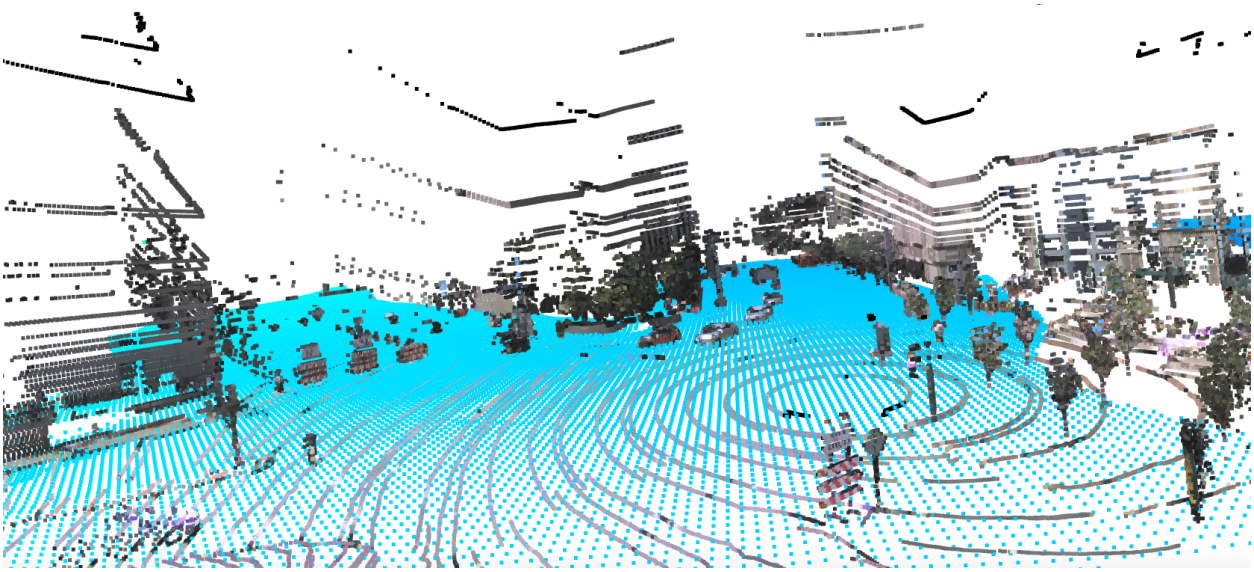


Figure 1: Lidar points for a driving scene. The lidar sensor itself is in the middle of the concentric rings, attached to a self-driving vehicle (not shown). The lidar sensor measures not just 3D location but also infrared intensity (brightness). However, this visualization instead shows color as projected from RGB cameras. The blue points come from an HD map of ground height, not used for this project. You can make out signs, other vehicles, trees, and buildings. In this project, we have cropped some of these objects out of the full scene point clouds and you will train a network to classify those 3D points.

Depth sensors

Lidar (Light Detection And Ranging) sends light pulses (often infrared) and measures the time it takes for the light pulses to return in order to estimate distance in a particular direction. Lidar is a "time of flight" sensor, in contrast to other distance measurement strategies such as stereo which instead rely on parallax between corresponding points between two views. Stereo has difficulty with textureless regions (e.g. smooth roads) where point correspondence is difficult to establish. But lidar has difficulty at long range (where light pulses might not be bright enough to return) and on dark objects (where light pulses will be absorbed).

Lidar sensors are *active* – they send out energy into the environment. Stereo has the advantage that it is *passive* – it doesn't require energy to send out light pulses because it relies on the ambient light in a scene. This also means that stereo systems don't need to worry about interference from other stereo systems, while lidar systems can interfere with each other.

Something of a hybrid of these approaches is structured light stereo in which one stereo camera is replaced by a projector. This gives the advantage that correspondence becomes easier because the pattern will be designed so that there are no 'textureless' regions and thus correspondence is easier. But such stereo systems inherit the disadvantages of active sensors – concerns about interference and difficulty with dark or distant objects.

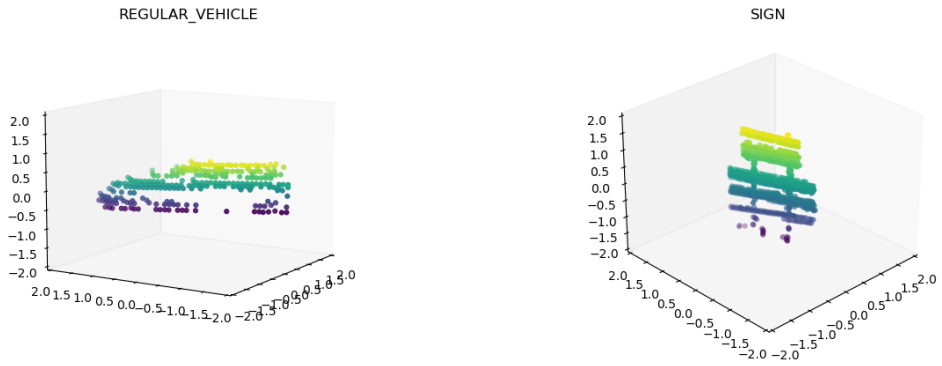
Recent lidar sensors have another disadvantage compared to stereo – resolution. This disadvantage may

be disappearing as more sensor research produces better lidar sensors, but at the moment lidar depth measurements often have a vertical resolution of 32, 64, or 128 scanlines. Lidar sensors are often physically rotating "pushbroom" sensors that measure one vertical column of a scene at time. It is common for lidar sensors to rotate at 10 or 20 Hz and capture hundreds of thousands of depth measurements during each rotation.

Regardless of the mechanism used to measure depth, many modern sensors can return a *depth map* – a "2.5D" image which encodes distance in various directions from the sensor. The term "2.5D" is used because these sensors typically recover only one depth measurement per direction, so the surface of an object oriented away from the sensor would not be measured. This is in contrast to human-authored 3D meshes where objects would typically form complete watertight surfaces. For this project (and many lidar-processing learning approaches) we have turned these depth measurements into 3D point clouds.

Dataset

The Argoverse 2 Sensor Dataset[4] is an open-source annotated collection of 3D lidar, stereo imagery, and ring camera data. The lidar-derived point clouds from Argoverse will be the basis of this project. While self-driving vehicles would typically want reason about the entire point cloud, we have simplified things by cropping objects out of the point clouds. We've skipped over objects with too few points (less than 20) and for objects with too many points (more than 200) we've subsampled the point clouds. Each cropped region spans a 4 m x 4 m x 4 m volume. That may leave a lot of empty space for small objects (e.g. pedestrians) while not capturing the full point cloud from other objects (e.g. buses), but hopefully our PointNet can learn to deal with that. The dataset you will use for this project contains 4000 total objects instances spanning 20 categories each with 200 point clouds. The dataset is divided into 3400 training instances (170 per class) and 600 testing instances (30 per class). We call these point cloud object instances the *Argoverse Object Dataset*.



(a) Regular Vehicle Point Cloud

(b) Sign Point Cloud

Figure 2: Example point clouds from the Argoverse Object Dataset. These point clouds have been cut out from larger LiDAR scans.

Your task will be to train a simplified PointNet architecture on the training dataset that achieves high accuracy in classifying the held-out test set instances.

Baseline Voxel Grid

Before you implement PointNet, let's examine a simple baseline. We will represent each object's point cloud with a voxel grid that counts how many points fall into each discrete subvolume of the overall 4 m x 4 m x

4 m region, and then we will train a simple one layer MLP to make a classification decision.

Let's use a voxel grid with $1m^3$ cells. This gives us a $4 \times 4 \times 4 = 64$ dimensional feature. For each cell, simply count the number of lidar returns falling into that subvolume. Normalize each resulting feature so that it sums to 1.

Next, train an one layer MLP (or linear classifier) to predict class from your voxel features. You should be able to achieve an accuracy of 30% after tuning learning rate and number of training epochs. This isn't great, but it is still significantly higher than chance performance of 5%.

Optionally, you can try raising the accuracy of this baseline representation by (i) changing the resolution of your voxel grid (ii) creating a multi-resolution voxel grid, e.g. 2 x 2 x 2 and 4 x 4 x 4 and 8 x 8 x 8 (iii) using 'soft' assignments of points to voxel cells, just like we had 'soft' assignment of gradients to SIFT histogram bins. E.g. a point might contribute, with interpolation, to its 8 nearest cells instead of the single cell it happens to fall in.

PointNet Implementation

PointNet[1] is a network architecture designed to work directly on point cloud data, rather than relying on converting to a different format such as voxels. Since there is no ordering between points, PointNet is designed to be *permutation invariant*, meaning that shuffling the order of points in the point cloud input does not change the output of the network. This is achieved by independently processing each point, and then aggregating their features through the use of a max function to form a *global feature*.

In this part, you will implement a simplified version of PointNet that takes in a cropped point cloud and returns class scores. We will not require the implementation of the input transform and feature transform. The simplified network structure is shown below. This network is relatively light-weight since it is just a sequence of a few MLPs (just PyTorch Linear layers). Because of the small number of parameters, this network can be trained relatively quickly using just a CPU.

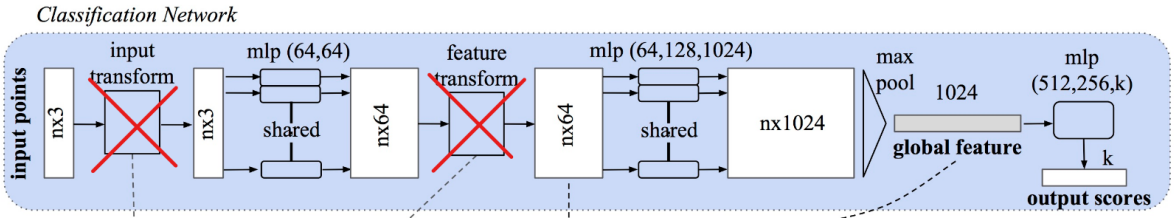


Figure 3: Each point is processed independently through a sequence of MLPs (each point goes through the same set of MLPs, but points have no influence on each other), and these outputs are combined via a max to form a global feature. This is then fed into an MLP classifier to produce output scores for each class. Be sure to use ReLUs between MLPs.

We were able to get around 65% accuracy on the test dataset of the Argoverse Object Dataset after training for 10 epochs which took around 3-4 minutes on CPU (around 20 seconds per epoch). This simplified PointNet may not be as accurate as the full PointNet described in the paper. For example, it achieves 73% accuracy on the ModelNet 40 dataset, which is the same as the 'baseline' reported in the PointNet paper but less than the 86% of the full PointNet architecture. Here is the link to the original PointNet paper for further reading. You will need to get at least 60% accuracy on the test set to receive full points.

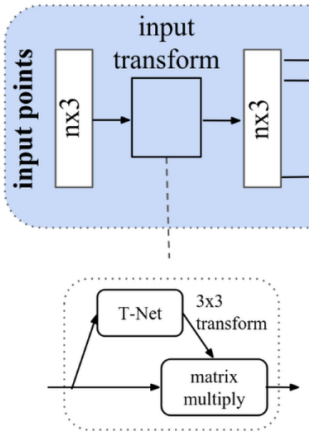


Figure 4: T-Net is applied to transform the inputs to the PointNet.

T-Net

*This section is **required** for CS 6476 students and **optional** for CS 4476.*

Because the PointNet encoder is simply an MLP with a maxpooling for global feature aggregation, it does not contain invariances to transformations to the input. The original PointNet paper implements a T-Net to try and account for this. A T-Net is itself a PointNet that predicts a 9-long vector per input point cloud instead of a set of class scores. Each 9-long vector is then reshaped to a 3 by 3 matrix, and is summed with an identity matrix. The following equation describes the action of the T-Net:

$$\mathbf{T}(\mathbf{x}) = \text{reshape}(\mathbf{P}(\mathbf{x}), (B, 3, 3)) + I_3 \tag{1}$$

where $\mathbf{T}(\mathbf{x})$ represents the output of the T-Net, $\mathbf{P}(\mathbf{x})$ represents the output of a PointNet with a 9-dimensional output, and B is batch size. You will implement the T-Net under the TNet class in `part5_tnet.py`. It is easiest to re-use most of your code from your original PointNet implementation as the architecture is almost identical, differing only in the dimension of the final linear layer.

Once you have implemented the T-Net, you will need to implement the PointNetTNet class in `part5_tnet.py`. This will consist of both a T-Net and the original PointNet you implemented in `part3_tnet.py`. PointNetTNet works by first transforming the input point cloud by multiplying the point cloud by the matrices calculated by the T-Net, like so:

$$\mathbf{x}_t = \mathbf{x}\mathbf{T}(\mathbf{x})$$

where $\mathbf{x}_t$ are the transformed points, thus a tensor of size $(B, N, 3)$ while $\mathbf{T}(\mathbf{x})$ is the output of the T-Net, hence a tensor of size $(B, 3, 3)$. You will implement this in `apply_tnet`.

Afterwards, the outputs of the PointNetTNet are generated by:

$$\text{classes}, \text{encodings} = \mathbf{P}(\mathbf{x_t}) \tag{2}$$

where $\mathbf{P}$ is the original PointNet implementation from Part 3. To receive full credit, you must achieve at least 65% accuracy on the test set.

Additional Bells and Whistles

It is likely that we can push PointNet well beyond 65% accuracy on the Argoverse Objects Dataset. Here are some ideas that may result in improved accuracy:

- Optimize the simplified PointNet architecture – e.g. change the depth, the dimensionality of the point representations, the type of pooling, the learning rate schedule, the initialization procedure, etc.

- Implement the 'transform' blocks of the full PointNet architecture.
- Incorporate lidar point *intensity* into your model (you may need to ask the TAs for a variant of the dataset to try this).
- The object point clouds are normalized such that the coordinates of points are zero-centered, but that means information is lost about the object's position and orientation relative to the lidar sensor. It may be that adding that information back in (e.g. knowing that an object was 40 meters away and to the front left of the lidar sensor) will improve accuracy. Again, the current dataset doesn't have this metadata so we may need to tweak the dataset for such experiments.
- Implement PointNet++, a hierarchical version of PointNet (arXiv link).
- Try implementing positional encodings, e.g. those described in Trading Positional Complexity vs. Deepness in Coordinate Networks.

Beyond these suggestions to increase accuracy, we're also interested in seeing deeper analysis of the dataset, better visualizations of the data or the decisions made by your PointNet. Implementation of bells and whistles can increase your grade by up to 5 points.

Writeup

For this project (and all other projects), you must do a project report using the template slides provided to you. Do ***not*** change the order of the slides or remove any slides, as this will affect the grading process on Gradescope and you will be deducted points. **You must assign a slide to every problem even in the case of problems you do not complete, including extra credit.** In the report you will describe your algorithm and any decisions you made to write your algorithm a particular way. Then you will show and discuss the results of your algorithm. The template slides provide guidance for what you should include in your report. A good writeup doesn't just show results, it tries to draw some conclusions from the experiments. You must convert the slide deck into a PDF for your submission, and then assign each PDF page to the relevant question number on Gradescope.

If you choose to do anything extra, add slides *after the slides given in the template deck* to describe your implementation, results, and analysis. You will not receive full credit for your extra credit implementations if they are not described adequately in your writeup.

Rubric

- +70 points - Code (Detailed point breakdown can be found on the Gradescope autograder)
- +30 points - Report

Submission format

This is very important as you will lose 5 points for every time you do not follow the instructions. You will submit two items to Gradescope:

1. `<your_gt_username>.zip` containing:
   (a) `src/vision/` - directory containing all your code for this assignment
   (b) `setup.cfg`: setup file for environment, no need to change this file
   (c) `additional_data/` - (optional) if you use any data other than what we provided you, please include them here

(d) `README.txt`: (optional) if you implement any new functions other than the ones we define in the skeleton code (e.g., any extra credit implementations), please describe what you did and how we can run the code. We will not award any extra credit if we can't run your code and verify the results.

2. `<your_gt_usernamme>_proj5.pdf` - your report

Do ***not*** install any additional packages inside the Conda environment. The TAs will use the same environment as defined in the config files we provide you, so anything that's not in there by default will probably cause your code to break during grading. Do ***not*** use absolute paths in your code or your code will break. Use relative paths like the starter code already does. Failure to follow any of these instructions will lead to point deductions. Create the zip file using `python zip_submission.py --gt_username <your_gt_username>` (it will zip up the appropriate directories/files for you!) and hand it in with your report PDF through Gradescope (please remember to mark which parts of your report correspond to each part of the rubric).

Credits

Assignment developed by James Hays, Ben Wilson, Sooraj Karthik, Otis Smith, Jim James, Sili Zeng, Xueqing Li, Haris Hussain, and Deepanshi.

References

[1] R. Qi Charles, Hao Su, Mo Kaichun, and Leonidas J. Guibas. "PointNet: Deep Learning on Point Sets for 3D Classification and Segmentation". In: *2017 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. 2017, pp. 77–85. DOI: 10.1109/CVPR.2017.16.

[2] Ben Mildenhall et al. "NeRF: Representing Scenes as Neural Radiance Fields for View Synthesis". In: *ECCV*. 2020.

[3] Matthew Tancik et al. "Fourier Features Let Networks Learn High Frequency Functions in Low Dimensional Domains". In: *NeurIPS* (2020).

[4] Benjamin Wilson et al. "Argoverse 2: Next Generation Datasets for Self-Driving Perception and Forecasting". In: *Proceedings of the Neural Information Processing Systems Track on Datasets and Benchmarks (NeurIPS Datasets and Benchmarks 2021)*. 2021.