Module 1: 3D Object Detection

1 Introduction

In this assignment, you'll learn about 3D perception by implementing a LiDAR-based 3D object detector, similar to those used in real self-driving vehicles! The assignment has two parts. In Part A you will learn about the fundamentals, and in Part B you will have the ability to explore. Starter code is provided in the `csc490-projects` repository. We'll use Python and PyTorch in this assignment. Instructions for how to install the required libraries and prepare the dataset are found in the repository. **Note**: You should not introduce additional dependencies for this project.

Grading: This assignment will be graded out of 100. There are 120 points available; extra points will carry over to future assignments.

Late Policy: If you submit the assignment late, you will incur a 20 points penalty on this assignment. For each additional late day thereafter, you will incur an additional 10 points penalty.

Collaboration: For this assignment, you may work in groups of 1 to 2 people. In either case, the grading rubric will be the same. You may not reference or copy code from other sources. Please do your own work! We will be checking that indeed you coded the module.

Submission: When you finish the module and prior to the deadline, please submit the following files:

- `pa_writeup.pdf` – A write-up of your answers to questions in Part A.
- `pa_code.zip` – A copy of `csc490-projects`, with your changes from Part A.
- `pb_report.pdf` – A write-up of your project from Part B. See Part B for details.
- `pb_code.zip` – A copy of your code from Part B. See Part B for details.

Presentation: Additionally, prepare a presentation of your project focusing on what you did for Part B. The actual presentation should be no longer than 5 minutes, with an additional 5 minutes after for Q&A. See Part B for details.

2 Part A: Learning the Fundamentals [60 points]

In this part you will implement a LiDAR-based object detector. Starter code is provided in the `csc490-projects` repository. There, you'll find a mostly complete codebase, with the caveat that some functions have been replaced with stubs which we've marked `# TODO: Replace this stub code.` In the upcoming questions, we'll guide you through the process of implementing these functions. By the end of this section, you should have a fully functional object detector!

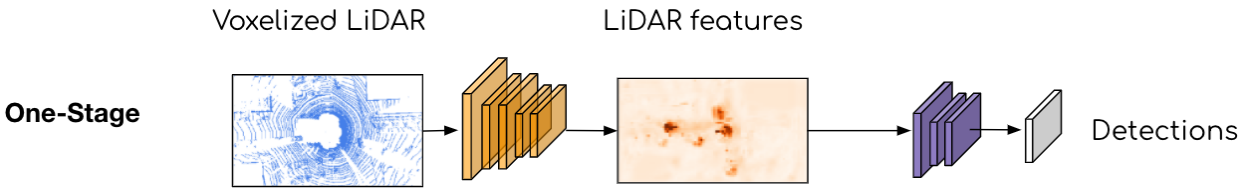

Figure 1: Overview of our one-stage object detector. The input LiDAR is first *voxelized* into a BEV voxel representation. Next, a CNN (typically termed the *backbone*) extracts features from the voxelized LiDAR. Finally, another CNN (typically termed the *detection head*) predicts BEV bounding box detections.

In this module, we will implement a one-stage object detector (see Fig. 1). The high-level idea is to use a neural network to predict a heatmap of where objects are located; each peak of the heatmap indicates an object's centroid. Then, for each object, we additionally regress other attributes such as bounding box sizes and heading angles. The implementation of this detector can be decomposed into five parts:

1. **LiDAR voxelization:** Given an input LiDAR point cloud, compute a bird's eye view ("BEV") voxel representation. This yields a 3-dimensional tensor $\mathcal{O} \in \{0,1\}^{D \times H \times W}$ where each $\mathcal{O}_{i,j,k}$ identifies whether the voxel (i,j,k) is occupied. D, H, and W denote the size of the BEV voxel representation along the z-, y-, and x-axes respectively.
2. **Feature extraction:** Given the BEV voxel representation $\mathcal{O}$, compute a feature grid $\mathcal{F} \in \mathbb{R}^{C \times H \times W}$ using a convolutional neural network ("CNN"). Next, given features $\mathcal{F}$, predict dense detection outputs $\mathcal{X} \in \mathbb{R}^{7 \times H \times W}$ using a second (smaller) CNN. Here, the seven channels encode the detection heatmap, coordinate offsets, box sizes, and heading angles.
3. **Model training:** To train the model, first compute a training target tensor $\mathcal{Y} \in \mathbb{R}^{7 \times H \times W}$ from ground truth detection labels. Next, use stochastic gradient descent ("SGD") to iteratively minimize the square loss[1] $\ell(\mathcal{X}, \mathcal{Y}) = \|\mathcal{X} - \mathcal{Y}\|_2^2$. Stop training just before the model overfits (this can be a bit of an art).
4. **Model inference:** To produce a set of BEV bounding box detections, first run a single forward pass of the model to obtain dense detection outputs $\mathcal{X}$ from $\mathcal{O}$. Then, run the inference algorithm to decode a set of BEV bounding boxes detections (see Question 2 Part 4 for details).
5. **Evaluation.** Finally, compute the precision/recall curve ("PR Curve") and average precision ("AP") to evaluate the performance of the detector.

We'll use a small subset of the Pandaset dataset [4] for experimentation, which we have split into a training set of 27 sequences and a validation set of 12 sequences. Each sequence consists of 80 LiDAR point clouds, each captured in intervals of 100ms. You will also find corresponding bounding box labels for each LiDAR point cloud. See the repository for instructions on how to download and prepare the dataset.

In the upcoming questions, you'll learn to implement aspects of parts 1, 3, 4, and 5. Let's get started!

2.1 LiDAR Voxelization [10 points]

In this question, you will implement the `Voxelizer` class for LiDAR voxelization. Starter code is provided in `detection/modules/voxelizer.py`.

[1] In practice, we use a weighted variant of the mean square error loss for better performance (see Eq. 9).

Part 1: Implement the `forward` method of the `Voxelizer` class.

This method converts a 3D LiDAR point cloud $\mathcal{P} = \{p_i \in \mathbb{R}^3\}_{i=1}^{N}$ into its occupancy voxel grid $\mathcal{O} \in \{0,1\}^{D \times H \times W}$. D is the voxel grid's size along the z-axis, H is its size along the y-axis, and W is its size along the x-axis. These sizes can be computed as follows:

$$D = \frac{(z_{\max} - z_{\min})}{\text{step}} \tag{1}$$

$$H = \frac{(y_{\max} - y_{\min})}{\text{step}} \tag{2}$$

$$W = \frac{(x_{\max} - x_{\min})}{\text{step}} \tag{3}$$

where $(x_{\min}, x_{\max})$, $(y_{\min}, y_{\max})$, and $(z_{\min}, z_{\max})$ configure the range of coordinates represented by the voxel grid along the x-, y-, and z-axes respectively, and step is the resolution of each voxel.

Each element $\mathcal{O}_{i,j,k}$ of the voxel grid denotes whether any point $p \in \mathcal{P}$ occupies the voxel (i,j,k), 1 if it is occupied and 0 otherwise. A point $p = (x,y,z) \in \mathcal{P}$ occupies the voxel at (i,j,k) if

$$\left\lfloor \frac{z - z_{\min}}{\text{step}} \right\rfloor = i \tag{4}$$

$$\left\lfloor \frac{y_{\max} - y}{\text{step}} \right\rfloor = j \tag{5}$$

$$\left\lfloor \frac{x - x_{\min}}{\text{step}} \right\rfloor = k \tag{6}$$

where $\lfloor x \rfloor$ is the floor function that returns the greatest integer less than or equal to x. Note that every point $(x,y,z) \in \mathcal{P}$ is ignored if $x \notin [x_{\min}, x_{\max}]$ or $y \notin [y_{\min}, y_{\max}]$. In contrast, we clip all points such that $z \in [z_{\min}, z_{\max}]$; this enables us to represent the full z-extent of the LiDAR point cloud.

Part 2: Visualize the result of `Voxelizer.forward` on one LiDAR point cloud for different resolutions step $\in \{0.25, 0.50, 1.0, 2.0\}$. How do the visualizations vary as you sweep different resolutions? What is the trade-off between memory consumption and the fidelity of the voxel grid?

2.2 Model Training & Inference [30 points]

In this question, you will implement aspects of the codebase for model training and inference. The relevant starter code is provided in `detection/modules/loss_target.py`, `detection/modules/loss_function.py`, and `detection/model.py`.

Part 1: Implement the `create_heatmap` function.

The purpose of `create_heatmap` is to compute the training target tensors $\mathcal{Y} \in \mathbb{R}^{7 \times H \times W}$. Specifically, this function computes a target detection heatmap $\mathcal{Y}_{\text{heat}}^{(n)} \in \mathbb{R}^{H \times W}$, which is used to train the detector to locate the centroid of one object $(x_n, y_n) \in \mathbb{R}^2$ in the scene. The idea is to construct a BEV heatmap whose peak indicates the centroid (x_n, y_n). Given indices $(i,j) \in \{1,\ldots,H\} \times \{1,\ldots,W\}$, the value of $\mathcal{Y}_{\text{heat},i,j}^{(n)}$ is given by a Gaussian kernel centered on (x_n, y_n) and with scale $\sigma > 0$.

$$\mathcal{Y}_{\text{heat},i,j}^{(n)} = \exp\left(-\frac{(x_n - i)^2 + (y_n - j)^2}{\sigma}\right) \tag{7}$$

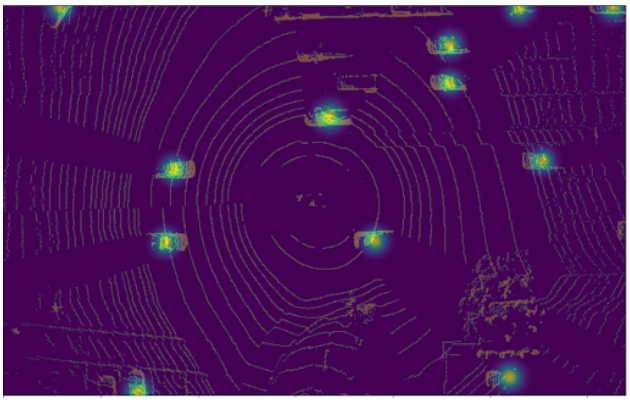

Figure 2: A target detection heatmap to train the detector. Brighter colours indicate larger values.

You may need to tune the σ a bit to make sure the heatmap looks resonable.

We also normalize $\mathcal{Y}^{(n)}_{\text{heat},i,j}$ such that its maximum value equals 1. To combine $\{\mathcal{Y}^{(n)}_{\text{heat}}\}_n$ into a single tensor $\mathcal{Y}_{\text{heat}} \in \mathbb{R}^{H\times W}$, for all indices $(i,j) \in \{1,\dots,H\} \times \{1,\dots,W\}$, we define

$$\mathcal{Y}_{\text{heat},i,j} = \max_n \mathcal{Y}^{(n)}_{\text{heat},i,j} \tag{8}$$

See Fig. 2 for an example heatmap $\mathcal{Y}_{\text{heat}}$. See `detection/modules/loss_target.py` for starter code.

Part 2: Implement the `build_target_tensor_for_label` method of `DetectionLossTargetBuilder`.

This method uses the `create_heatmap` function defined earlier to compute the full training target tensor $\mathcal{Y}^{(n)} \in \mathbb{R}^{7\times H\times W}$ for one object in the scene. In the starter code, this method is already partially filled in. Your task is to replace the stub code meant for computing the following:

1. $\mathcal{Y}^{(n)}_{\text{offset}} \in \mathbb{R}^{2\times H\times W}$, the target training tensor for offsets from the object centroid (x_n, y_n) to the indices $(i,j) \in \{1,\dots,H\} \times \{1,\dots,W\}$; i.e. $\mathcal{Y}^{(n)}_{\text{offset},i,j} = (x_n - i, y_n - j)$.

2. $\mathcal{Y}^{(n)}_{\text{size}} \in \mathbb{R}^{2\times H\times W}$, the target training tensor for the object's BEV bounding box size; i.e. $\mathcal{Y}^{(n)}_{\text{size},i,j} = (\ell_n, w_n)$, where ℓ_n (resp. w_n) is the size of the bounding box along the x-axis (resp. y-axis).

3. $\mathcal{Y}^{(n)}_{\text{head}} \in \mathbb{R}^{2\times H\times W}$, the target training tensor for the object's heading angle encoded with trigonometric functions; i.e., $\mathcal{Y}^{(n)}_{\text{head},i,j} = (\sin(\theta_n), \cos(\theta_n))$, where θ_n is the object's heading angle.

We also set the value of these tensors to 0 for all indices (i,j) where $\mathcal{Y}^{(n)}_{\text{heat},i,j} \leq 0.01$. We suggest you visualize each channel of the target tensor for debugging purposes.

See `detection/modules/loss_target.py` for starter code.

Part 3: Implement the function `heatmap_weighted_mse_loss`.

Given the detector's dense detection outputs $\mathcal{X} \in \mathbb{R}^{7\times H\times W}$ and the target training tensor $\mathcal{Y} \in \mathbb{R}^{7\times H\times W}$, this function computes a weighted mean squared error ("MSE") loss $\ell(\mathcal{X}, \mathcal{Y})$. Specifically, we will weight the loss

at each index (i, j) using its heatmap value $\mathcal{Y}_{\text{heat},i,j}$ and average over all indices such that $\mathcal{Y}_{\text{heat},i,j} > 0.01$,

$$\ell(\mathcal{X}, \mathcal{Y}) = \frac{1}{M} \sum_{i,j:\ \mathcal{Y}_{\text{heat},i,j} > 0.01} \mathcal{Y}_{\text{heat},i,j} \times \|\mathcal{X}_{\cdot,i,j} - \mathcal{Y}_{\cdot,i,j}\|_2^2 \tag{9}$$

where M is the number of indices (i, j) such that $\mathcal{Y}_{\text{heat},i,j} > 0.01$. Intuitively, this weighting scheme focuses our loss function on areas nearby an object.

See `detection/modules/loss_function.py` for starter code.

Part 4: Implement the `inference` method of `DetectionModel`.

So far, we've described how the detection model computes a dense detection output $\mathcal{X} \in \mathbb{R}^{7 \times H \times W}$. In this question, we will implement the function to decode a set of detections $\hat{\mathcal{D}} = \{(\hat{x}_\alpha, \hat{y}_\alpha, \hat{\theta}_\alpha, \hat{\ell}_\alpha, \hat{w}_\alpha, \hat{s}_\alpha)\}_\alpha$ from $\mathcal{X}$, where $(\hat{x}_\alpha, \hat{y}_\alpha)$ is the detection's BEV centroid, $\hat{\theta}_\alpha$ is its heading, $(\hat{\ell}_\alpha, \hat{w}_\alpha)$ is its bounding box size, and $\hat{s}_\alpha$ is its detection score.

There are several steps to this process.

1. **Forward pass.** Given the voxelized LiDAR $\mathcal{O}$, run a single forward pass of the detection model to obtain dense detection outputs $\mathcal{X}$.
2. **Localize detections.** Identify the local maxima (i.e., peaks) in the predicted heatmap $\mathcal{X}_{\text{heat}} \in \mathbb{R}^{H \times W}$. An index $(i, j) \in \{1, \ldots, H\} \times \{1, \ldots, W\}$ is a local maximum if $\mathcal{X}_{\text{heat},i,j}$ is the maximum value within a 5×5 window centered on (i, j). This yields a set of initial detections $\hat{\mathcal{D}}_1 = \{(i_\alpha, j_\alpha, \hat{s}_\alpha)\}_\alpha$, where (i_α, j_α) are indices in the $H \times W$ grid and $\hat{s}_\alpha = \mathcal{X}_{\text{heat},i_\alpha,j_\alpha}$ is the detection score.
3. **Refine locations.** For each detection $(i_\alpha, j_\alpha, \hat{s}_\alpha) \in \hat{\mathcal{D}}_1$, compute its centroid using the predicted offsets $\mathcal{X}_{\text{offset}} \in \mathbb{R}^{2 \times H \times W}$ as $(\hat{x}_\alpha, \hat{y}_\alpha) = (i_\alpha, j_\alpha) + \mathcal{X}_{\text{offset},i_\alpha,j_\alpha}$. This yields a set of refined detections $\hat{\mathcal{D}}_2 = \{(\hat{x}_\alpha, \hat{y}_\alpha, \hat{s}_\alpha)\}_\alpha$.
4. **Predict bounding box.** For each detection $(\hat{x}_\alpha, \hat{y}_\alpha, \hat{s}_\alpha) \in \hat{\mathcal{D}}_2$, compute its bounding box using the predicted box sizes $\mathcal{X}_{\text{size}} \in \mathbb{R}^{2 \times H \times W}$ as $(\hat{\ell}_\alpha, \hat{w}_\alpha) = \mathcal{X}_{\text{size},i_\alpha,j_\alpha}$. This yields a set of refined detections $\hat{\mathcal{D}}_3 = \{(\hat{x}_\alpha, \hat{y}_\alpha, \hat{\ell}_\alpha, \hat{w}_\alpha \hat{s}_\alpha)\}_\alpha$.
5. **Predict heading.** For each detection $(\hat{x}_\alpha, \hat{y}_\alpha, \hat{\ell}_\alpha, \hat{w}_\alpha, \hat{s}_\alpha) \in \hat{\mathcal{D}}_3$, decode its heading angle from the predicted heading angles $\mathcal{X}_{\text{head}} \in \mathbb{R}^{2 \times H \times W}$ as follows. Let $(\sin(\hat{\theta}_\alpha), \cos(\hat{\theta}_\alpha)) = \mathcal{X}_{\text{head},i_\alpha,j_\alpha}$. Then $\hat{\theta}_\alpha = \text{atan2}(\sin(\hat{\theta}_\alpha), \cos(\hat{\theta}_\alpha))$ by trigonometric identities. This yields the final set of detections $\hat{\mathcal{D}} = \{(\hat{x}_\alpha, \hat{y}_\alpha, \hat{\theta}_\alpha \hat{\ell}_\alpha, \hat{w}_\alpha \hat{s}_\alpha)\}_\alpha$.

See `detection/model.py` for starter code.

Part 5: Try overfitting to a single training sample using the default hyperparameters. Your detector should be able to do this and (almost) perfectly detect every vehicle in that one scene. Include a visualization of the resulting detections after overfitting in your write-up.

Part 6: After you've successfully overfitted your detector to a single training sample, it's time to train it on the full dataset. A word of warning — this could take up to an hour using the default hyperparameters. When training is complete, test your model on the held-out validation dataset. Include several visualizations of your model's detections on the validation dataset in your write-up.

2.3 Evaluation [20 points]

In this question, you will implement a variant of the precision/recall curve ("PR curve") and average precision ("AP") [1]. Starter code is provided in `detection/metrics/average_precision.py`.

Part 1: Implement the function `compute_precision_recall_curve`.

The PR curve plots the trade-off between precision and recall when sweeping across different score thresholds for your detections. To compute precision and recall for a score threshold s_i, consider the set of detections with scores greater than or equal to s_i. A detection is a true positive if it matches a ground truth label; it is a false positive if it does not. With this, we define

$$\text{precision} = \frac{\text{TP}}{\text{TP} + \text{FP}} \tag{10}$$

$$\text{recall} = \frac{\text{TP}}{\text{TP} + \text{FN}} \tag{11}$$

$$\tag{12}$$

where TP is the number of true positive detections, FP is the number of false positive detections, and FN is the number of false negative labels (i.e., the number of ground truth labels that did not match any detections). By varying the score threshold s_i over all detection scores, we can plot the PR curve.

What does it mean for a detection to match a ground truth label? In this assignment, we use the following definition: A detection matches a ground truth label if: (1) the Euclidean distance between their centers is at most τ; and (2) no higher scoring detection satisfies condition (1) with respect to the same label. Here, τ is a hyperparameter controlling the strictness of your metric.

Part 2: Implement the function `compute_area_under_curve`.

AP is defined as the area under the PR curve. To compute its value, we will use the following algorithm: Let (r_i, p_i) be the recall and precision values for the top i detections (ordered according to their detection scores). The AP is then computed as

$$\text{AP} = \sum_{i=1}^{n} p_i \times (r_i - r_{i-1}) \tag{13}$$

where n is the total number of detections and $r_0 = 0$. Intuitively, this is computing the integral of the step function defined by the PR curve.

Part 3: Implement the function `compute_average_precision` using Parts 1 and 2. Using the provided starter code, evaluate your trained detection model. Plot the PR curve for thresholds $\tau \in \{2.0, 4.0, 8.0, 16.0\}$ and report their corresponding AP. How does the PR curve and AP change as τ varies?

Part 4: Explain how the ground truth for the metrics is computed. Describe in detail the transformations applied to go from the raw data (the Pandaset files) to the model targets and the inputs to the `compute_average_precision` function. Discuss the implications of the transforms and how they influence what the model learns. Are there learning targets that we use during training that we could not realistically expect the model to predict at inference time?

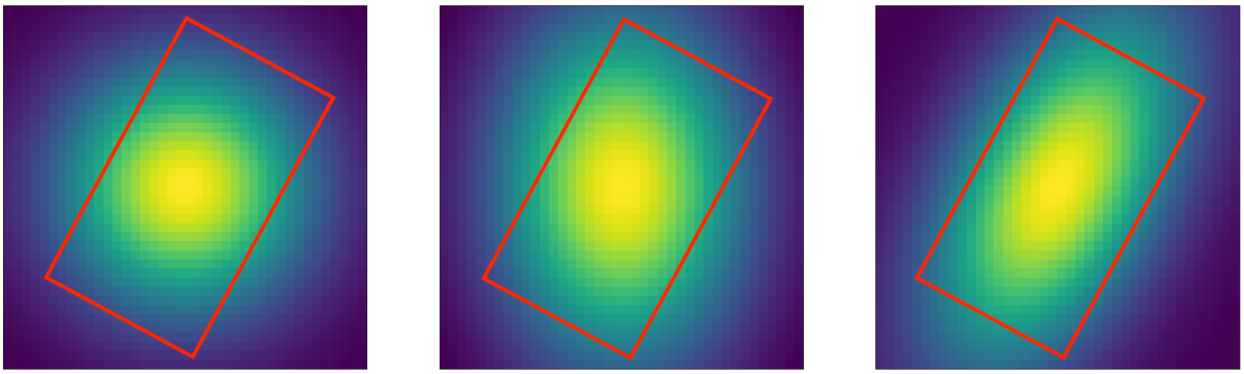

Figure 3: Left: Isotropic Gaussian with covariance matrix $\Sigma = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$. Middle: Anisotropic Gaussian with covariance matrix $\Sigma = \begin{bmatrix} 1 & 0 \\ 0 & 2 \end{bmatrix}$. Right: Rotated Gaussian with covariance matrix $\Sigma = \begin{bmatrix} 1 & 0.7 \\ -0.7 & 2 \end{bmatrix}$. Σ is determined by the box scale and heading (see the red bounding box).

3 Part B: Exploration [40 + 20 points]

To make it easy for you to get started with 3D detector, the minimal implementation in Part A is not well-tuned. There is a lot of room for improvement. You will now explore ways to improve the object detector that you implemented in Part A. This part of the assignment is open-ended; your goal is to identify an area of improvement that interests you, experiment with techniques to achieve the improvement, and prepare a report and presentation summarizing your findings.

There is an endless number of directions you can take for this. Below, we list three directions that are indicative of the scope we expect for this project. We encourage you to explore other ideas as well; if you choose to do so, please consult the TAs about your proposal.

- Experiment with a different input representation. The detector from Part A uses a simple BEV voxel representation, but other input representations (e.g., the hybrid representation of PointPillars [2]) may improve its performance. Implement at least one more representation and analyze how it affects the performance of your detector.
- Implement a multi-class detector. The detector from Part A only detects vehicles, which is obviously insufficient for driving in the real world. Can you extend it to detect other classes as well; e.g., pedestrians, bicyclists, etc.?
- Explore more sophisticated loss functions. We used a simple weighted mean-squared error ("MSE") loss to train the detector from Part A. In contrast, state-of-the-art detectors typically use more sophisticated loss functions; e.g. Focal Loss [3] or MSE with negative hard mining. In addition, the current `create_heatmap` function uses an isotropic Gaussian kernel, which produces circular heatmaps. Generalizing this function to produce elliptical heatmaps that match the shape and heading of each bounding box may improve performance (see Fig. 3). Experiment with these improvements and analyze how each one affects the performance of your detector. Note: make sure you tune the loss hyperparameters to fairly compare between the three techniques.

We emphasize that you will not be graded on the performance of your detector. Instead, we are interested

in the clarity of your report and presentation, and the quality of your analysis. Your report should be 2 to 3 pages in length and your presentation should be 5 minutes long (with an additional 5 minutes for Q&A).

Exploring a direction of similar scope to one of the three proposals above will be sufficient for this part of the assignment. Optionally, you may expand the scope of your project to earn up to 20 additional points. See below for the grading rubric for this part of the assignment.

1. **Report [30 points]**
   - Motivation [5 points]
     - What is the problem being tackled?
     - Why is this problem relevant?
     - Briefly (1 to 2 paragraphs) explain other approaches to this problem
     - Motivate why you chose your approach
   - Techniques [5 points]
     - Describe the intuition for why your approach would help
     - Articulate your approach mathematically / algorithmically
   - Evaluation [10 points]
     - Articulate your evaluation plan mathematically / algorithmically
     - Show compelling qualitative results / comparisons
     - Show compelling quantitative results
   - Limitations [5 points]
     - Are there setups in which your approach fails?
     - Are there potential paths of future work to improve your approach?
2. **Presentation [10 points]**
   - Clarity of presentation visualizations, illustrations, etc)
   - Conciseness (motivate, describe and show results of your approach within the 5 minutes)
   - Preparedness (flow of presentation, answers to Q&A, etc)
3. **Bonus [20 points]:** At the discretion of TA's and Instructor considering breadth (explore multiple topics), depth (explore a topic beyond surface level), or novelty (explore your own idea) of exploration.

References

[1] 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 *2020 IEEE/CVF Conference on Computer Vision and Pattern Recognition, CVPR 2020, Seattle, WA, USA, June 13-19, 2020*, pages 11618–11628. Computer Vision Foundation / IEEE, 2020.

[2] Alex H. Lang, Sourabh Vora, Holger Caesar, Lubing Zhou, Jiong Yang, and Oscar Beijbom. Pointpillars: Fast encoders for object detection from point clouds. In *IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2019, Long Beach, CA, USA, June 16-20, 2019*, pages 12697–12705. Computer Vision Foundation / IEEE, 2019.

[3] Tsung-Yi Lin, Priya Goyal, Ross B. Girshick, Kaiming He, and Piotr Dollár. Focal loss for dense object detection. In *IEEE International Conference on Computer Vision, ICCV 2017, Venice, Italy, October 22-29, 2017*, pages 2999–3007. IEEE Computer Society, 2017.

[4] Pengchuan Xiao, Zhenlei Shao, Steven Hao, Zishuo Zhang, Xiaolin Chai, Judy Jiao, Zesong Li, Jian Wu, Kai Sun, Kun Jiang, Yunlong Wang, and Diange Yang. Pandaset: Advanced sensor suite dataset for autonomous driving. *CoRR*, abs/2112.12610, 2021.