Forecast-MAE[1]&EMP[2]&DeMo[3]

Forecast-MAE[1]是基于自监督学习方法的轨迹预测模型,分为预训练和微调两个阶段，其掩码策略首次将场景 polylines(agent 轨迹 polylines、map polylines)掩码后送入同一个编码器，以此来捕获整个 scene polylines 之间的潜在交互，发表在 ICCV 会议。EMP[2]和 DeMo[3]分别发表在 2024IROS 和 2024NeurIPS 上，从其开源的代码可知，均以 Forecast-MAE[1]为 baseline，甚至 EMP[2]只是在 Forecast-MAE[1]微调模型上简单修改了编码器。到底 Forecast-MAE[1]模型有什么奇妙之处？让我们解析一下。

创新点的提出：

问题：轨迹预测任务需要大量的标签数据，包括但不限于轨迹信息、地图信息；PreTraM[4]提出从局部区域裁剪出一个栅格化地图（类似于 multi-path 内的语义地图）补丁来生成高清地图，通过对比学习来生成强图像编码器来提高数据量；SSL-Lanes，通过 pretext 任务，无需通过额外的数据就可以提高性能指标。轨迹预测涉及到多模态输入，而传统的 SSL 仅涉及图像或者文本的单模态输入；PreTraM 通过图像与文本的对比学习来实现多模态的互联.

解决办法：作者采用 MAE（Masked AutoEncoders）方法，通过屏蔽掉一部分的输入数据，使用自编码器重建缺失部分数据；具体而言：历史轨迹和预测轨迹使用互补的 mask 方法，只 mask 历史轨迹或者只 mask 预测轨迹，随机 mask 车道线。然后将 mask 掉的轨迹和车道线送入基于 transfomer 架构的 Encoder、Decoder 进行重建。掩码示意图如 Figure 1 所示。

优势： 学习历史与未来轨迹的双向关联（如通过历史预测未来，或通过未来反推历史）。通过车道与轨迹的联合掩码，建模跨模态交互（如车道结构如何影响轨迹）

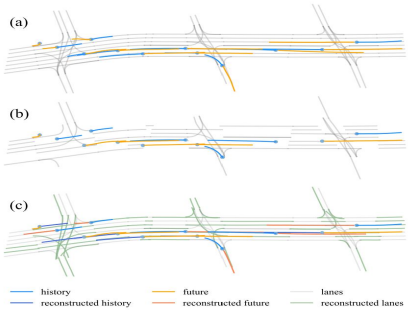

Figure 1. Reconstruction result on Argoverse 2 *validation* scenario. (a) The origin scenario. (b) 50% of agents' trajectory is masked using a complementary masking strategy (either history or future is masked). 50% of the lane segments are masked randomly. (c) Scenario reconstructed by the proposed Forecast-MAE.

Figure 1 掩码示意图

结构与方法

本模型分为预训练和微调两个阶段，预训练框架图如下：

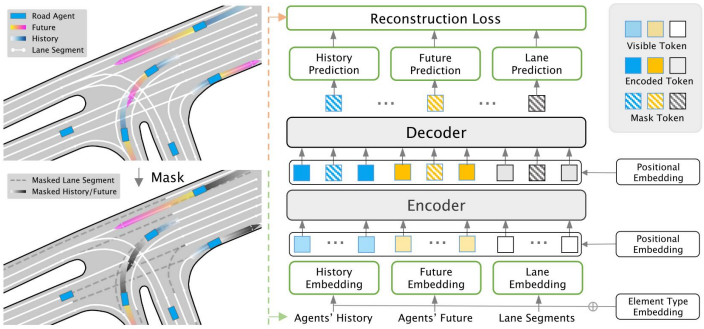

Figure 2 Overall pre-training scheme of our Forecast-MAE

如 Figure 2 所示，预训练阶段输入未被掩码的 Agent History,Agent Future,Lane Segments,其中 Agent History,Agent Future 输入到 FPN 进行编码，Lane Segments 通过 mini-Pointnet 编码，至于 Mask Tokens 则是随机初始化的 Query,Encoded Token 作为 K,V，通过 Decoder 查询得到，具体代码如下：

场景数据获取

```
def forward(self, data):
    hist_padding_mask = data["x_padding_mask"][:, :, :50]#(16,48,50)
    hist_feat = torch.cat(#(16,48,50,4)这个张量拼接上每一个时间步的mask
        [
            data["x"],#(16,48,50,2)
            data["x_velocity_diff"][..., None],#(16,48,50)->(16,48,50,1)
            ~hist_padding_mask[..., None],#(16,48,50)->(16,48,50,1)
        ],
        dim=-1,
    )
    B, N, L, D = hist_feat.shape
    hist_feat = hist_feat.view(B * N, L, D)
    hist_feat = self.hist_embed(hist_feat.permute(0, 2, 1).contiguous())
    hist_feat = hist_feat.view(B, N, hist_feat.shape[-1])#(16,48,128)

    future_padding_mask = data["x_padding_mask"][:, :, 50:]#(16,48,60)
    future_feat = torch.cat([data["y"], ~future_padding_mask[..., None]], dim=-1)#(16,48,60,3)
    B, N, L, D = future_feat.shape
    future_feat = future_feat.view(B * N, L, D)
    future_feat = self.future_embed(future_feat.permute(0, 2, 1).contiguous())
    future_feat = future_feat.view(B, N, future_feat.shape[-1])

    lane_padding_mask = data["lane_padding_mask"]#(16,125,20)
    lane_normalized = data["lane_positions"] - data["lane_centers"].unsqueeze(-2)#(16,125,20,2)
    lane_feat = torch.cat([lane_normalized, ~lane_padding_mask[..., None]], dim=-1)
    B, M, L, D = lane_feat.shape
    lane_feat = self.lane_embed(lane_feat.view(-1, L, D).contiguous())
    lane_feat = lane_feat.view(B, M, -1)
```

掩码处理

```
(
    hist_masked_tokens,
    hist_keep_ids_list,
    hist_key_padding_mask,
    fut_masked_tokens,
    fut_keep_ids_list,
    fut_key_padding_mask,
) = self.agent_random_masking(
    hist_feat,
    future_feat,
    self.actor_mask_ratio,
    future_padding_mask,
    data["num_actors"],
)

lane_mask_ratio = self.lane_mask_ratio
(
    lane_masked_tokens,
    lane_key_padding_mask,
    lane_ids_keep_list,
) = self.lane_random_masking(
    lane_feat, lane_mask_ratio, data["lane_key_padding_mask"]
)

x = torch.cat(##(16, 115, 128)
    [hist_masked_tokens, fut_masked_tokens, lane_masked_tokens], dim=1
)
```

解码部分

```
# decoding
x_decoder = self.decoder_embed(x)#(16, 115, 128)
Nh, Nf, Nl = (
    hist_masked_tokens.shape[1],
    fut_masked_tokens.shape[1],
    lane_masked_tokens.shape[1],
)
assert x_decoder.shape[1] == Nh + Nf + Nl
hist_tokens = x_decoder[:, :Nh]
fut_tokens = x_decoder[:, Nh : Nh + Nf]
lane_tokens = x_decoder[:, -Nl:]

decoder_hist_token = self.history_mask_token.repeat(B, N, 1)#随机初始化(1, 1, 128),
hist_pred_mask = ~data["x_key_padding_mask"]#(16, 48)  取反操作将掩码中的 True（填充位置
for i, idx in enumerate(hist_keep_ids_list):
    decoder_hist_token[i, idx] = hist_tokens[i, : len(idx)]
    hist_pred_mask[i, idx] = False  #以表明这些位置已经被处理并需要保持有效

decoder_fut_token = self.future_mask_token.repeat(B, N, 1)
future_pred_mask = ~data["x_key_padding_mask"]
for i, idx in enumerate(fut_keep_ids_list):
    decoder_fut_token[i, idx] = fut_tokens[i, : len(idx)]
    future_pred_mask[i, idx] = False

decoder_lane_token = self.lane_mask_token.repeat(B, M, 1)
lane_pred_mask = ~data["lane_key_padding_mask"]
for i, idx in enumerate(lane_ids_keep_list):
    decoder_lane_token[i, idx] = lane_tokens[i, : len(idx)]
    lane_pred_mask[i, idx] = False
```

```
x_decoder = torch.cat(
    [decoder_hist_token, decoder_fut_token, decoder_lane_token], dim=1
)
x_decoder = x_decoder + self.decoder_pos_embed(pos_feat)
decoder_key_padding_mask = torch.cat(#(16,221)
    [
        data["x_key_padding_mask"],#(16,48)
        future_padding_mask.all(-1),
        data["lane_key_padding_mask"],
    ],
    dim=1,
)

for blk in self.decoder_blocks:
    x_decoder = blk(x_decoder, key_padding_mask=decoder_key_padding_mask)

x_decoder = self.decoder_norm(x_decoder)
hist_token = x_decoder[:, :N].reshape(-1, self.embed_dim)
future_token = x_decoder[:, N_: 2 * N].reshape(-1, self.embed_dim)
lane_token = x_decoder[:, -M:]

# lane pred loss
lane_pred = self.lane_pred(lane_token).view(B, M, 20, 2)
lane_reg_mask = ~lane_padding_mask
lane_reg_mask[~lane_pred_mask] = False
lane_pred_loss = F.mse_loss(
    lane_pred[lane_reg_mask], lane_normalized[lane_reg_mask]
)
```

从上述代码可以看出，预训练部分的代码包括掩码过程和损失计算，这严重拖慢了预训练的速度（4 块 A10 训练一个 epoch 花费将近二十分钟），但是微调过程主要使用预训练代码中的编码器，解码器使用了 MLP，故整个微调代码参数量很小且推理速度很快。微调阶段的代码如下：

```
def forward(self, data):
    hist_padding_mask = data["x_padding_mask"][:, :, :50]
    hist_key_padding_mask = data["x_key_padding_mask"]
    hist_feat = torch.cat(
        [
            data["x"],
            data["x_velocity_diff"][..., None],
            ~hist_padding_mask[..., None],
        ],
        dim=-1,
    )

    B, N, L, D = hist_feat.shape
    hist_feat = hist_feat.view(B * N, L, D)
    hist_feat_key_padding = hist_key_padding_mask.view(B * N)
    actor_feat = self.hist_embed(
        hist_feat[~hist_feat_key_padding].permute(0, 2, 1).contiguous()
    )
    actor_feat_tmp = torch.zeros(
        B * N, actor_feat.shape[-1], device=actor_feat.device
    )
    actor_feat_tmp[~hist_feat_key_padding] = actor_feat
    actor_feat = actor_feat_tmp.view(B, N, actor_feat.shape[-1])

    lane_padding_mask = data["lane_padding_mask"]
    lane_normalized = data["lane_positions"] - data["lane_centers"].unsqueeze(-2)
    lane_normalized = torch.cat(
        [lane_normalized, ~lane_padding_mask[..., None]], dim=-1
    )
    B, M, L, D = lane_normalized.shape
    lane_feat = self.lane_embed(lane_normalized.view(-1, L, D).contiguous())
    lane_feat = lane_feat.view(B, M, -1)
```

```
x_angles = torch.stack([torch.cos(angles), torch.sin(angles)], dim=-1)
pos_feat = torch.cat([x_centers, x_angles], dim=-1)
pos_embed = self.pos_embed(pos_feat)

actor_type_embed = self.actor_type_embed[data["x_attr"][..., 2].long()]
lane_type_embed = self.lane_type_embed.repeat(B, M, 1)
actor_feat += actor_type_embed
lane_feat += lane_type_embed

x_encoder = torch.cat([actor_feat, lane_feat], dim=1)
key_padding_mask = torch.cat(
    [data["x_key_padding_mask"], data["lane_key_padding_mask"]], dim=1
)

x_encoder = x_encoder + pos_embed
for blk in self.blocks:
    x_encoder = blk(x_encoder, key_padding_mask=key_padding_mask)
x_encoder = self.norm(x_encoder)

x_agent = x_encoder[:, 0]
y_hat, pi = self.decoder(x_agent)

x_others = x_encoder[:, 1:N]
y_hat_others = self.dense_predictor(x_others).view(B, -1, 60, 2)

return {
    "y_hat": y_hat,
    "pi": pi,
    "y_hat_others": y_hat_others,
}
```

与 EMP/Demo 的对比

EMP[2]抓住 Forecast-MAE[1]微调部分的天然优势，将微调部分的解码器替换为 DETR-LIKE 类的解码器，在增加些许参数量和推理速度的前提下提高预测精度，具体代码可以自查。EMP[2]以小参数量，低推理速度，可实际部署的优势发表在 IROS 会议上。可以看出，一篇顶会的产生，不需要在算法和想法上有多大创新，重点是抓住会议的要求，并根据这些要求讲一个故事明显凸显自己的优势。那 Forecast-MAE[1]又与顶会 DeMo[3]有什么关系？相较于 Forecast-MAE[1]微调部分的代码，DeMo 增加的代码如下所示：

```
# unidirectional mamba
actor_feat = self.hist_embed_mlp(hist_feat[hist_feat_key_valid].contiguous())
residual = None
for blk_mamba in self.hist_embed_mamba:
    actor_feat, residual = blk_mamba(actor_feat, residual)
fused_add_norm_fn = rms_norm_fn if isinstance(self.norm_f, RMSNorm) else layer_norm_fn
actor_feat = fused_add_norm_fn(
    self.drop_path(actor_feat),
    self.norm_f.weight,
    self.norm_f.bias,
    eps=self.norm_f.eps,
    residual=residual,
    prenorm=False,
    residual_in_fp32=True
)
```

```
#  intra-interaction learning for scene context features
for blk in self.blocks:
    x_encoder = blk(x_encoder, key_padding_mask=~key_valid_mask)
x_encoder = self.norm(x_encoder)

###### Trajectory decoding with decoupled queries ######
new_y_hat = None
new_pi = None
dense_predict = None
mode = None

# outputs of other agents
x_others = x_encoder[:, 1:N]
y_hat_others = self.dense_predictor(x_others).view(B, x_others.size(1), -1, 2)

# state query initialization
time = torch.arange(60).long().to(x_encoder.device)
time = time * 0.1 + 0.1
time = time.unsqueeze(-1)
mode = self.time_embedding_mlp(time)
mode = mode.repeat(x_encoder.size(0), 1, 1)

# decoder module with decoupled queries
dense_predict, y_hat, pi, x_mode, new_y_hat, new_pi, mode_dense, scal, scal_new = \
self.time_decoder(mode, x_encoder, mask=~key_valid_mask)
```

从 Demo[]论文中的性能指标来看，仅仅这一小部分代码，为什么会让预测性能提升这么多？主要原因有以下几点：

1.mode queries capturing distinct directional intentions and state queries tracking the agent’s dynamic statesover time.这一思路不再是只用 query 查询场景上下文直接得到一条轨迹，而是先利用 query 得到方向意图再在此基础上利用 query 回归出与意图点相关的轨迹，这个思路很清奇。

2.充分利用 mamba 对序列建模的优势。每条轨迹的轨迹点设为 Query,假如有 60 个轨迹点就有 60 个 Query,这些 Query 与作为 K,V 的场景上下文交互后，每个轨迹点就找到自己再场景中的位置，再通过 mamba 建立 60 个轨迹点之间的逻辑信息。这样，轨迹的精确度大大提升。

总结

轨迹预测领域的论文大多数都是相通的，想发好的的论文，可以从以下几个方面着手：

1.蹭热点：现在的 VLM，LLM 可以用在轨迹预测中，一旦加入，只要精度不是特差，就有了发高水平论文的底气。轨迹预测只是自动驾驶技术栈里面的一小部分，加入 LLM 等也有点大材小用，但是能将不同领域的东西结合好也是一种创新，虽然并不实用。

2.几行代码将 baseline 变得精度更高，推理速度更快，也可以发论文，论文的质量取决于你对背后原理的解释清晰度，解释的越有道理，论文的质量越高。

3.上述轨迹预测方法，输入的场景数据为轨迹 polylines 和 map polylines，polylines 是折线，折线就是向量，个人建议此方向的同学可以去研究端到端轨迹预测也可以换方向，因为这个方向局限性太大（方向小众且实际应用价值不大），如果你只是想发论文毕业，那忽略这段话。

参考文献

[1]Cheng, J., Mei, X., & Liu, M. (2023). Forecast-MAE: Self-supervised Pre-training for Motion Forecasting with Masked Autoencoders. 2023 IEEE/CVF International Conference on Computer Vision (ICCV), 8645-8655.

[2]Prutsch, Alexander et al.”Efficient Motion Prediction: A Lightweight & Accurate Trajectory Prediction Model With Fast Training and Inference Speed”. IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS). 2024.

[3]Zhang, B., Song, N., & Zhang, L. (2024). DeMo: Decoupling Motion Forecasting into Directional Intentions and Dynamic States.ArXiv, abs/2410.05982.

[4]Chenfeng Xu, Tian Li, Chen Tang, Lingfeng Sun et al.Pretram: Self-supervised pre-training via connecting trajec-tory and map. In Computer Vision–ECCV 2022: 17th Eu-ropean Conference, Tel Aviv, Israel, October 23–27, 2022,Proceedings, Part XXXIX, pages 34–50. Springer, 2022. 2