Reimplementing VectorNet

`process_data.py` 处理数据，并保存到 `./processed_data` 目录中；

`train.py` 训练 VectorNet 网络；

`visual.ipynb` 可视化训练结果，对比真值与预测结果；

0. 准备工作

配置好 NuPlan 环境的 Docker 镜像，或者在本地配置 NuPlan 仿真环境：

```
# 拉取 Docker 镜像
docker pull runqiqiu/nuplan_conda:v1

# 创建容器
xhost +local:
docker run -itd \
    --network host \
    --gpus all \
    -e DISPLAY=$DISPLAY \
    --name nuplan_display \
    --shm-size=16g \
    -v /tmp/.X11-unix:/tmp/.X11-unix \
    -v ~/nuplan:/root/nuplan \ # 这里替换为本地的 nuplan 数据集路径
    runqiqiu/nuplan_conda:v1
```

启动 Docker 容器后，需要手动安装一下对应版本的 PyTorch Geometry。推荐安装老版本，参考链接为 https://pytorch-geometric.readthedocs.io/en/2.0.4/notes/installation.html。

```
# 在 https://data.pyg.org/whl/torch-1.9.0%2Bcu111.html 手动下载依赖并安装
pip install ./torch_cluster-1.5.9-cp39-cp39-linux_x86_64.whl
pip install ./torch_scatter-2.0.9-cp39-cp39-linux_x86_64.whl
pip install ./torch_sparse-0.6.12-cp39-cp39-linux_x86_64.whl
pip install ./torch_spline_conv-1.2.1-cp39-cp39-linux_x86_64.whl

# 安装 PyG
pip install torch-geometric==2.0.4
```

1. 数据处理

1.1 从 NuPlan 中获取训练数据

获取场景信息的流程如下：

- 导入必要的库和模块；
- 定义获取的每种类型的场景数量、总场景数量、是否打乱场景顺序等参数；
- 创建保存数据的文件夹；
- 使用 `get_scenario_map` 和 `ScenarioMapping` 创建场景映射对象；

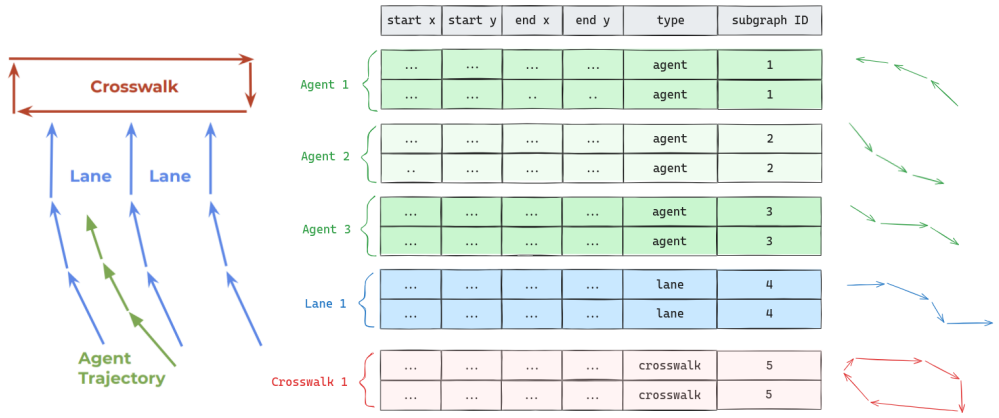
- 使用 `NuPlanScenarioBuilder` 构建一个场景生成器对象，所需参数包括数据路径、地图路径、传感器根路径、数据库文件、地图版本和场景映射；
- 使用 `ScenarioFilter` 和 `get_filter_parameters` 过滤需要的场景；
- 启用 `SingleMachineParallelExecutor` 启用并行处理，提升处理速度；
- 使用场景生成器和并行执行器获取符合过滤条件的场景；

经过上述流程后，可以获得一个装满 `Scenario` 的列表，每个 `Scenario` 都是一帧的场景数据，包含高精度地图信息、全局导航路径、自车状态、周围交通参与者的状态、自车历史轨迹与未来轨迹、周围交通参与者的历史轨迹与未来轨迹。

具体接口可参考 `/nuplan/planning/scenario_builder/abstract_scenario.py`，每个接口都有较为细致的注释；

此外，为了保证网络输入输出的一致性，需要将 Agent 和 Lane 的信息都旋转到自车坐标系来进行处理，NuPlan 同样提供了对应的工具。可参考 `/nuplan/planning/training/preprocessing/features/trajectory_utils.py → convert_absolute_to_relative_poses`

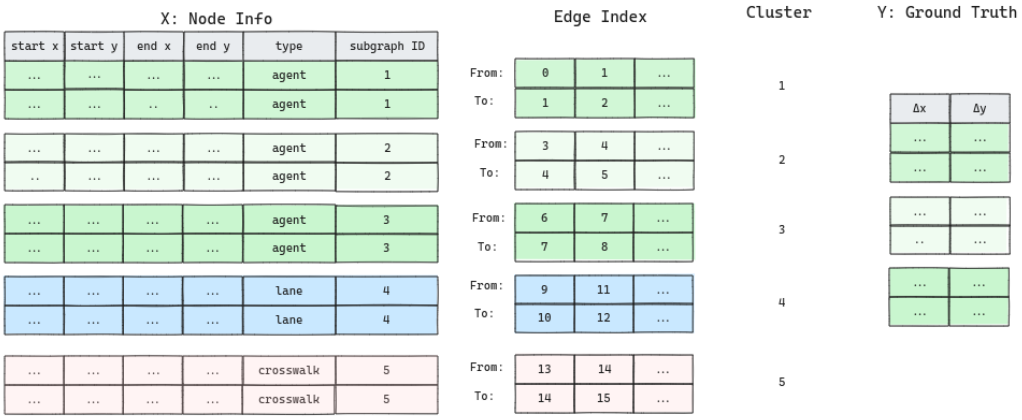
由于 VectorNet 需要的输入是向量形式的，包含向量的起点坐标、终点坐标，以及对应的类别信息（用 One-hot 编码的形式，以浮点数代替字符串来描述属性）。同时，考虑到所有的交通参与者和地图信息都是组合成一个大 Tensor 输入到 VectorNet 中的，因此输入的数据中还应包含一个用于表示向量从属关系的 `Subgraph ID` 属性。



VectorNet 的输出即为 Agent 的未来轨迹，这里为了减少网络的负担，让网络的输出不至于和真实结果有过分的偏差，一个比较简介有效的策略是将 Agent 的未来轨迹描述成增量的形式，即一个 $[\Delta x, \Delta y]$ 的序列。虽然如此操作需要在网络输出结果时多一道用 $[x_0, y_0]$ 和 $[\Delta x_0, \Delta y_0; \cdots; \Delta x_N, \Delta y_N]$ 回溯 $[x_1, y_1; \cdots; x_N, y_N]$ 的工序，但是能保证输出的轨迹都是从 Agent 当前时刻坐标出发的路径点序列，而不至于给出一团乱麻的结果。

1.2 构建 Subgraph

由于 VectorNet 是个图神经网络的结构，那么必不可少的一点就是构建 `subgraph` 的连通关系，并整理成 `PyG` 这个图神经网络工具所需要的输入形式。



2. 网络结构实现

2.1 Polyline Subgraphs

原文中的 Subgraph GNN 每一层的结构是这样的：

```
graph TD
  Input["Input Nodes Features"] --> Node_Encoder

  subgraph Node_Encoder
          direction LR
          FC["Fully Connected layer"] --> LN["Layer Normalization"]
          LN --> ReLu
  end

  Node_Encoder--> Concat
  Node_Encoder--> Aggregator["Aggregator: Max Pooling"]

  Aggregator --> Concat --> output["Output Node Features: Max Pooling"]
```

a. 输入的信息一个 MLP 形式的 Node Encoder 处理；

b. MLP 的结果会兵分两路，一路经过 Max Pooling 提取体征，另一路原封不动的输出；

c. MLP 的两路输出会在 Concat 部分进行 Concatenation 拼接操作，得到一个全新的多维向量，作为当前 GNN 层的输出

利用`PyG`可以根据上述流程快速搭建一个 GNN Layer。：

```
class GraphLayerProp(MessagePassing):
    """继承自 MessagePassing 基类的一层 GNN 的实现在执行 forward 方法时会调用
       propagate 方法，propagate 会顺序调用 message, aggregate 和 update 来
       实现消息传递的全流程

    Args:
        MessagePassing: PyG 提供的基类
    """

    def __init__(self, in_channels, hidden_unit=64, verbose=False):
```

```
        """初始化 GNN 结构中涉及到的 Aggregator 和 Node Encoder 以及调试信息
           输出的标志位

        Args:
            in_channels (int)          : 输入的尺寸
            hidden_unit (int, optional): 隐藏层的尺寸. 缺省为 64.
            verbose (bool, optional)   : 输出调试信息的标志位. 缺省为 False.
        """
        # 初始化 GNN 的聚合操作为 Max Pooling
        super(GraphLayerProp, self).__init__(aggr='max')

        # 初始化输出调试信息的标志位
        self.verbose = verbose

        # 初始化 MLP 形式的 Node Encoder
        # 由三层组成，Fully Connected Layer ->  Layer Normalization -> ReLu
        # FIXME: 这个开源代码多加了一个 Linear 层，不知道会造成什么影响
        self.mlp = nn.Sequential(
            nn.Linear(in_channels, hidden_unit),
            nn.LayerNorm(hidden_unit),
            nn.ReLU(),
            nn.Linear(hidden_unit, in_channels)
        )

    def forward(self, x, edge_index):
        """GNN 层的推理流程，对应论文中的 Figure 3

        Args:
            x         : Polyline Features
            edge_index: Adjacency Array

        Returns: Output Nodes Features
        """
        # 如果 verbose = True，则打印 MLP 之前的 Input Node Feature
        if self.verbose:
            print(f'x before mlp: {x}')

        # 使用先前定义的 MLP 充当 Node Encoder，对 Input Node Feature 进行处理
        x = self.mlp(x)

        # 如果 verbose = True，则打印 MLP 输出的节点特征
        if self.verbose:
            print(f"x after mlp: {x}")

        # 使用 propagate 方法执行 message，update 和 max pooling aggregate
        return self.propagate(edge_index, size=(x.size(0), x.size(0)), x=x)

    def message(self, x_j):
        """Message Passing 的消息生成阶段，根据原论文的意思，我们并没有对传递给
           邻居节点的消息做任何处理。
```

```
        Args:
            x_j: Node feature

        Returns: Exactly the input
        """
        return x_j

    def update(self, aggr_out, x):
        """Concat 阶段，将 MLP 的输出和 Aggregator 的输出拼接在一起

        Args:
            aggr_out: Aggregator output
            x       : Node Encoder output

        Returns: Output Node Features
        """
        # 如果 verbose = True，则打印被 Concat 的两个元素
        # 一个是 Node Encoder output，一个是 Aggregator output
        if self.verbose:
            print(f"x after mlp: {x}")
            print(f"aggr_out: {aggr_out}")

        # 返回拼接后的特征
        return torch.cat([x, aggr_out], dim=1)
```

多层 GNN 的处理方式就相对简单了，只需要将单层 GNN 给堆叠起来即可，上一层 GNN 的输出就是下一层 GNN 的输入。

```
class SubGraph(nn.Module):
    """完整的 Polyline Subgraph

    Args:
        nn: PyTorch 提供的基类
    """

    def __init__(self, in_channels, num_subgraph_layers=3, hidden_unit=64):
        """初始化 Polyline Subgraph 的参数

        Args:
            in_channels (int)                   : 输入的维度
            num_subgraph_layers (int, optional): GNN 的层数. 缺省为 3.
            hidden_unit (int, optional)         : 隐层的维度. 缺省为 64.
        """
        # 初始化 nn.Module
        super(SubGraph, self).__init__()

        # 初始化 subgraph 的层数
        self.num_subgraph_layers = num_subgraph_layers

        # 初始化 GNN 的多个图层
```

```
        self.layer_seq = nn.Sequential()

        # 循环创建多个图层，并将它们塞入 layer_seq 中
        for i in range(num_subgraph_layers):
            # 添加 GraphLayerProp 图层并命名为 glp_i
            self.layer_seq.add_module(
                f'glp_{i}',
                GraphLayerProp(in_channels, hidden_unit)
            )

            # GraphLayerProp 每次输出的节点数量都会 double
            # 所以这里的 in_channels 会一直乘以2
            in_channels *= 2

    def forward(self, sub_data):
        """GNN 的推理过程

        Args:
            sub_data: polyline 特征数据

        Returns: _description_
        """
        # 从 sub_data 中取出向量化的特征 x 和邻接矩阵 edge_index 信息
        x, edge_index = sub_data.x, sub_data.edge_index

        # 遍历 layer_seq 中的所有层进行推理
        for _, layer in self.layer_seq.named_modules():

            # 如果当前层是 GraphLayerProp，则将图数据传给图层并更新节点特征
            if isinstance(layer, GraphLayerProp):
                x = layer(x, edge_index)

        # 用 GNN 的输出结果更新 sub_data
        sub_data.x = x

        # 对输出结果进行 Max Pooling
        out_data = max_pool(sub_data.cluster, sub_data)

        # 确保 Pooling 后的特征维度正确
        assert out_data.x.shape[0] % int(sub_data.time_step_len[0]) == 0

        # 对节点特征进行 L2 正则化并输出
        out_data.x = out_data.x / out_data.x.norm(dim=0)
        return out_data
```

1.2 Global Interaction Graph

全局网络就是在一个全连通图上进行 Self-Attention 操作，并且利用 $\mathrm{softmax}(\cdot)$ 函数来将结果归一化：

$$\mathrm{GNN}(\mathrm{P}) = \mathrm{softmax}(\mathrm{P}_Q \mathrm{P}_K^T)\mathrm{P}_V$$

- $\mathrm{softmax}(\cdot)$ 的原理并不用深究，`pyTorch` 直接提供了对应的接口 `torch.nn.functional.softmax()` 来实现这个功能；
- 式中的 $\mathrm{P}_Q, \mathrm{P}_K, \mathrm{P}_V$ 是 self-attention 的核心概念，分别是查询 (Query)、键 (Key) 和值 (Value) 的投影，它们是 P 通过不同的线性变换得到的。

```
q_lin = nn.Linear(in_channels, global_graph_width)
k_lin = nn.Linear(in_channels, global_graph_width)
v_lin = nn.Linear(in_channels, global_graph_width)
```

- 将三个线性变换和 $\mathrm{softmax}(\cdot)$ 组合，即可得到推理结果：

```
query = self.q_lin(x) # Pq
key   = self.k_lin(x) # Pk
value = self.v_lin(x) # Pv

# 计算 Pq * Pk.T
scores = torch.bmm(query, key.transpose(1, 2))

# 计算 softmax(Pq * Pk.T)
attention_weights = masked_softmax(scores, valid_len)

# 计算 softmax(Pq * Pk.T) * Pv
result = torch.bmm(attention_weights, value)
```

1.3 Prediction MLP

Global Graph 输出的依然是所有节点的特征，而真正将这些特征转化为轨迹的一个多层感知机。结构很简单 Linear ⇒ Linear Norm ⇒ ReLu ⇒ Linear，输入 Agent Node 特征，输出预测的离散点轨迹：

```
class TrajPredMLP(nn.Module):
    """输入 Agent Node Feature，输出预测轨迹的 MLP

    Args:
        nn: PyTorch 的基类
    """

    def __init__(self, in_channels, out_channels, hidden_unit):
        """初始化网络结构

        Args:
            in_channels (int) : 输入的维度
            out_channels (int): 输出的维度
            hidden_unit (int) : 隐层的尺寸
        """
        # 初始化网络
        super(TrajPredMLP, self).__init__()

        # 创建一个 Linear -> LinearNorm -> ReLu -> Linear 结构的 MLP
        self.mlp = nn.Sequential(
            nn.Linear(in_channels, hidden_unit),
            nn.LayerNorm(hidden_unit),
```

```
            nn.ReLU(),
            nn.Linear(hidden_unit, out_channels)
        )

    def forward(self, x):
        """MLP 推理

        Args:
            x (tensor): Agent 节点的特征

        Returns: 预测的轨迹
        """
        return self.mlp(x)
```

1.4 集合成一个 VectorNet

将多个网络集合成一个完整的 VectorNet 的工作也并不复杂，甚至和 MLP 的操作差不多，就是将很多子网络堆叠起来：

```
class VectornetGNN(nn.Module):
    def __init__(
        self,
        in_channels,
        out_channels,
        num_subgraph_layers=3,
        num_global_graph_layer=1,
        subgraph_width=64,
        global_graph_width=64,
        traj_pred_mlp_width=64
    ):
        super(VectornetGNN, self).__init__()
        self.polyline_vec_shape = in_channels * (2 ** num_subgraph_layers)

        # create polyline subgraph
        self.subgraph = SubGraph(
            in_channels,
            num_subgraph_layers,
            subgraph_width
        )

        # create global graph
        self.self_atten_layer = SelfAttentionLayer(
            self.polyline_vec_shape,
            global_graph_width
        )

        # create prediction MLP
        self.traj_pred_mlp = TrajPredMLP(
            global_graph_width,
            out_channels,
            traj_pred_mlp_width
```

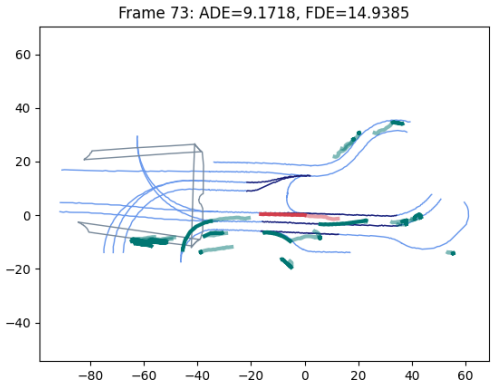
```
        )

    def forward(self, data):
        """
        args:
            data (Data): [x, y, cluster, edge_index, valid_len, batch]
        """
        subgraph_out = self.subgraph(data)

        self_atten_input = subgraph_out.x.view(
            -1,
            config.NUM_GRAPH,
            subgraph_out.num_features
        )

        self_atten_out = self.self_atten_layer(self_atten_input)

        agent_feature = self_atten_out[:, 0:config.NUM_AGENTS+1, :]

        mlp_input = agent_feature.contiguous().view(
            -1,
            agent_feature.shape[2]
        )

        prediction = self.traj_pred_mlp(mlp_input)

        return prediction
```

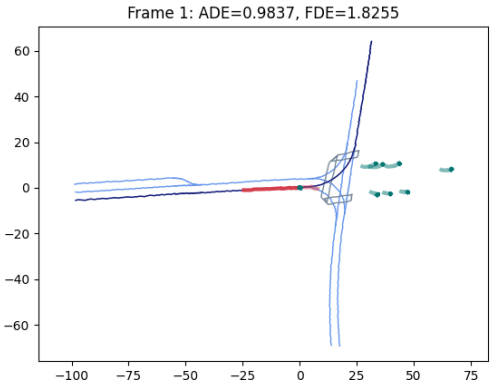
3. 训练结果

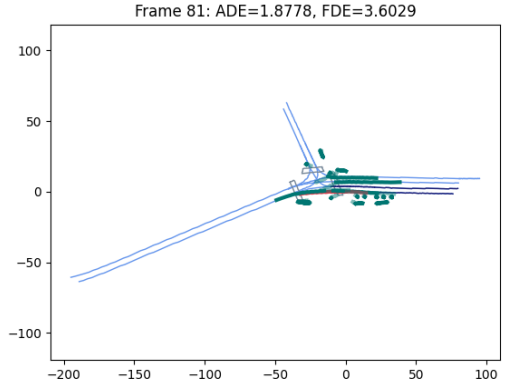
使用如下参数训练网络：

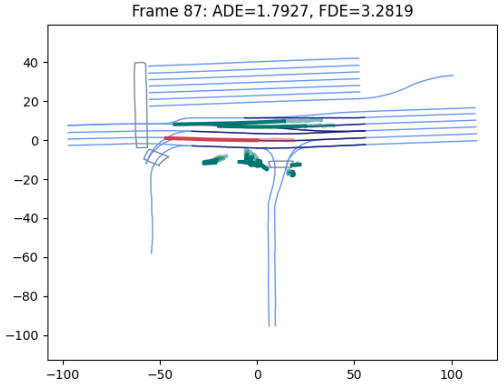
- `SEED` = 999
- `EPOCHS` = 50
- `BATCH_SIZE` = 1
- `LEARNING_RATE` = 0.001

得到的结果如下：






Frame 81: ADE=1.8778, FDE=3.6029
100
50
0
−50
−100
−200
−150
−100
−50
0
50
100


Frame 87: ADE=1.7927, FDE=3.2819
40
20
0
−20
−40
−60
−80
−100
−100
−50
0
50
100