人体骨骼生成实验报告

本赛题旨在利用点云学习方法，探索一种自动预测骨骼节点和蒙皮权重的深度学习方案。根据骨骼与蒙皮特性，结合一般性深度学习网络可以得到如图 1 所示的模型主要训练架构。为提高骨骼与蒙皮预测的精度与鲁棒性，本文在主办方提供的 PCT-based baseline 基础上进行了多个方面的建模优化，下文将对做出的优化改进以及相关重要尝试进行详细说明。

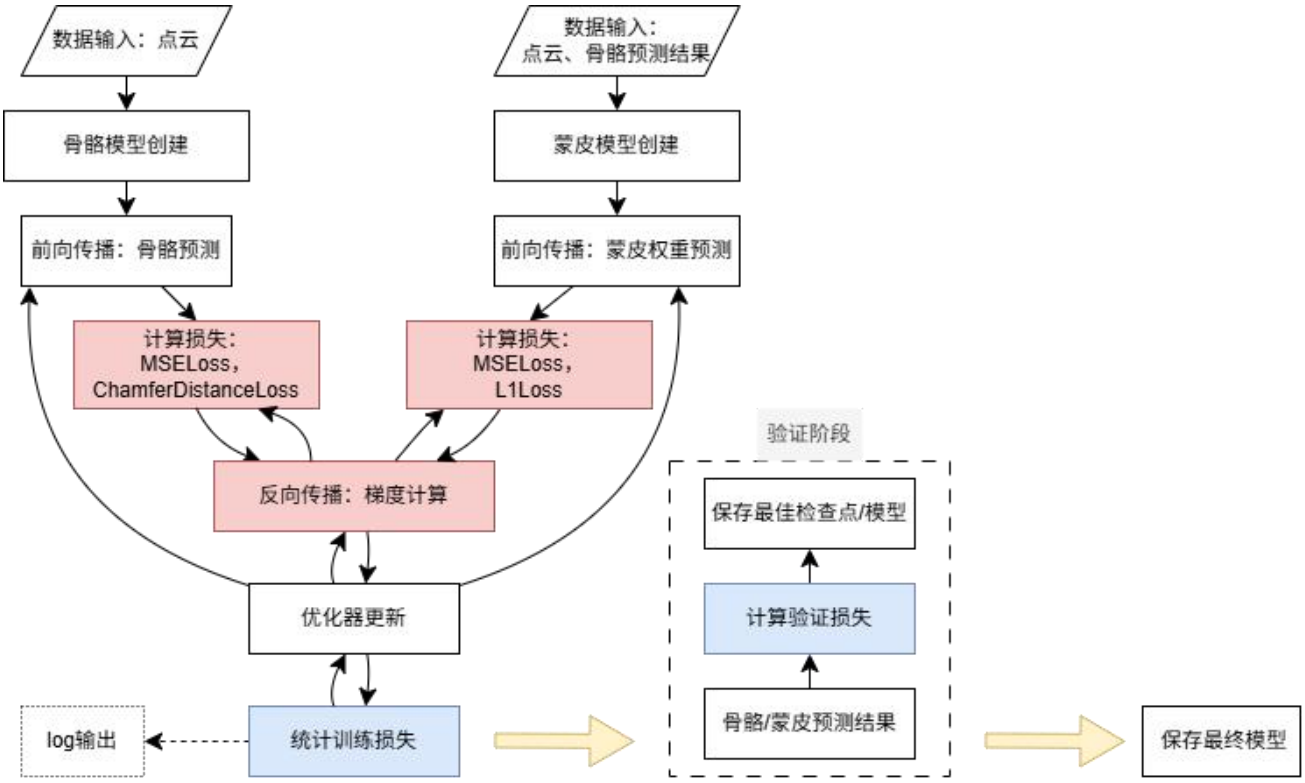


图 1 模型训练流程

1 网络架构优化

1.1 基于 PTCNN 的注意力机制模型

基于赛题提供的 baseline，我们将神经网络架构改为基于 PTCNN（Point Cloud Transformer Convolutional Neural Network）的预测模型。该模型通过引入多层特征提取、局部与全局信息融合以及多头自注意力（Multi-Head Attention）机制，有效地处理复杂的点云数据，提升了骨架关节位置的预测精度（详见附录 1）。

有上述设计的 PTCNNSkeletonModel_Advanced（PTCNNSkinModel_Advanced）模型相较于赛题提供的 baseline 中 SimpleSkeletonModel(SimpleSkeletonModel)，表现非常出色（如表 1 所示）。同时，我们发现在引入自注意力机制后，模型的收敛速度显著增强（如附录 4 所示）。

骨骼与蒙皮的模型均基于该模型（如图 2 所示），在特征提取结构上非常相似。主要区别在于关键实现部分，具体地，骨骼关节位置预测是通过 Attention 机制生成关节 3D 坐标，而蒙皮权重则通过计算定点特征和关节特征的相似度，并结合 softmax 操作进行计算。下面将进行具体介绍。

baseline	PTCNN（未引入 Attention）	PTCNN+Attention
j2j_loss：0.139 skin_l1loss：0.0836 vertex_loss：0.3243 score：21.5253	j2j_loss：0.0549 skin_l1loss：0.0284 vertex_loss：0.1236 score：54.0297	j2j_loss：0.0404 skin_l1loss：0.0165 vertex_loss：0.0945 score：68.1432

表 1 网络架构调整带来的评分跃升

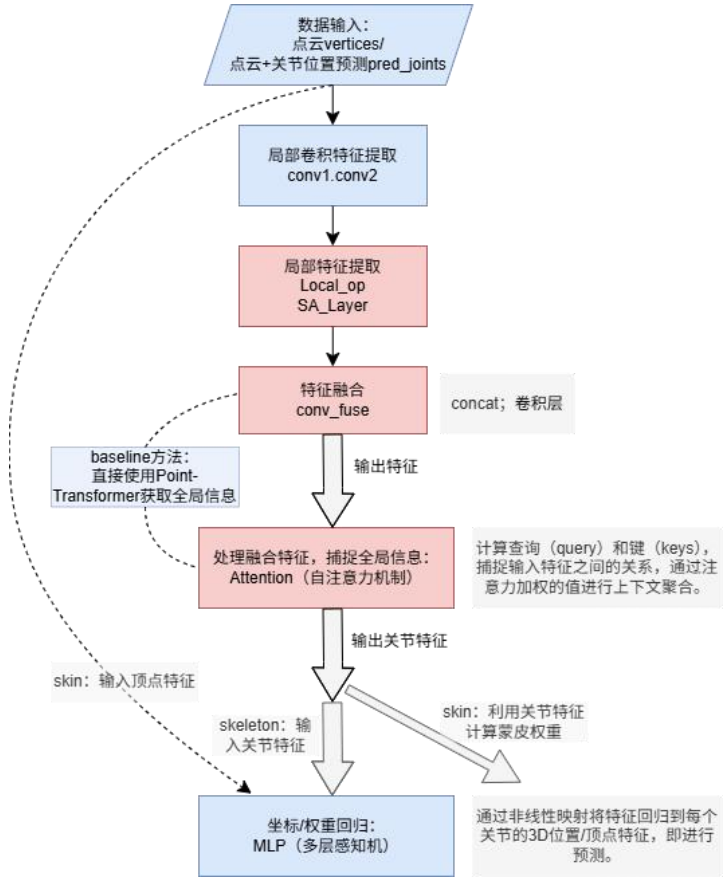


图 2 PTCNN 具体实现

1.1.1 多层特征提取

PTCNN 的输入为点云数据（3D 顶点坐标），该网络首先通过两个一维卷积层（conv1 和 conv2）对输入点云数据进行多层特征提取。具体来说，第一个卷积层将输入的 3 维坐标转化为 64 维特征向量，第二个卷积层进一步将这些 64 维特征向量转换为新的 64 维特征。这一层的作用是捕捉局部细节信息，并为后续的特征处理提供基础。每个卷积层后接批归一化层，旨

在提高训练稳定性并加速模型收敛。这一层级化的特征提取结构类似 CNN 中的池化层，能将网络从局部邻域细节（如手指关节）逐步聚合到全局形状轮廓（如身体躯干），这对于定位分布在不同尺度上的关节点是有利的，也能在更少的代表性点上运行计算成本更高的自注意力模块，可以更有效地学习高级语义特征。

在所有局部特征和全局特征提取后，PTCNN 通过 conv_fuse 模块将来自自注意力层和 Point-Transformer 模块的信息进行融合。这一操作通过卷积层和 LeakyReLU 激活函数的组合，不仅提高了模型的非线性表达能力，还使得融合后的特征能够更好地表达点云的复杂结构。

1.1.2 多头自注意力机制

在多层特征提取后，PTCNN 通过局部操作模块（Local_op）进一步提取局部特征，并将这些特征进行聚合；然后使用自注意力层（SA_Layer）使得每个点能关注到其他点的信息，强化点云中远距离点之间的关联。具体而言，通过多头注意力机制，网络能够从局部信息中提取全局语义信息，有利于加强复杂关系建模。

多头自注意力机制的核心在于将信息通过多个子空间进行并行计算，从而获得更多的上下文信息。具体地，在 Attention 机制中指定 num_heads 个头（详见附录 1），每个头在计算时都处理不同的表示子空间，最终，这些头的输出被拼接并通过线性变换组合成最终的全局特征表示。

1.1.3 全局感知与 Point-Transformer 模块

为了进一步提升模型的全局感知能力，我们在网络的深层加入基于 Point-Transformer 模型的 Point_Transformer_Last 模块，通过深层次的转换操作增强点云特征的全局理解。

1.2 学习率调度机制

学习率过大容易发生梯度爆炸，会增大模型收敛难度，过小则也可能使得收敛速度过慢。为了防止模型收敛到局部最优点或鞍点，设定如下学习率调度机制：在预热期后，使学习率线性衰减，最终降低到指定的最小值。具体地，在训练循环中，每个 epoch 开始时，都调用 adjust_learning_rate（详见附录 2）来更新优化器的学习率，帮助我们提高模型在不同训练阶段的收敛速度和稳定性。

1.3 端到端（End-to-End）模型的尝试

在骨骼和蒙皮预测任务中，传统的做法是将这两个任务模块化，并分别进行训练，以便分别优化蒙皮权重和骨骼位置。然而，考虑到这两个任务在实际应用中的紧密关系，优化两者的联合表现有可能进一步提升整体效果，我们尝试进行“端到端训练”的重构：模型的输入是点云（或三维网格的顶点信息），并同时预测骨骼关节的位置与蒙皮权重。具体而言，模型使用线性混合蒙皮（LBS）和基于变换的姿态估计来联合学习，并在训练过程中参照指标所规定的 J2J 损失和 L1 损失构造优化函数，联合优化骨骼预测和蒙皮权重（详见附录 3）。

端到端的方式让骨骼的预测与蒙皮权重在同一框架下进行调整，预期提高任务间的协同作用，允许模型自动学习骨骼和蒙皮之间的最佳映射而非独立调节两个模块任务的损失函数等。然而在实验中这一重构并未取得预期效果，损失呈现不良趋势（如图 3 所示）。

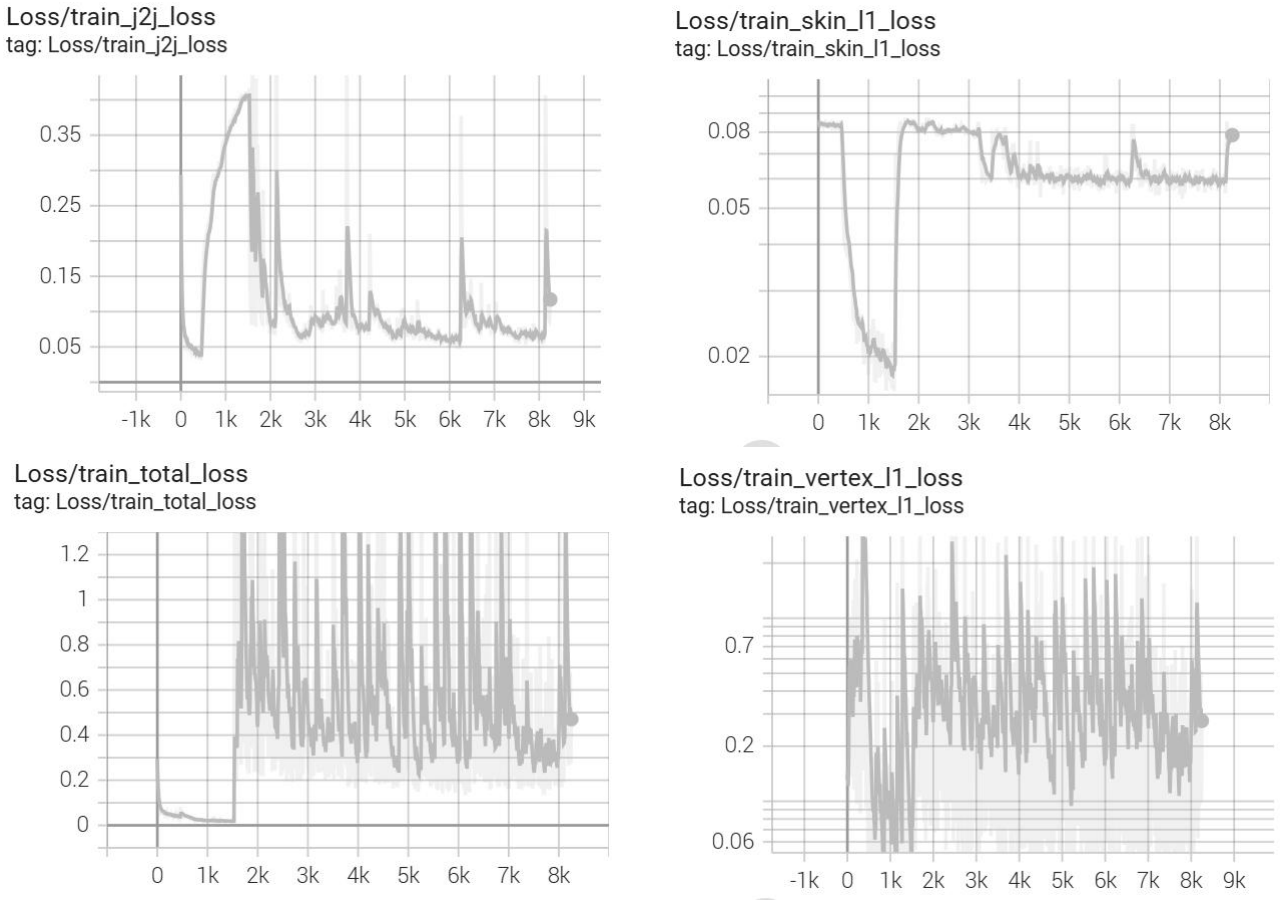


图 3 端到端模型训练效果

分析提出可能原因如下：

1) 实验中，数据集的规模较小，未能为模型提供足够的多样性，导致模型未能学到有效的特征，无法有效地联合优化骨骼和蒙皮任务。
2) 骨骼和蒙皮预测任务之间的关系较为复杂，并非简单的线性映射，尽管我们为模型设计了多任务损失函数，但训练过程中梯度更新并未有效协调这两个任务，优化方向容易发生冲突，最终导致整体性能不如预期。
3) 我们的方案虽尝试根据训练的 epoch 数调整不同任务的权重（例如，前期关注骨骼预测，后期加入蒙皮优化），但这种动态调整的策略可能没有充分考虑到任务间的优化难度差异，导致部分任务在训练过程中被过度优化，影响了模型的整体学习效果。

2 损失函数优化

在 baseline 框架中，骨骼预测采用点对点回归方式，以最小化预测关键点位置与标签之间

的 Euclid 误差，采用均方误差损失（MSELoss）作为损失函数。显然这是一个极其粗略的版本，在很多极端情况下并不能良好处理。

2.1 Chamfer Distance Loss

在骨骼位置预测中，仅使用点对点损失忽略了整体结构的几何分布一致性。为弥补这一不足，我们参照赛题评分标准，引入了 Chamfer 距离，预测关键点集与真实关键点集之间计算双向最邻近距离均值，从而引导网络学习整体结构形状的匹配而非仅关注对应点（详见附录 5）。可惜的是，在引入此损失函数后，相应的损失并没有取得优化，分析可能原因与不同损失函数的权重设置有关。

2.2 人体结构先验约束

在常见的 3D 人体骨骼结构中，有一些基本的拓扑结构和对称性要求（例如图 4），预期借助此先验知识，构造出更合理的损失函数来提升模型的结构感知和泛化能力。我们在 SymmetricJointLoss 类中分别实现了两种形式的对称性要求。

第一种是位置（Position）对称性，要求对应关节预测结果满足空间坐标镜像关系，较为严格，仅适用于标准 T-pose 等完全对称的姿态。

第二种是结构（Structure）对称性，要求关节相对跟节点方向的一致性，如图 4 所示，选取 0 号 Pelvis（骨盆）作为人体根节点，适用于非对称的训练样本（详见附录 6）。可以看出，人体对称性的要求较高，在对有强非对称性动作（例如左臂上举，右臂下放）的网格而言，反而有可能适得其反，因此在实验中我们倾向于对该项损失设定较低的权重比例以控制正则影响。

3 计算模块优化

3.1 可微分 LBS

传统线性混合蒙皮（LBS）方法的本质是通过关节变换矩阵对顶点进行加权平均，虽然计算简便且广泛应用，但其过程中的蒙皮权重和顶点位置无法直接参与反向传播，因此无法对蒙皮过程中的权重进行优化。引入可微分 LBS 层预期将会增强深度学习框架的可训练性，提高蒙皮预测的精度。具体地，通过将蒙皮权重 W 和关节变换矩阵 pose 作为可训练参数，结合求解加权变换后的顶点位置，从而实现对蒙皮权重的学习；引入了线性变换和矩阵乘法，这些操作是完全可微的，因此可以通过自动微分计算梯度并优化模型参数（详见附录 3）。

4 实验辅助优化

4.1 MPI 分布训练

为了提高训练效率，我们引入了 MPI（Message Passing Interface）并行训练优化，实现多

GPU 训练。使用 jt.rank 和 jt.world_size 处理分布式进程，将数据自动分配到多个 GPU 进行处理，同时在不同进程之间平均训练和验证过程的损失与指标。

ID	Joint Name	Symmetric With (ID)	Note
0	hips（髋部）	-1	中心根节点
1	spine（脊柱）	-1	中轴
2	chest（胸部）	-1	中轴
3	upper_chest（上胸）	-1	中轴
4	neck（脖子）	-1	中轴
5	head（头部）	-1	中轴
6	l_shoulder（左肩）	10	左右对称
7	l_upper_arm（左上臂）	11	左右对称
8	l_lower_arm（左前臂）	12	左右对称
9	l_hand（左手）	13	左右对称
10	r_shoulder（右肩）	6	左右对称
11	r_upper_arm（右上臂）	7	左右对称
12	r_lower_arm（右前臂）	8	左右对称
13	r_hand（右手）	9	左右对称
14	l_upper_leg（左大腿）	18	左右对称
15	l_lower_leg（左小腿）	19	左右对称
16	l_foot（左脚）	20	左右对称
17	l_toe_base（左脚趾）	21	左右对称
18	r_upper_leg（右大腿）	14	左右对称
19	r_lower_leg（右小腿）	15	左右对称
20	r_foot（右脚）	16	左右对称
21	r_toe_base（右脚趾）	17	左右对称

图 4 人体骨骼对称性

4.2 TensorBoard

为实时监控模型的训练状态与性能表现，引入 TensorBoard 作为训练过程的可视化工具。观察记录训练集与验证集在各轮次的总损失、主任务损失（如 Skin-L1）、辅助对称性损失等指标，便于观察收敛趋势与潜在过拟合。

5 实验效果与团队贡献

5.1 实验效果

综合得分最好的一次提交结果如图 5 所示。

2025062002004 37607616	predict 9....	陆泓臻	2025-06-20 02:00:43	完成	j2j_loss：0.0404 skin_l1loss：0.0165 vertex_loss：0.0945 score：68.1432	
16		记得开源代码队	清华大学	0	68.1432	68.1432

图 5 截止评测结果

5.2 团队贡献

王子轩	1. 负责大作业环境搭建和整体框架，完成项目 MPI 多卡训练的脚本撰写 2. 提出整体优化思路，控制进度与分工 3. 搭建和跑通最高分版本的模型代码 4. 搭建端到端训练实验代码和进行训练
陆泓臻	1. 跑通 baseline、PointTransformer2、PointTransformerv3[1]、PTCNN 的模型代码 2. 搭建 TensorBoard 记录训练过程 3. 调整模型参数进行多次训练
刘玥瑶	1. 梳理骨干网络，对各流程影响因素等进行理论分析 2. 实现损失函数的优化与评测指标的对齐 3. 绘制算法流程图，撰写实验报告

[1] 直接改用 PT2 和 PT3 的训练效果并未在 baseline 基础上有显著提升，故在本报告中未进行介绍。

附录

附录 1：注意力机制

```
class Attention(nn.Module):
    def __init__(self, embed_dim, num_heads, dropout=0.0):
        super().__init__()
        self.embed_dim = embed_dim
        self.num_heads = num_heads
        self.head_dim = embed_dim // num_heads
        assert self.head_dim * num_heads == self.embed_dim, "embed_dim 必须能被 num_heads 整除
"
        self.q_proj = nn.Linear(embed_dim, embed_dim)
        self.k_proj = nn.Linear(embed_dim, embed_dim)
        self.v_proj = nn.Linear(embed_dim, embed_dim)
        self.out_proj = nn.Linear(embed_dim, embed_dim)
        self.dropout = nn.Dropout(dropout)
        self.softmax = nn.Softmax(dim=-1)
    def execute(self, query, key, value):
        B, N_q, C = query.shape; B, N_k, C_k = key.shape
        q = self.q_proj(query).reshape(B, N_q, self.num_heads, self.head_dim).permute(0, 2, 1, 3)
        k = self.k_proj(key).reshape(B, N_k, self.num_heads, self.head_dim).permute(0, 2, 3, 1)
        v = self.v_proj(value).reshape(B, N_k, self.num_heads, self.head_dim).permute(0, 2, 1, 3)
        attn_scores = jt.matmul(q, k) * (self.head_dim ** -0.5)
        attn_probs = self.softmax(attn_scores)
        attn_probs = self.dropout(attn_probs)
        context = jt.matmul(attn_probs, v).permute(0, 2, 1, 3).reshape(B, N_q, C)
        return self.out_proj(context)
```

附录 2：学习率调度

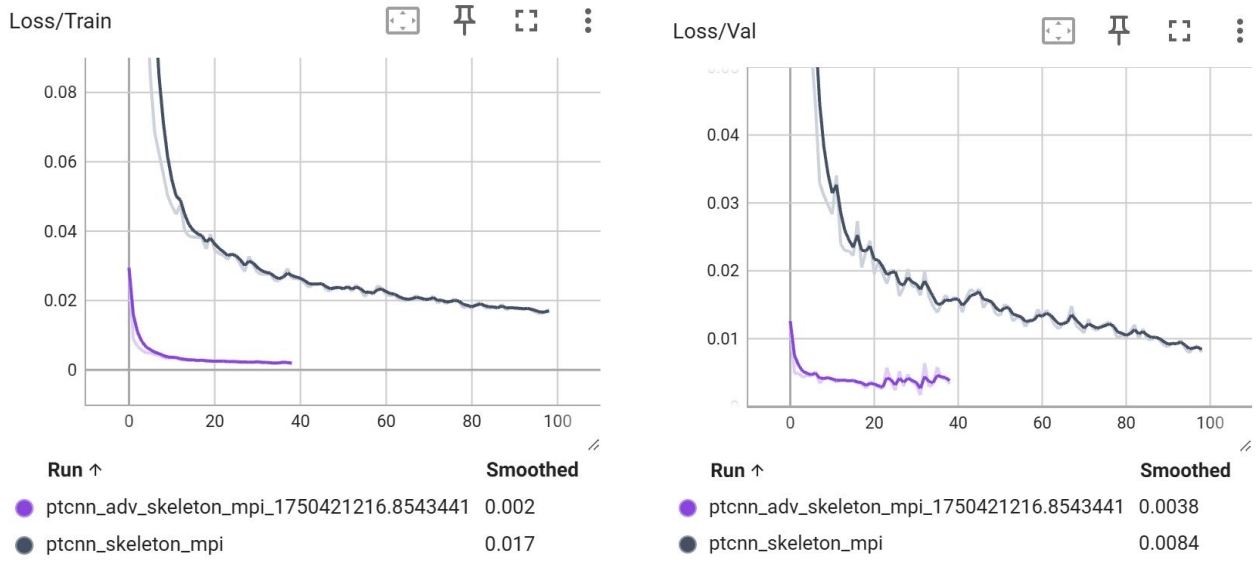
```
def adjust_learning_rate(optimizer, epoch, args):
    """设置优化器的学习率，根据当前的 epoch 进行调整"""
    if epoch < args.lr_warmup_epochs:
        lr = args.learning_rate  # 如果在预热阶段，使用初始学习率
    else:
        # 线性衰减学习率
```

```
        decay_progress = (epoch-args.lr_warmup_epochs) / (args.epochs- args.lr_warmup_epochs)
        lr_range = args.learning_rate - args.lr_end
        lr = args.learning_rate - lr_range * decay_progress    # 计算衰减后的学习率
    optimizer.lr = lr    # 更新优化器的学习率
    return lr
```

附录 3：可微分 LBS

```
class DifferentiableLBS(nn.Module):
    def __init__(self):
        super(DifferentiableLBS, self).__init__()
    def execute(self, V, W, J_inv_mats, pose):
        batch_size, num_points, _ = V.shape
        T = jt.matmul(pose, J_inv_mats)
        V_homo = concat([V, jt.ones((batch_size, num_points, 1))], dim=2)
        W_expanded = W.unsqueeze(-1).unsqueeze(-1)
        T_expanded = T.unsqueeze(1)
        G = (W_expanded * T_expanded).sum(dim=2)
        V_homo_reshaped = V_homo.unsqueeze(-1)
        V_posed_homo = jt.matmul(G, V_homo_reshaped)
        V_posed = V_posed_homo.squeeze(-1)[:, :, :3]
        return V_posed
```

附录 4：PTCNN 模型收敛速度对比



附录 5：Chamfer Distance Loss

```
class ChamferDistanceLoss(nn.Module):
    def execute(self, pred_points: jt.Var, gt_points: jt.Var) -> jt.Var:
        """
        Args:
            pred_points (jt.Var): 预测的点集，形状为 (B, N*3)，其中 B 是批大小，N 是点数。
            gt_points (jt.Var): 真实的点集，形状为 (B, M*3)，其中 M 可以不等于 N。
        Returns:
            jt.Var: 一个标量，表示该批次的平均 Chamfer Distance 损失。
        """
        if pred_points.ndim == 2:
            pred_points = pred_points.reshape(pred_points.shape[0], -1, 3)
        if gt_points.ndim == 2:
            gt_points = gt_points.reshape(gt_points.shape[0], -1, 3)

        pred_expanded = pred_points.unsqueeze(2)
        gt_expanded = gt_points.unsqueeze(1)

        dist_matrix_sq = ((pred_expanded - gt_expanded)**2).sum(dim=-1)
        dist_matrix = jt.sqrt(dist_matrix_sq + 1e-12) # 避免在求梯度时出现 sqrt(0) 导致 nan

        # 1. 计算从 pred_points 到 gt_points 的最短距离
        #    对于 pred_points 中的每个点，找到 gt_points 中最近的点的距离
        dist_pred_to_gt = dist_matrix.min(dim=2) # 形状: (B, N)

        # 2. 计算从 gt_points 到 pred_points 的最短距离
        #    对于 gt_points 中的每个点，找到 pred_points 中最近的点的距离
        dist_gt_to_pred = dist_matrix.min(dim=1) # 形状: (B, M)

        # 计算两边距离的平均值，得到 CD loss
        cd_loss = dist_pred_to_gt.mean(dim=1) + dist_gt_to_pred.mean(dim=1) # 形状: (B,)
        return cd_loss.mean()
```

附录 6：Symmetric Loss

```
class SymmetricJointLoss(nn.Module):
```

```
def __init__(self, mode='none', joint_pairs=None, root_joint_id=0):
    super().__init__()
    assert mode in ['none', 'position', 'structure'], \
        f"未知对称模式: {mode}"
    self.mode = mode
    self.symmetric_joint_pairs = joint_pairs or [
        (6, 10),   # l_shoulder - r_shoulder
        (7, 11),   # l_upper_arm - r_upper_arm
        (8, 12),   # l_lower_arm - r_lower_arm
        (9, 13),   # l_hand - r_hand
        (14, 18), # l_upper_leg - r_upper_leg
        (15, 19), # l_lower_leg - r_lower_leg
        (16, 20), # l_foot - r_foot
        (17, 21)   # l_toe_base - r_toe_base
    ]
    self.root_id = root_joint_id

def execute(self, joints: jt.Var) -> jt.Var:
    """
    joints: [B, J, 3]  预测关节坐标
    """
    if joints.ndim == 2:
        joints = joints.reshape(joints.shape[0], -1, 3)
    if self.mode == 'none':
        return jt.zeros(1)
    B = joints.shape[0]
    loss = 0.0
    for left, right in self.symmetric_joint_pairs:
        left_joint = joints[:, left, :] # [B, 3]
        right_joint = joints[:, right, :] # [B, 3]
        root_joint = joints[:, self.root_id, :].unsqueeze(1)   # [B, 1, 3]

        if self.mode == 'position':
            #  复制右侧关节坐标并再 X 轴上镜像（假设人体朝着 Z 正方向）
            mirrored_right = right_joint.clone()
```

```
                mirrored_right[:, 0] = -mirrored_right[:, 0]
                # 计算欧式距离
                loss += jt.norm(left_joint - mirrored_right, dim=1).mean()

            elif self.mode == 'structure':
                left_vector = left_joint - root_joint.squeeze(1)  # [B, 3]
                right_vector = right_joint - root_joint.squeeze(1)  # [B, 3]
                mirrored_right_vector = right_vector.clone()
                mirrored_right_vector[:, 0] = -mirrored_right_vector[:, 0]
                # 计算欧式距离
                loss += jt.norm(left_vector - mirrored_right_vector, dim=1).mean()

        return loss / len(self.symmetric_joint_pairs)

class SymmetricSkinLoss(nn.Module):
    '''
    对称蒙皮权重损失：用于约束对称点的皮肤权重一致。
    需要传入 symmetry_index 映射。
    '''
    def __init__(self, symmetry_index):
        super().__init__()
        self.symmetry_index = symmetry_index  # [N] numpy or jt.Var
    def execute(self, skin: jt.Var) -> jt.Var:
        """
        skin: [B, N, J]
        self.symmetry_index: [N], 每个点的对称点索引
        """
        mirrored_skin = skin[:, self.symmetry_index, :]  # [B, N, J]
        return jt.abs(skin - mirrored_skin).mean()
```