GROUP 05: Report Project 2 Problem 3

Carl Brander Tanguy Dieudonné Han Xi

ETH Zürich, Switzerland

Abstract

In this paper, we present three methods to enhance the mean Average Precision (mAP) performance of the refinement stage in a two-stage 3D object detector for autonomous driving. Specifically, we explore the implementation of Generalized Intersection over Union (GIoU) loss, Soft Non-Maximum Suppression (Soft-NMS), and self-attention (SA) layers. These methods are evaluated against a baseline model using the CodaLab test set, with comparisons made across the mAP easy, moderate, and hard metrics.
Among the three approaches, the self-attention model yields the highest improvement, achieving a 4.37% increase in mAP moderate compared to the baseline. Our findings demonstrate the potential of these methods to significantly enhance the accuracy and robustness of 3D object detection systems.

1. Introduction

Object detection is a crucial task in the autonomous driving domain. Being able to perform this in the 3D space might provide us with the best results as it is one of the richest sources of data. Current 3D object detection architectures can be grouped into one-stage and two-stage architectures. One-stage architectures, like [7] [9] [10], attempt to simultaneously predict both the object class and the bounding box in a single pass, offering faster inference times but at the cost of lower precision. On the other hand, two-stage architectures, such as PointRCNN [6], PV-RCNN [5] , VoxelNet [11] and PointNeXT [3], first generate region proposals and then refine these proposals for more accurate classification and bounding box regression, often achieving higher accuracy at the expense of slower inference.
The baseline model in Exercise 2 is a simplified adaptation of the two-stage model PointRCNN, which leverages PointNet++ [2] for point cloud feature extraction and separates the classification and regression tasks into distinct heads. The baseline method is done in a naive way, as it directly concatenates the point coordinates with the features from the first-stage decoder network, and does not implement the canonical transformation to transform the points in the proposals from the LiDAR coordinate system to the canonical coordinate system of the corresponding 3D proposal. This does not fully exploit the potential of feature representations. As a result, these limitations open up many possible directions for improvement.

To improve upon the baseline, we introduce the GIoU loss to handle non-overlapping bounding boxes more effectively while maintaining scale invariance. We also apply Soft-NMS, which replaces standard NMS by decaying detection scores for overlapping objects instead of outright suppression, reducing the risk of false negatives. Lastly, we integrate a self-attention layer into the network to capture long-range dependencies and refine feature representations.

2. Related Work

The **Generalized Intersection over Union** (GIoU) [4] extends the conventional IoU metric by considering the smallest enclosing box, thereby addressing issues related to non-overlapping boxes during bounding-box regression. This enhancement is particularly valuable for improving localization in complex environments, such as 3D point clouds.

Soft-NMS [1] offers an alternative to traditional non-maximum suppression by decaying the confidence scores of overlapping bounding boxes instead of discarding them entirely, leading to more nuanced detections and reduced false negatives.

Furthermore, **Self-Attention mechanisms** [8], popularized in transformer architectures, enable networks to capture long-range dependencies and contextual relationships. This is especially pertinent for point cloud data, where spatial relationships play a crucial role in understanding object boundaries and interactions. By integrating these techniques into 3D object detection pipelines, we aim to address challenges related to precise localization, false positive suppression, and contextual understanding, ultimately enhancing detection performance in point cloud environments.

3. Method

This section outlines the methods we adopted and implemented to address the problem. For each method, we train the model for 35 epochs using the same set of hyperparameters to ensure consistency in evaluation and comparison. Furthermore, each method is implemented separately on the basis of the baseline to isolate their effects.

3.1. Generalized IoU Loss

To improve the handling of non-overlapping bounding boxes and maintain scale invariance, we replace the standard IoU with the GIoU. Unlike IoU, which only considers overlapping areas between bounding boxes, GIoU accounts for the entire encompassing area, offering a more robust evaluation metric. Additionally, this replacement improves the loss computation, making it more sensitive to the spatial alignment of non-overlapping boxes. The GIoU pseudocode is shown in Algorithm 1.

Algorithm 1: Generalized Intersection over Union (GIoU)

Input: Predicted bounding box B_p, Ground-truth bounding box B_g
Output: *GIoU*

Step 1: Calculate Intersection and Union
$B_{\text{int}} \leftarrow B_p \cap B_g$; `// Intersection area`
$B_{\text{union}} \leftarrow B_p \cup B_g$; `// Union area`
Step 2: Compute IoU
IoU $\leftarrow \frac{|B_{\text{int}}|}{|B_{\text{union}}|}$; `// Standard IoU`
Step 3: Find the Smallest Enclosing Box
$B_c \leftarrow$ smallest enclosing box of B_p and B_g ; `// Minimum bounding box containing both`
Step 4: Compute GIoU
GIoU $\leftarrow$ IoU $- \frac{|B_c \setminus B_{\text{union}}|}{|B_c|}$; `// GIoU includes non-overlapping area`
return GIoU

3.2. Soft NMS

In task 2.5, we initially apply Non-Maximum Suppression (NMS) to reduce the number of predicted bounding boxes per frame. NMS operates by sorting all bounding boxes based on their confidence scores and selecting the bounding box with the highest score ($\mathcal{M}$). It then suppresses other bounding boxes with significant overlap, based on a predefined threshold. This process is recursively applied to all remaining boxes.

Despite its effectiveness, NMS has a notable flaw: it entirely eliminates any bounding boxes that meet the overlap threshold, which can result in the loss of valuable predictions. For instance, in Figure 1, two overlapping boxes (green and red) are shown with confidence scores of 0.80 and 0.95, respectively. Instead of eliminating the green box outright, it may be more advantageous to reduce its confidence score to a lower value, such as 0.4.

To address this limitation, we employ Soft-NMS [1]. Instead of removing overlapping boxes, Soft-NMS adjusts their confidence scores based on their degree of overlap with $\mathcal{M}$. Specifically, the suppression is calculated as a function of overlap, f, which we define as a Gaussian function in our implementation, with a deviation of 0.05. This modification allows overlapping objects to remain in the prediction set with reduced confidence. Figure 2 illustrates the algorithmic differences between NMS (red) and Soft-NMS (green), highlighting the improved approach we adopt for task 3.

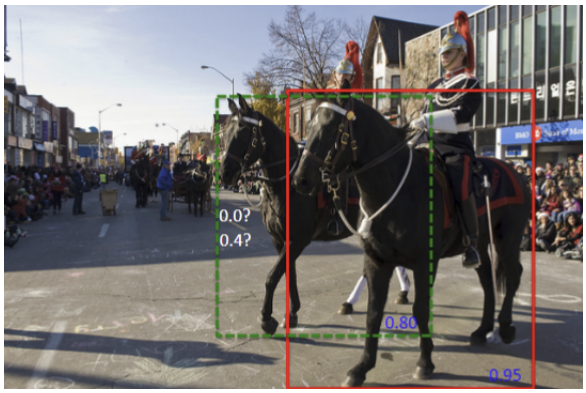

Figure 1. There are two green and red overlapping detection boxes with confidence 0.80 and 0.95 respectively. Is it better to eliminate the green box and set its confidence to 0 or slightly lower it to 0.4? (figure taken from the paper [1])

3.3. Attention

To further refine the final feature representations, we integrate a self-attention layer before the classification and regression heads. This self-attention mechanism captures long-range dependencies and interactions between features, enhancing the model's capacity to discriminate subtle variations. Importantly, placing the self-attention layer late in the architecture ensures minimal interference with earlier layers, thereby preserving the learned feature patterns. This addition effectively improves the quality of final predictions without destabilizing the overall learning process.

4. Results

In table 1 we compare the baseline with the three described methods. It is clear that the Self-Attention method improves the performance the most compared to the baseline, especially for harder samples. We obtain an improvement of

Input : $\mathcal{B} = \{b_1, .., b_N\}, \mathcal{S} = \{s_1, .., s_N\}, N_t$
$\mathcal{B}$ is the list of initial detection boxes
$\mathcal{S}$ contains corresponding detection scores
N_t is the NMS threshold

begin
  $\mathcal{D} \leftarrow \{\}$
  while $\mathcal{B} \neq$ *empty* **do**
    $m \leftarrow$ argmax $\mathcal{S}$
    $\mathcal{M} \leftarrow b_m$
    $\mathcal{D} \leftarrow \mathcal{D} \bigcup \mathcal{M}; \mathcal{B} \leftarrow \mathcal{B} - \mathcal{M}$
    for b_i *in* $\mathcal{B}$ **do**
      if $iou(\mathcal{M}, b_i) \geq N_t$ **then**
        $\mathcal{B} \leftarrow \mathcal{B} - b_i; \mathcal{S} \leftarrow \mathcal{S} - s_i$
      end NMS
      $s_i \leftarrow s_i f(iou(\mathcal{M}, b_i))$ Soft-NMS
    end
  end
  return $\mathcal{D}, \mathcal{S}$
end

Figure 2. The pseudocode in red (NMS, task 2.5) is replaced with the one in green (Soft-NMS) for task 3. We use a Gaussian function of overlap for f in our experiment.

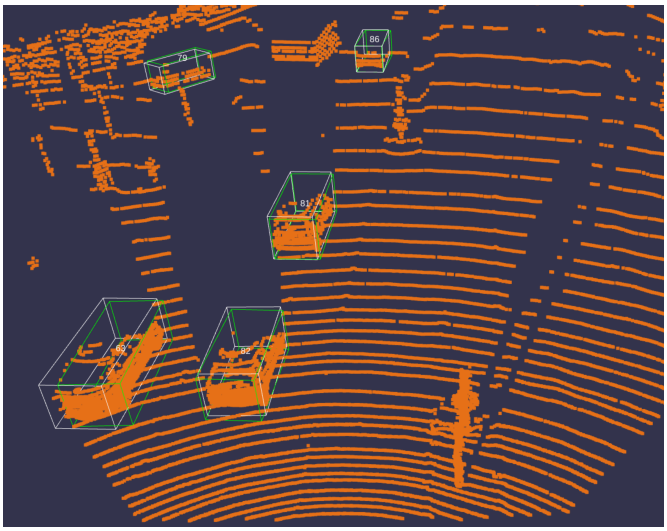

Figure 3. Visualization of 3D bounding box predictions results of the **attention layer based model**. Predictions are based on the Validation-Set.

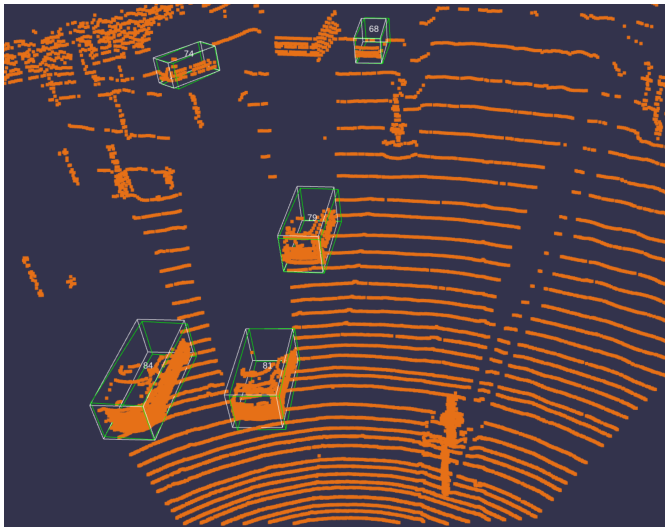

Figure 4. Visualization of 3D bounding box predictions results of the **baseline model** from Task 2.6. Predictions are based on the Validation-Set.

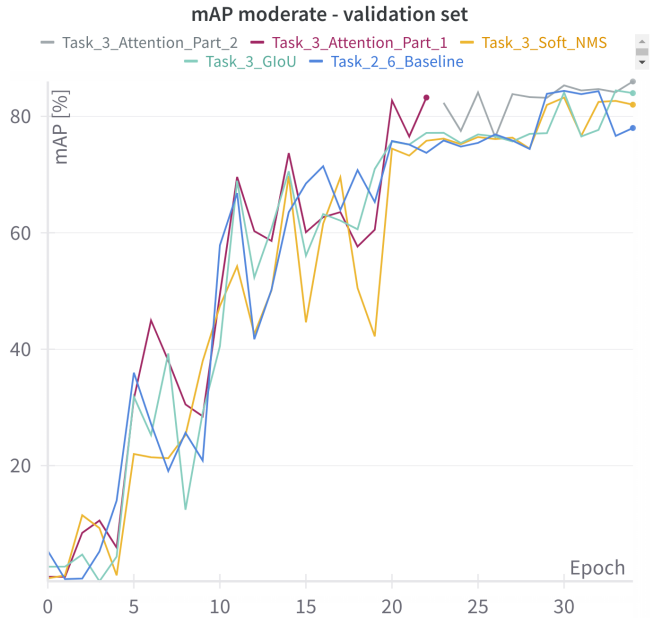

Figure 5. mAP performance on moderate samples during training for each model based on the Validation-Set data. Note an interruption of training for the Attention-based model due to a crash which was resumed on its latest checkpoint.

4.37% for medium difficulty and 0.27% for the hard one. GIoU and Soft-NMS however show weaker performance, although not by a big margin.

The mAP performance results of the three suggested improvements as seen in table 1 show that only the Attention-based model increased the performance compared to the Baseline from Task 2.6. It improved the mAP of moderate difficulty by 4.37% while approximately tying the Baseline's performance on easy and hard examples.

Figure 5 shows the small variance between the models during training while some training noise was observed. Com-

Model	mAP easy	mAP moderate	mAP hard
Baseline	**85.25**	75.04	72.66
Soft NMS	81.68	73.15	70.44
GIoU	81.45	72.66	71.09
Attention	85.05	**79.41**	**72.96**

Table 1. Comparison of mAP performance on easy, moderate and hard detections in percent using the proposed improvements compared to the baseline of Task 2.6. Measured on the Test-Set data from CodaLab.

paring the example predictions of both the baseline model from Task 2.6 in Figure 4 and the attention-based model in Figure 3 only small improvements can be made out.
One can observe, that the baseline model seems to do a far better job in the bottom left car prediction while the attention-based model has a better consistency of prediction throughout the other four predicted cars.

5. Conclusion

In conclusion, our findings highlight the significant potential of optimizing second-stage point-cloud 3D bounding box detectors through an attention-based implementation. The observed performance improvements from this method, coupled with the growing efficacy of attention-based vision models, underscore its promise for future advancements.
A key avenue for future work is incorporating attention layers during the input feature processing stage. This could help focus on critical regions in noisy or sparse point clouds, enhancing downstream performance by refining the input before deeper layers.
Additionally, combining the optimized attention-based model with other complementary approaches, such as skip-connections, could further elevate performance.
Finally, we hypothesize that the limited improvements from GIoU and Soft-NMS are due to the characteristics of the test set, which may lack sufficient scenarios—such as crowded scenes with multiple cars or significant occlusions—where these methods typically excel.

References

[1] Navaneeth Bodla, Bharat Singh, Rama Chellappa, and Larry S. Davis. Soft-nms – improving object detection with one line of code, 2017. 1, 2

[2] Charles R. Qi, Li Yi, Hao Su, and Leonidas J. Guibas. Pointnet++: Deep hierarchical feature learning on point sets in a metric space, 2017. 1

[3] Guocheng Qian, Yuchen Li, Houwen Peng, Jinjie Mai, Hasan Abed Al Kader Hammoud, Mohamed Elhoseiny, and Bernard Ghanem. Pointnext: Revisiting pointnet++ with improved training and scaling strategies, 2022. 1

[4] Hamid Rezatofighi, Nathan Tsoi, JunYoung Gwak, Amir Sadeghian, Ian Reid, and Silvio Savarese. Generalized intersection over union: A metric and a loss for bounding box regression, 2019. 1

[5] Shaoshuai Shi, Chaoxu Guo, Li Jiang, Zhe Wang, Jianping Shi, Xiaogang Wang, and Hongsheng Li. Pv-rcnn: Point-voxel feature set abstraction for 3d object detection, 2021. 1

[6] Shaoshuai Shi, Xiaogang Wang, and Hongsheng Li. Pointrcnn: 3d object proposal generation and detection from point cloud, 2019. 1

[7] Zhi Tian, Xiangxiang Chu, Xiaoming Wang, Xiaolin Wei, and Chunhua Shen. Fully convolutional one-stage 3d object detection on lidar range images, 2022. 1

[8] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. Attention is all you need, 2023. 1

[9] Honghui Yang, Wenxiao Wang, Minghao Chen, Binbin Lin, Tong He, Hua Chen, Xiaofei He, and Wanli Ouyang. Pvt-ssd: Single-stage 3d object detector with point-voxel transformer, 2023. 1

[10] Xingyi Zhou, Dequan Wang, and Philipp Krähenbühl. Objects as points, 2019. 1

[11] Yin Zhou and Oncel Tuzel. Voxelnet: End-to-end learning for point cloud based 3d object detection, 2017. 1

Appendix 3: Code for Task 3

Code used for Method 3.1 - GIoU

```
import torch
import torch.nn as nn

class GIoULoss(nn.Module):
    def __init__(self, config):
        super().__init__()
        self.config = config
        self.eps = 1e-7

    def calculate_giou(self, pred_boxes, target_boxes):
        """
        Calculate GIoU for 3D boxes represented as
        (x, y, z, l, w, h, theta)
        Args:
            pred_boxes: (N, 7) predicted boxes
            target_boxes: (N, 7) target boxes
        Returns:
            giou: (N,) generalized IoU values
        """
        # Extract coordinates and dimensions
        pred_center = pred_boxes[:, :3]
        pred_dims = pred_boxes[:, 3:6]
        target_center = target_boxes[:, :3]
        target_dims = target_boxes[:, 3:6]

        # Calculate corners for intersection
        pred_min = pred_center - pred_dims/2
        pred_max = pred_center + pred_dims/2
        target_min = target_center - target_dims/2
        target_max = target_center + target_dims/2

        # Calculate intersection
        intersect_min = torch.max(pred_min, target_min)
        intersect_max = torch.min(pred_max, target_max)
        intersect_dims =
            torch.clamp(intersect_max - intersect_min, min=0)
        intersection = intersect_dims[:, 0]
            * intersect_dims[:, 1]
            * intersect_dims[:, 2]

        # Calculate volumes
        pred_volume = pred_dims[:, 0] * pred_dims[:, 1]
        * pred_dims[:, 2]
        target_volume = target_dims[:, 0]
        * target_dims[:, 1] * target_dims[:, 2]
        union = pred_volume + target_volume - intersection

        # Calculate IoU
        iou = intersection / (union + self.eps)

        # Calculate enclosing box
        enclosing_min = torch.min(pred_min, target_min)
        enclosing_max = torch.max(pred_max, target_max)
        enclosing_dims = enclosing_max - enclosing_min
        enclosing_volume = enclosing_dims[:, 0] * enclosing_dims[:, 1] * enclosing_dims[:, 2]

        # Calculate GIoU
        giou = iou - (enclosing_volume - union) / (enclosing_volume + self.eps)

        return giou

    def forward(self, pred, target, iou):
        """
        Calculate GIoU loss for positive samples only
        Args:
            pred: (N, 7) predicted boxes
            target: (N, 7) target boxes
```

```
            iou: (N,) initial IoU values
        Returns:
            loss: scalar GIoU loss
        """
        # Filter positive samples
        positive_reg_lb = self.config['positive_reg_lb']
        positive_mask = iou >= positive_reg_lb

        if positive_mask.sum() == 0:
            return torch.tensor(0.0, device=pred.device, requires_grad=True)

        pred = pred[positive_mask]
        target = target[positive_mask]

        # Calculate GIoU for positive samples
        giou = self.calculate_giou(pred, target)

        # Convert GIoU to loss (1 - GIoU)
        loss = 1 - giou

        return loss.mean()

------------
Implement the get_giou function in task1.py,
and import it in task5.py for the nms/soft-nms function.
------------

import numpy as np

def get_giou(pred, target):
    '''
    Calculate pairwise Generalized IoU between
    predicted and target 3D bounding boxes
    input
        pred (N,7) 3D bounding box corners
        target (M,7) 3D bounding box corners
    output
        giou (N,M) pairwise 3D generalized
        intersection-over-union
    '''
    # Get the corners
    pred_corners = label2corners(pred)
    target_corners = label2corners(target)

    # Initialize giou matrix
    giou = np.zeros((pred.shape[0], target.shape[0]))

    for i in range(pred.shape[0]):
        for j in range(target.shape[0]):
            # Get min/max corners for intersection calculation
            min_corner_i = np.min(pred_corners[i], axis=0)
            max_corner_i = np.max(pred_corners[i], axis=0)
            min_corner_j = np.min(target_corners[j], axis=0)
            max_corner_j = np.max(target_corners[j], axis=0)

            # Calculate intersection bounds
            intersection_min =
                np.maximum(min_corner_i, min_corner_j)
            intersection_max =
                np.minimum(max_corner_i, max_corner_j)

            # Calculate intersection volume
            dx = max(0, intersection_max[0] - intersection_min[0])
            dy = max(0, intersection_max[1] - intersection_min[1])
            dz = max(0, intersection_max[2] - intersection_min[2])
            intersection = dx * dy * dz

            # Calculate volumes of both boxes
            vol_i = np.prod(max_corner_i - min_corner_i)
            vol_j = np.prod(max_corner_j - min_corner_j)

            # Calculate union
            union = vol_i + vol_j - intersection
```

```
            # Calculate IoU
            iou = intersection / (union + 1e-8)

            # Calculate smallest enclosing box
            enclosing_min = np.minimum(min_corner_i, min_corner_j)
            enclosing_max = np.maximum(max_corner_i, max_corner_j)

            # Calculate volume of enclosing box
            enclosing_volume = np.prod(enclosing_max - enclosing_min)

            # Calculate GIoU
            giou[i,j] = iou - (enclosing_volume - union) / (enclosing_volume + 1e-8)

    return giou
```

Code used for Method 3.2 - Soft-NMS

```python
import numpy as np
import os
from utils.task1 import get_iou

def nms(pred, score, threshold, sigma=0.05):
    '''
    Implemented Soft-NMS to reduce the number of
    predictions per frame.
    input:
        pred (N,7)      3D bounding box with (x,y,z,h,w,l,ry)
        score (N,)      confidence scores
        threshold (float) upper bound threshold for suppression
        sigma (float)   Gaussian sigma for score decay
    output:
        s_f (M,7)       3D bounding boxes after Soft-NMS
        c_f (M,1)       corresponding confidence scores
    '''
    # set z to 0 and h to 1 for BEV boxes
    bev_boxes = pred.copy()
    bev_boxes[:, 1] = 0
    bev_boxes[:, 3] = 1

    # Sort the scores in descending order
    sorted_indices = np.argsort(score)[::-1]

    # Initialize the list for selected indices
    keep_indices = []

    while len(sorted_indices) > 0:
        # Take the box with the highest score
        current_index = sorted_indices[0]
        keep_indices.append(current_index)

        # Current box and score
        current_box = bev_boxes[current_index].reshape(1, 7)
        current_score = score[current_index]

        # List of indices to update or keep
        remaining_indices = []

        for j in sorted_indices[1:]:
            other_box = bev_boxes[j].reshape(1, 7)
            # Get scalar IoU value
            iou = get_iou(current_box, other_box)[0][0]

            # Apply Gaussian decay to the score of the other box
            score[j] *= np.exp(-(iou ** 2) / sigma)

            # Keep box if its score is above the threshold
            if score[j] > 1e-3:
                remaining_indices.append(j)

        # Update the sorted indices with remaining indices
        sorted_indices = np.array(remaining_indices)

    # Collect the remaining boxes and scores
    s_f = pred[keep_indices]
    c_f = np.expand_dims(score[keep_indices], axis=1)

    return s_f, c_f
```

Code used for Method 3.3 - Self-Attention

```
# Course: Computer Vision and Artificial Intelligence
for Autonomous Cars, ETH Zurich
# Material for Project 2

import torch
import torch.nn as nn
import torch.nn.functional as F
from pointnet2_ops.pointnet2_modules import PointnetSAModule

class SelfAttentionLayer(nn.Module):
    def __init__(self, channel):
        super().__init__()
        self.query_conv = nn.Conv1d(channel, channel // 8, kernel_size=1)
        self.key_conv = nn.Conv1d(channel, channel // 8, kernel_size=1)
        self.value_conv = nn.Conv1d(channel, channel, kernel_size=1)
        self.gamma = nn.Parameter(torch.zeros(1))
        self.softmax = nn.Softmax(dim=-1)

    def forward(self, x):
        batch_size, channels, num_points = x.size()

        # Generate query, key, and value
        query = self.query_conv(x).permute(0, 2, 1)  # (B, N, C/8)
        key = self.key_conv(x)  # (B, C/8, N)
        value = self.value_conv(x)  # (B, C, N)

        # Scaled Dot-Product Attention
        scale_factor = key.size(1) ** -0.5
        # Scaled energy (B, N, N)
        energy = torch.bmm(query, key) * scale_factor
        attention = self.softmax(energy)

        # Weighted aggregation (B, C, N)
        out = torch.bmm(value, attention.permute(0, 2, 1))

        # Residual connection and scaling
        out = self.gamma * out + x
        return out


# Modified Model with Attention
class Model(nn.Module):
    def __init__(self, config):
        super().__init__()
        self.__dict__.update(config)
        print("Model with Self Attention Layer")
        # Encoder (same as original)
        channel_in = self.channel_in
        self.set_abstraction = nn.ModuleList()
        for k in range(len(self.npoint)):
            mlps = [channel_in] + self.mlps[k]
            npoint = self.npoint[k] if self.npoint[k]!=-1 else None
            self.set_abstraction.append(
                    PointnetSAModule(
                        npoint=npoint,
                        radius=self.radius[k],
                        nsample=self.nsample[k],
                        mlp=self.mlps[k],
                        use_xyz=True,
                        bn=False
                    )
                )
            channel_in = mlps[-1]

        # Add Self-Attention Layer
        self.self_attention1 = SelfAttentionLayer(channel_in)
        self.self_attention2 = SelfAttentionLayer(channel_in)

        # Classification head (same as original)
        cls_layers = []
```

```
        pre_channel = channel_in
        for k in range(len(self.cls_fc)):
            cls_layers.extend([
                nn.Conv1d(
                pre_channel,
                self.cls_fc[k],
                kernel_size=1),
                nn.ReLU(inplace=True)
            ])
            pre_channel = self.cls_fc[k]
        cls_layers.extend([
            nn.Conv1d(pre_channel, 1, kernel_size=1),
            nn.Sigmoid()
        ])
        self.cls_layers = nn.Sequential(*cls_layers)

        # Regression head (same as original)
        det_layers = []
        pre_channel = channel_in
        for k in range(len(self.reg_fc)):
            det_layers.extend([
                nn.Conv1d(
                pre_channel,
                self.reg_fc[k],
                kernel_size=1),
                nn.ReLU(inplace=True)
            ])
            pre_channel = self.reg_fc[k]
        det_layers.append(nn.Conv1d(pre_channel, 7, kernel_size=1))
        self.det_layers = nn.Sequential(*det_layers)

    def forward(self, x):
        xyz = x[..., 0:3].contiguous() # (B,N,3)
        feat = x[..., 3:].transpose(1, 2).contiguous() # (B,C,N)

        for layer in self.set_abstraction:
            xyz, feat = layer(xyz, feat)

        # Add self-attention
        feat = self.self_attention2(feat)

        pred_class = self.cls_layers(feat).squeeze(dim=-1) # (B,1)
        pred_box = self.det_layers(feat).squeeze(dim=-1) # (B,7)
        return pred_box, pred_class
```