



深度学习

hw2 实验报告

姓 名: 穆新宇
学 号: 2024213660
班 级: 软硕 42
学 院: 软件学院

2024 年 11 月 24 日

目录

目录 Table of Contents	I
1 Modeling World Dynamics with Recurrent Neural Networks	1
1.1 Define and train an LSTM-based world model	1
1.1.1 代码实现	1
1.1.2 训练	1
1.2 Implement an LSTM layer from scratch	1
1.2.1 代码实现	1
1.2.2 训练	3
1.3 Evaluate Model with Two Rollout Strategies	3
1.3.1 代码实现	3
1.3.2 结果分析	4
1.4 Question Answering	5
1.4.1 Solution1: Scheduled Sampling	5
1.4.2 Solution2: 加权时间步损失	5
1.4.3 Solution3: Data Augmentation	5
2 Graph Neural Networks (GNN)	5
2.1 Task A	5
2.1.1 GCN	5
2.1.2 GAT	6
2.1.3 GraphSAGE	7
2.2 Task B	8
2.2.1 代码实现	8
2.2.2 训练	9

1 Modeling World Dynamics with Recurrent Neural Networks

1.1 Define and train an LSTM-based world model

1.1.1 代码实现

代码 1: WorldModel 的实现

```
1 class WorldModel(nn.Module):
2     def __init__(self, action_size, hidden_size, output_size):
3         super(WorldModel, self).__init__()
4         self.cnn = nn.Sequential(
5             nn.Conv2d(3, 16, kernel_size=3, stride=2, padding=1),
6             nn.ReLU(),
7             nn.Conv2d(16, 32, kernel_size=3, stride=2, padding=1),
8             nn.ReLU(),
9             nn.Flatten()
10        )
11        self.cnn_output_size = 32 * 8 * 8
12        self.lstm_input_size = self.cnn_output_size + action_size
13        self.lstm = nn.LSTM(self.lstm_input_size, hidden_size, batch_first=True)
14        self.fc = nn.Linear(hidden_size, output_size)
15
16    def forward(self, state, action, hidden=None):
17        batch_size = state.size(0)
18        cnn_features = self.cnn(state)
19        action = action.view(batch_size, -1)
20        lstm_input = torch.cat([cnn_features, action], dim=1).unsqueeze(1)
21        lstm_out, hidden = self.lstm(lstm_input, hidden)
22        next_state_pred = self.fc(lstm_out.squeeze(1))
23        return next_state_pred, hidden
```

代码1展示了WorldModel类的实现，在构造函数中，定义了cnn的结构，它负责从输入的图像状态(state)中提取特征，减少输入数据的维度以提升计算效率，由两个卷积层组成，激活函数均使用ReLU；lstm使用了Pytorch内置的模块，直接将参数传入即可；全连接层的输入是LSTM最后一层的输出，形状为[batch_size, hidden_size]，输出是下一时刻的状态预测值，形状为[batch_size, hidden_size]

在前向传播中，首先使用cnn提取图像的高级特征，紧接着将其与动作张量按列拼接，加入时间维度后作为lstm的输入，结合hidden进行时间序列建模，最后一步是使用全连接层预测下一状态，将其与hidden一起返回即可。

1.1.2 训练

使用 wandb 绘制损失曲线，如图3所示。

1.2 Implement an LSTM layer from scratch

1.2.1 代码实现

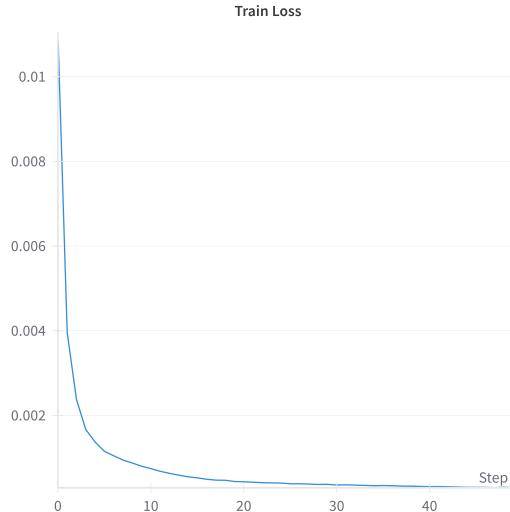


图 1: Train Loss

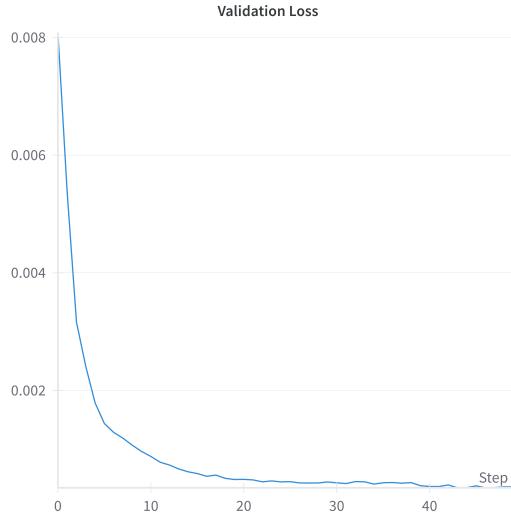


图 2: Validation Loss

图 3: 使用 torch.nn.LSTM 的 loss 曲线

代码 2: WorldModel 的实现 (不使用 torch.nn.LSTM)

```

1 class WorldModel(nn.Module):
2     def __init__(self, action_size, hidden_size, output_size):
3         super(WorldModel, self).__init__()
4         self.action_size = action_size
5         self.hidden_size = hidden_size
6         self.output_size = output_size
7         self.fc_state = nn.Linear(3 * 32 * 32, hidden_size) # Flatten image and project to
8             hidden_size
9         self.fc_action = nn.Linear(action_size, hidden_size) # Project action to hidden_size
10        self.fc_input = nn.Linear(hidden_size * 2, hidden_size * 4) # Input + Action
11        self.fc_output = nn.Linear(hidden_size, output_size) # Project hidden state to output
12
13    def forward(self, state, action, hidden=None):
14        batch_size = state.size(0)
15        state_flat = state.view(batch_size, -1) # [batch_size, 3*32*32]
16        state_proj = self.fc_state(state_flat) # [batch_size, hidden_size]
17        action_proj = self.fc_action(action) # [batch_size, hidden_size]
18        combined = torch.cat([state_proj, action_proj], dim=-1) # [batch_size, hidden_size*2]
19        if hidden is None:
20            h_t = torch.zeros(1, batch_size, self.hidden_size, device=state.device)
21            c_t = torch.zeros(1, batch_size, self.hidden_size, device=state.device)
22        else:
23            h_t, c_t = hidden
24        gates = self.fc_input(combined) # [batch_size, hidden_size*4]
25        i_t, f_t, o_t, g_t = torch.chunk(gates, 4, dim=-1)
26        i_t = torch.sigmoid(i_t)
27        f_t = torch.sigmoid(f_t)
28        o_t = torch.sigmoid(o_t)
29        g_t = torch.tanh(g_t)

```

```

29   c_t = f_t * c_t.squeeze(0) + i_t * g_t # [batch_size, hidden_size]
30   h_t = o_t * torch.tanh(c_t)           # [batch_size, hidden_size]
31   hidden = (h_t.unsqueeze(0), c_t.unsqueeze(0))
32   next_state_pred = self.fc_output(h_t) # [batch_size, output_size]
33   return next_state_pred, hidden

```

代码2的WorldModel手动实现 LSTM 完成对时间序列的建模。构造函数中，将state展平后通过 `fc_state` 投影到隐藏层大小，将action通过 `fc_action` 投影至同一维度后，与状态特征拼接形成输入。随后，通过 `fc_input` 生成 LSTM 的四个门值：输入门、遗忘门、输出门和候选单元状态。

在前向传播中，首先提取状态和动作特征，将二者拼接后计算各门值，分别通过 `sigmoid` 和 `tanh` 激活函数处理，用于更新单元状态 `c_t` 和隐藏状态 `h_t`，实现时间序列建模。最后，通过 `fc_output` 将隐藏状态映射到预测空间，输出下一时刻的状态预测值 `next_state_pred` 和更新后的隐藏状态 `hidden`。

1.2.2 训练

使用 wandb 绘制损失曲线，如图6所示。

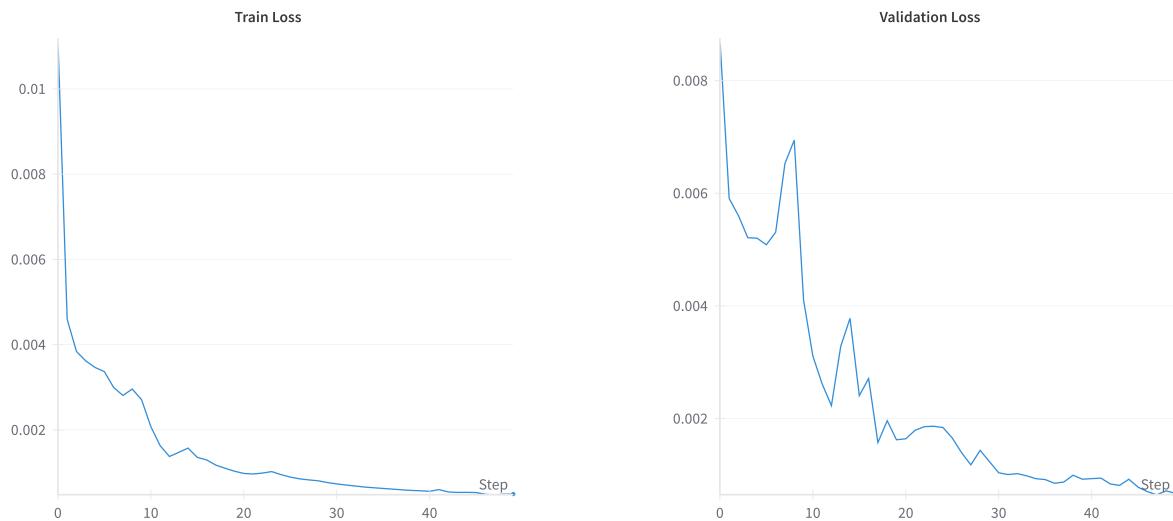


图 4: Train Loss

图 5: Validation Loss

图 6: 自己实现 LSTM 的 loss 曲线

最终的 Test Loss 是 0.0006855568292343782，与内置的 LSTM 相比，减少了约 81.40%。

1.3 Evaluate Model with Two Rollout Strategies

1.3.1 代码实现

Teacher Forcing 在每一步预测中使用真实的下一状态作为输入，模拟模型在理想条件下的预测表现；Autoregressive Rollout 在每一步中使用模型自身的预测作为下一步输入，更接近实际使用场景，测试模型在长期预测中的稳定性和误差累积情况。

移除原有的`test`，编写`test_rollout`函数，如代码3所示。

代码 3: 分别用 Teacher Forcing 和 Autoregressive 策略

```

1 def test_rollout():
2     model. eval()
3     data_loader.set_test()
4     states, actions, _ = data_loader.get_batch()
5     initial_state = states[0][0] # 使用测试集中的一个序列
6     sequence_actions = actions[0]
7     ground_truth_states = states[0]
8     teacher_forcing_predictions = teacher_forcing_rollout(
9         model, initial_state, sequence_actions, ground_truth_states, sequence_length
10    )
11    autoregressive_predictions = autoregressive_rollout(
12        model, initial_state, sequence_actions, sequence_length
13    )
14    steps = sequence_length - 1
15    # 可视化对比
16    plot_rollout_comparison(
17        ground_truth_states[:steps],
18        teacher_forcing_predictions,
19        autoregressive_predictions,
20        steps=steps
21    )
22    # 误差分析
23    mse_teacher_forcing = torch.mean(
24        (ground_truth_states[1:steps + 1].to(device) - teacher_forcing_predictions) ** 2, dim=(1, 2,
25            3)
26    )
26    mse_autoregressive = torch.mean(
27        (ground_truth_states[1:steps + 1].to(device) - autoregressive_predictions) ** 2, dim=(1, 2,
28            3)
28    )
29    # 打印误差并上传到wandb, 这里略去

```

1.3.2 结果分析

使用自己实现的 LSTM 进行训练和验证，测试阶段调用 `test_rollout`，输出如下：

```

1 MSE 每步误差对比:
2 Teacher Forcing: tensor ([0.0018, 0.0008, 0.0011, 0.0007, 0.0008, 0.0009, 0.0008, 0.0008, 0.0008],
3     device='cuda:0')
4 Autoregressive: tensor ([0.0018, 0.0009, 0.0010, 0.0015, 0.0028, 0.0034, 0.0023, 0.0007, 0.0010],
5     device='cuda:0')
6
7 累计误差:
8 Teacher Forcing: 0.008508468046784401
9 Autoregressive: 0.015489230863749981

```

对于逐步误差，可以看出，Teacher Forcing: 每步 MSE 均维持在较低水平（平均约 0.0009），这表明当模型能够获取真实的状态信息时，误差不会累积；而 Autoregressive 初始几步误差接近

Teacher Forcing (前 3 步平均约 0.0012)，但从第 4 步开始，误差逐渐增加，特别是在第 5 步和第 6 步达到峰值。这种现象反映了模型在长时间序列预测中误差累积的趋势。

对于累计误差，Teacher Forcing 累计误差为 0.0085，显著低于 Autoregressive 策略，这说明模型在理想条件下能够保持较高的预测精度；Autoregressive 累计误差为 0.0155，比 Teacher Forcing 高出约 82%，表明误差在长期预测中快速放大，尤其是在中间阶段。

1.4 Question Answering

1.4.1 Solution1: Scheduled Sampling

Scheduled Sampling 是一种训练策略，逐步将模型自身的预测作为输入加入到训练过程中，而不是完全依赖真实的输入。在训练初期，完全使用真实输入；随着训练的进行，逐步增加使用模型自身预测的概率。

Scheduled Sampling 有效的原因很容易理解：模型在训练中经常依赖真实状态作为输入，但在测试中需要使用自身预测，这种训练与测试的差异导致模型在 Autoregressive Rollout 中表现不佳。而 Scheduled Sampling 模拟了测试时的实际条件，使模型学会在自身预测有误时如何恢复，从而减少误差累积。

1.4.2 Solution2: 加权时间步损失

在 Autoregressive Rollout 中，前几步的预测通常较为准确，而后续时间步由于误差累积，预测效果会逐渐恶化，加权损失函数鼓励模型在后续时间步优化预测，减少后期误差对整体性能的影响。

假设序列长度为 T ，模型在每个时间步的预测损失为 Loss_t ，调整后的损失函数为：

$$\text{Weighted Loss} = \sum_{t=1}^T w_t \cdot \text{Loss}_t$$

其中 w_t 是第 t 个时间步的权重，设置成关于 t 的单调增函数即可。

1.4.3 Solution3: Data Augmentation

在训练数据中加入随机扰动，模拟预测输入中的轻微误差，例如在状态和动作上增加随机噪声。

2 Graph Neural Networks (GNN)

2.1 Task A

2.1.1 GCN

为了计算 R^2 得分，需要对提供的代码做一些小修改，如代码4所示。

代码 4: 计算 R^2 得分

```

1 def evaluate(loader, mode="Validation"):
2     # 前面的部分不做修改，略去
3     # 将预测值和真实值拼接为完整数组
4     all_preds = np.concatenate(all_preds, axis=0)
5     all_true = np.concatenate(all_true, axis=0)

```

```

6   if mode == "Test":
7       # 计算 R2 分数
8       r2 = r2_score(all_true, all_preds)
9       wandb.log({"Test R2": r2})
10      print(f"R2 Score on Test Set: {r2:.4f}")
11      return avg_loss

```

设置epoch为 200，不改变提供的代码中的其他参数，利用 wandb 作图，loss 曲线如图9所示。

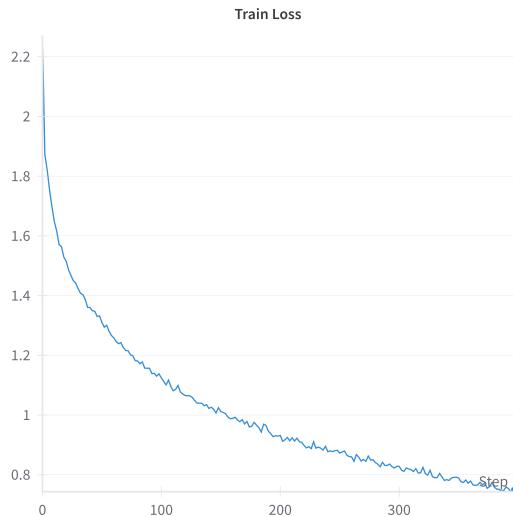


图 7: Train Loss

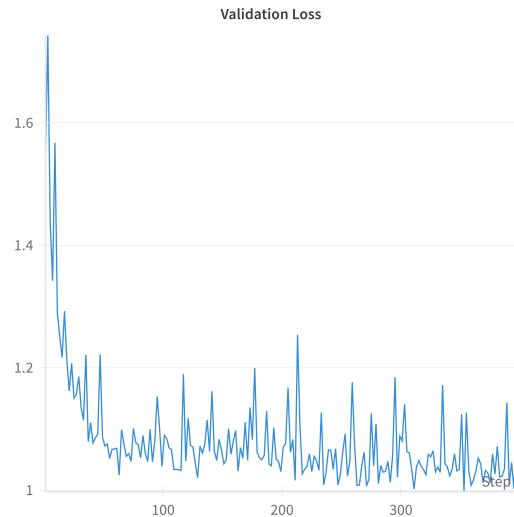


图 8: Validation Loss

图 9: GCN 的 loss 曲线

wandb 的输出：

```

1 wandb: Run summary:
2 wandb:     Test Loss 0.94064
3 wandb:     Test R2 0.41477
4 wandb:     Train Loss 0.75113
5 wandb: Validation Loss 1.09325

```

即最后的 R^2 得分为 0.41477，Test Loss 为 0.94064，说明 GCN 对偶极矩的预测具有一定的效果，但仍有较大的改进空间。

2.1.2 GAT

对gat.py的修改与gcn.py类似，都是为了让其向 wandb 传输训练数据和计算 R^2 得分，这里不再给出。

epoch同样设置为 200，wandb 作图，loss 曲线如图12所示。

wandb 的输出：

```

1 wandb: Run summary:
2 wandb:     Test Loss 1.1343
3 wandb:     Test R2 0.29428

```

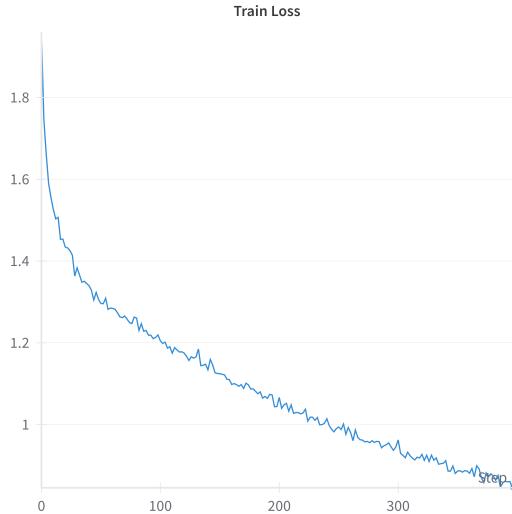


图 10: Train Loss

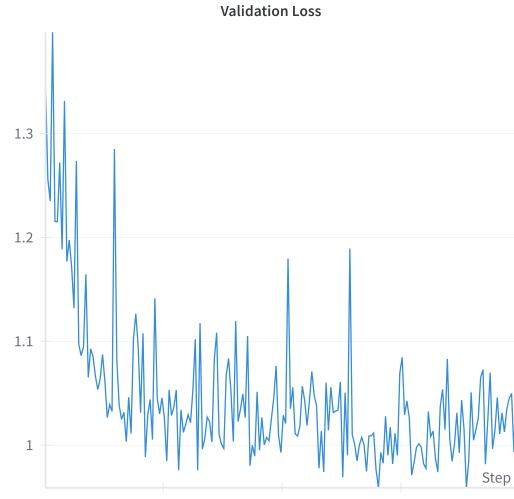


图 11: Validation Loss

图 12: GAT 的 loss 曲线

```
4 wandb:      Train Loss 0.84548
5 wandb: Validation Loss 1.06535
```

即最后的 R^2 得分为 0.29428, Test Loss 为 1.1343, 表明 GAT 在捕捉分子偶极矩特性上存在明显不足。

2.1.3 GraphSAGE

epoch 设置为 200, wandb 作图, loss 曲线如图15所示。

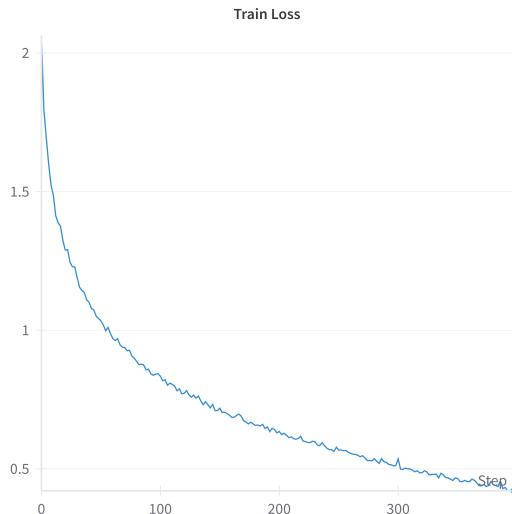


图 13: Train Loss

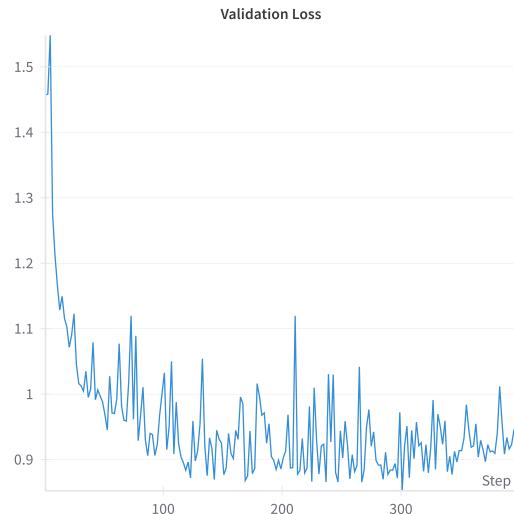


图 14: Validation Loss

图 15: GraphSAGE 的 loss 曲线

wandb 的输出:

```
1 wandb: Run summary:  
2 wandb:      Test Loss 0.91076  
3 wandb:      Test R2 0.43336  
4 wandb:      Train Loss 0.42136  
5 wandb: Validation Loss 0.95322
```

即最后的 R^2 得分为 0.43336, Test Loss 为 0.91076, 表明 GraphSAGE 可以捕捉分子偶极矩的部分特性, 但预测精度尚未达到较高水平。

2.2 Task B

2.2.1 代码实现

实现的 GIN 包含三层图卷积 (GINConv), 每层通过一个两层的 MLP 对节点特征进行更新, 并结合批归一化提高训练稳定性, 卷积后的特征通过两层全连接网络进行进一步处理, 并用 Dropout 减少过拟合, 如代码5所示。

代码 5: GIN 实现

```
1 class GIN(torch.nn.Module):  
2     def __init__(self, hidden_dim=128, dropout=0.3):  
3         super(GIN, self).__init__()  
4         nn1 = torch.nn.Sequential(  
5             torch.nn.Linear(dataset.num_features + 3, hidden_dim),  
6             torch.nn.ReLU(),  
7             torch.nn.Linear(hidden_dim, hidden_dim)  
8         )  
9         self.conv1 = GINConv(nn1)  
10        self.bn1 = BatchNorm(hidden_dim)  
11        nn2 = torch.nn.Sequential(  
12            torch.nn.Linear(hidden_dim, hidden_dim),  
13            torch.nn.ReLU(),  
14            torch.nn.Linear(hidden_dim, hidden_dim)  
15        )  
16        self.conv2 = GINConv(nn2)  
17        self.bn2 = BatchNorm(hidden_dim)  
18        nn3 = torch.nn.Sequential(  
19            torch.nn.Linear(hidden_dim, hidden_dim),  
20            torch.nn.ReLU(),  
21            torch.nn.Linear(hidden_dim, hidden_dim)  
22        )  
23        self.conv3 = GINConv(nn3)  
24        self.bn3 = BatchNorm(hidden_dim)  
25        self.fc1 = torch.nn.Linear(hidden_dim, hidden_dim)  
26        self.fc2 = torch.nn.Linear(hidden_dim, 1)  
27        self.dropout = torch.nn.Dropout(dropout)
```

2.2.2 训练

第一次训练时每个 epoch 前没有做 shuffle 处理，效果不是很理想；询问大模型并在每个 epoch 前使用shuffle_dataset打乱数据集，显著提升了训练效果，如代码6所示。

代码 6: 每个 epoch 前随机打乱训练集

```
1 def shuffle_dataset(dataset):
2     indices = torch.randperm( len(dataset) )
3     return Subset(dataset, indices)
```

损失曲线如图18所示。

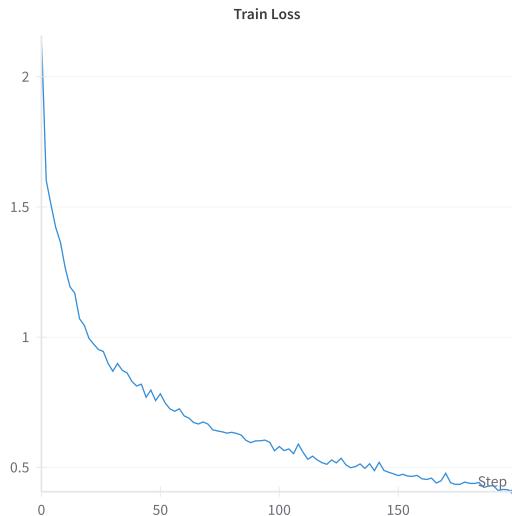


图 16: Train Loss

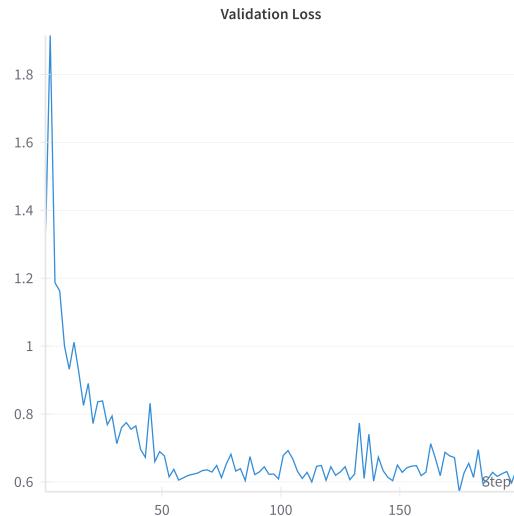


图 17: Validation Loss

图 18: GIN 的 loss 曲线

最终 epoch 选用 100, wandb 的输出如下：

```
1 wandb: Run summary:
2 wandb:      Test Loss 0.58857
3 wandb:      Test R2 0.63381
4 wandb:      Train Loss 0.40659
5 wandb: Validation Loss 0.63561
6 wandb: Validation R2 0.65993
```

可见最后实现的 GIN 的 R^2 得分为 0.63381, Test Loss 为 0.58857。