

AllTracker: Efficient Dense Point Tracking at High Resolution —— 将点跟踪重构为高分辨率下的长距离光流问题

论文网址: <https://arxiv.org/abs/2506.07310>

github: <https://github.com/aharley/alltracker>

1.论文内容概述

1.1 研究背景与动机

估计视频中的长距离点轨迹是计算机视觉理解动态场景的基础构建模块。通过光流法或稀疏点跟踪算法，现已能分别获取密集的瞬时运动或长时程的稀疏轨迹。然而，这些方法并非完美的通用解决方案，因其分别受限于光流法的累积漂移误差，或点跟踪法的数据稀疏性。对于高分辨率视频的全像素分析，目前尚不存在能同时提供全像素密集对应关系、长时程鲁棒性且计算高效的“野外”跟踪方案。实现这种可靠的高分辨率密集跟踪系统，将把点跟踪任务重构为长距离光流问题，为视频理解开辟新的研究途径。

1.2 相关工作

1.2.1 从光流到长时程轨迹的挑战

估计任意点的长距离轨迹是计算机视觉的一个核心挑战，这一目标至少可以追溯到 2006 年 Sand & Teller 提出的 Particle Video。光流（即像素的瞬时速度）显然是实现这一目标的工具。然而，直接将光流用于长时程跟踪存在天然缺陷：即通过“链接”（Chaining）瞬时流场来形成多帧轨迹。这种方法存在严重的局限性：不完美的光流会导致漂移误差迅速累积，且在遇到遮挡时必须强制停止采样，无法维持轨迹的连续性。

[40] Sand & Teller (CVPR 2006): 早期经典工作，试图通过粒子视频来估计长距离运动轨迹，但受限于当时的计算能力和光流精度。

- [16] Gibson (1950) / [38] Niehorster (2021): 奠定了光流 (Optic Flow) 作为视觉感知基础的概念。
- [47] RAFT (ECCV 2020) / [54] SEA-RAFT (ECCV 2024): 现代光流的基石。AllTracker 借鉴了它们在低分辨率网格上进行迭代推理并在最后上采样的设计理念，以实现高效的空间信息传播。

1.2.2 稀疏点跟踪

鉴于光流法的局限性，近年来出现了一系列定制化的点跟踪器（Bespoke Point Trackers），如 PIPs 和 TAPIR。这些工作明确指出光流法缺乏时序上下文是其致命弱点，因此专注于学习多帧时间先验，从而成功实现了穿透遮挡（Tracking through occlusions）的跟踪能力。随着 TAP-Vid 基准的提出，该领域迅速发展。然而，这些方法为了增加时间感知能力，被迫牺牲了空间感知能力，只能为稀疏的点集提供轨迹，无法做到全像素覆盖。

[19] PIPs (ECCV 2022): 引入了每像素的时间模块（Temporal Module）来学习运动先验，这是 AllTracker 借用的核心组件之一。

[11] TAP-Vid (NeurIPS 2022): 提出了 "Tracking Any Point" 任务及基准，推动了后续如 [12] TAPIR, [24] CoTracker3 等工作的发展。

[24] CoTracker3 (2024): 论文指出的最强竞品（SOTA）。它引入了“虚拟点”机制来压缩视频运动信息。虽然精度极高，但它要求用户预先选择稀疏的查询点，且性能受限于查询点的分布方式。

1.2.3. 密集跟踪

为了填补“稀疏跟踪”与“密集光流”之间的空白，近期出现了尝试密集跟踪的工作，如 DTF 和 DELTA。这些并发工作与本研究类似，也试图通过迭代估计流场来关联参考帧与后续帧。

然而，它们采用了基于transformer的特殊架构，仅通过稀疏的“锚点”或“质心”Token 来近似全局空间消息传递。这导致它们在处理高分辨率输入（如本研究的 768×1024 ）时面临严重的显存溢出（OOM）问题，且准确率未能达到稀疏跟踪器的水平

[50] DTF (ACCV 2024) / [37] DELTA (ICLR 2025): 两项并发工作。它们发表的时间非常扎堆，都集中在 2024–2025 年。DELTA 尽管使用了低分辨率推理（类似 RAFT），但在高分辨率上仍难以扩展。DTF 也试图通过迭代估计流场，把参考帧和其他帧联系起来，实现密集跟踪，但是他们的性能无法与最新的点跟踪器竞争。

[27] Le Moing et al. (CVPR 2024): 尝试进行“后处理致密化”（Post-hoc densification），但这不如直接端到端生成密集轨迹高效。

1.2.4 联合训练

现有的光流和点跟踪方法高度依赖数据驱动。光流领域依赖 FlyingChairs, TartanAir 等合成数据集，而点跟踪领域则依赖 Kubric, PointOdyssey。以往的点跟踪方法从未利用光流数据，可能是因为模型不支持两帧推理或密集输出。AllTracker 通过统一架构，首次实现了在光流数据集和点跟踪数据集上的联合训练，这被证明是实现 SOTA 性能的关键。

[17] Kubric / [59] PointOdyssey / [14] FlyingChairs：本研究混合使用的核心数据集，涵盖了从短距离两帧运动到长距离多帧轨迹的各种数据分布。引用一个数据集，本质上就是引用“首次提出或发布该数据集”的那篇论文。所以也就是这三篇。

1.3. 创新点

- 提出了一种全新的视角，将点跟踪任务重构为长距离、多帧的密集光流估计问题。
- 设计了一种混合时空架构 (Hybrid Spatial–Temporal Architecture)，并在推断时采用滑动窗口 (Sliding Window) 策略。
- 首个能同时实现高分辨率 (768×1024)、全像素密集 (All-pixel Dense) 且显存高效的长时程跟踪系统。
- 验证了光流数据集与点跟踪数据集联合训练 (Joint Training) 的有效性。
- 视角重构：摒弃了传统光流逐帧累积 ($t \rightarrow t + 1$) 的模式，改为直接计算查询帧与视频中其他每一帧 ($Query \rightarrow All$) 之间的流场。

优势：这种“直连”方式从根本上消除了长距离跟踪中的累积漂移 (Drift)，同时保留了光流方法的密集特性。

- **混合架构：**巧妙结合了光流模型（如 SEA-RAFT）的2D 卷积与点跟踪模型（如 PIPs）的时间注意力。

2D 卷积 (Spatial)：在低分辨率网格上高效处理空间信息，捕捉纹理和边缘。

时间注意力 (Temporal)：使用像素对齐 (Pixel-aligned) 的 Transformer 模块在时间轴上传播信息，解决遮挡和长时程关联。

- **联合训练：**利用架构的通用性，该方法可以混合使用光流数据集（如 FlyingChairs）和点跟踪数据集（如 Kubric）。

联合训练的必要性：实验证明，仅使用单一数据源无法达到 SOTA 性能；混合训练弥补了光流数据缺乏长时程标注和点跟踪数据不够密集的短板。

论文指出，AllTracker 之所以能实现光流和点跟踪数据的联合训练，主要得益于以下三个核心设计：

1.AllTracker 被设计为输出全分辨率、全像素的对应图 (Dense Correspondence Map)。这意味着它的输出格式本质上就是光流图。当使用光流数据时，它的输出直接对应光流真值 (Flow GT)。当使用点跟踪数据时，它的输出包含了所有像素的轨迹，我们只需要从中采样 (Sample) 出那些有标注的点来计算 Loss 即可。

2.AllTracker 的时间模块采用了 Transformer (Pixel-aligned Attention)。其中 Transformer 对序列长度不敏感。当输入是光流数据 (2帧) 时，Transformer 依然可以正常工作 (此时时间注意力虽然冗余，但无需修改架构即可运行)。当输入是视频数据 (多帧) 时，Transformer 自然地处理长序列。这使得模型可以用同一套权重，在训练循环中无缝切换两种数据源。

3. 模型采用了一种灵活的监督策略：

- 对于密集光流数据：把它看作是“长度为2、所有像素都有标注”的特殊轨迹，直接计算全图 Loss。
- 对于稀疏点跟踪数据：使用双线性采样 (Bilinear Sampling)，只在有 Ground Truth 的位置提取模型的预测值并计算 Loss，忽略其他位置。

这种设计使得模型不需要知道当前输入的是什么数据集，只需要根据提供的标签算 Loss 就行。

1.4 method

Alltrackers 输入的是一个形状为 $T, H, W, 3$ 的视频，其中 T 表示帧数， H, W 表示每帧的空间分辨率。我们还给定一个索引 $t \in T$ ，指示哪个帧包含我们需要跟踪的像素。最终输出是一个形状为 $T, H, W, 4$ 的张量：前两个通道是光流图，表示查询帧中每个像素到其他所有帧中对应像素的偏移量；后两个通道分别估计可见性和置信度。

1.4.1 编码(encoding)

AllTracker 的第一步是将原始视频帧映射为适合密集对应关系计算的低分辨率特征表示。

模型接收视频序列输入 $\mathbf{I} \in \mathbb{R}^{T \times H \times W \times 3}$ ，其中 T 为帧数， H, W 为空间分辨率。为了处理任意长度的视频，模型采用滑动窗口 (Sliding Window) 策略，处理长度为 S 的子序列 (本研究中 $S = 16$)，步长为 $S/2$ 。

为了实现这一目标，作者采用了 ConvNeXt-Tiny 作为骨干网络，并进行了特定的结构修改以适应密集跟踪任务。

[31]* *A ConvNet for the 2020s* (2022, CVPR): 提出 ConvNeXt 架构。本研究使用其前三个阶段 (Blocks) 作为特征提取器，利用其优秀的纯卷积性能提取语义与纹理特征。

同时，为了防止特征图分辨率过低导致运动信息丢失，作者对 ConvNeXt 的第三个阶段进行了关键修改：

- **步长调整：** 将原本 stride=2 的卷积层改为 stride=1。
- **核参数调整：** 利用双线性插值 (bicubic interpolation) 将原本 2×2 的下采样卷积核调整为 3×3 的普通卷积核。
- **权重重缩放：** 对调整后的权重数值乘以 $4/9$ 进行重缩放，以适应卷积核面积的变化（从 4 个像素扩展到 9 个像素）。

ConvNeXt类位于nets/blocks.py文件中，从第327行开始定义：

```
1  class ConvNeXt(nn.Module):  
2      def __init__(  
3          self,  
4          block_setting: List[CNBlockConfig],  
5          stochastic_depth_prob: float = 0.0,  
6          layer_scale: float = 1e-6,  
7          num_classes: int = 1000,  
8          block: Optional[Callable[..., nn.Module]] = None,  
9          norm_layer: Optional[Callable[..., nn.Module]] = None,  
10         init_weights=True):
```

后续作者对convnext的调整如以下代码所示：

```

1  if self.init_weights:
2      from torchvision.models import convnext_tiny, ConvNeXt_Tiny_Weights
3      pretrained_dict = convnext_tiny(weights=ConvNeXt_Tiny_Weights.DEFAULT).state_dict()
4      # from torchvision.models import convnext_base, ConvNeXt_Base_Weights
5      # pretrained_dict = convnext_base(weights=ConvNeXt_Base_Weights.DEFAULT).state_dict()
6      model_dict = self.state_dict()
7      pretrained_dict = {k: v for k, v in pretrained_dict.items() if k in model_dict}
8
9      for k, v in pretrained_dict.items():
10         if k == 'features.4.1.weight': # 这是负责2x2下采样的层
11             # 转换为3x3滤波器 同时*4/9 进行缩放
12             pretrained_dict[k] = F.interpolate(v, (3, 3), mode='bicubic',
13 align_corners=True) * (4/9.0)
14
15     model_dict.update(pretrained_dict)
16     self.load_state_dict(model_dict, strict=False)

```

由于 AllTracker 的核心任务是计算查询帧（通常为窗口内的第 0 帧）到窗口内其他所有帧的光流（即 $Query \rightarrow All$ ），模型需要一种机制将查询帧的信息广播给每一个时间步。确保在后续的相关性计算中，每一个时间步 t 都能并行地访问到“始发站”的特征信息，从而构建 4D 相关性体。

步骤如下：

- ① 提取查询帧的特征图 \mathbf{F}_{query}
- ② 并将其沿时间轴 复制并平铺 (Tile) S 次 (在这个例子中, S 等于 16)
- ③ 构建出 $\mathbf{F}_{query_vol} \in \mathbb{R}^{S \times H/8 \times W/8 \times D}$

平铺操作的代码如下所示：

```

1  # fmap_anchor 是第一帧的特征图
2  fmap_anchor = fmaps[:, 0]
3
4  # 在 forward_window 方法中，对第一帧的图进行复制 S 次，以便后续帧对其进行对比
5  fmap1 = fmap1_single.unsqueeze(1).repeat(1, S, 1, 1, 1) # B, S, C, H, W

```

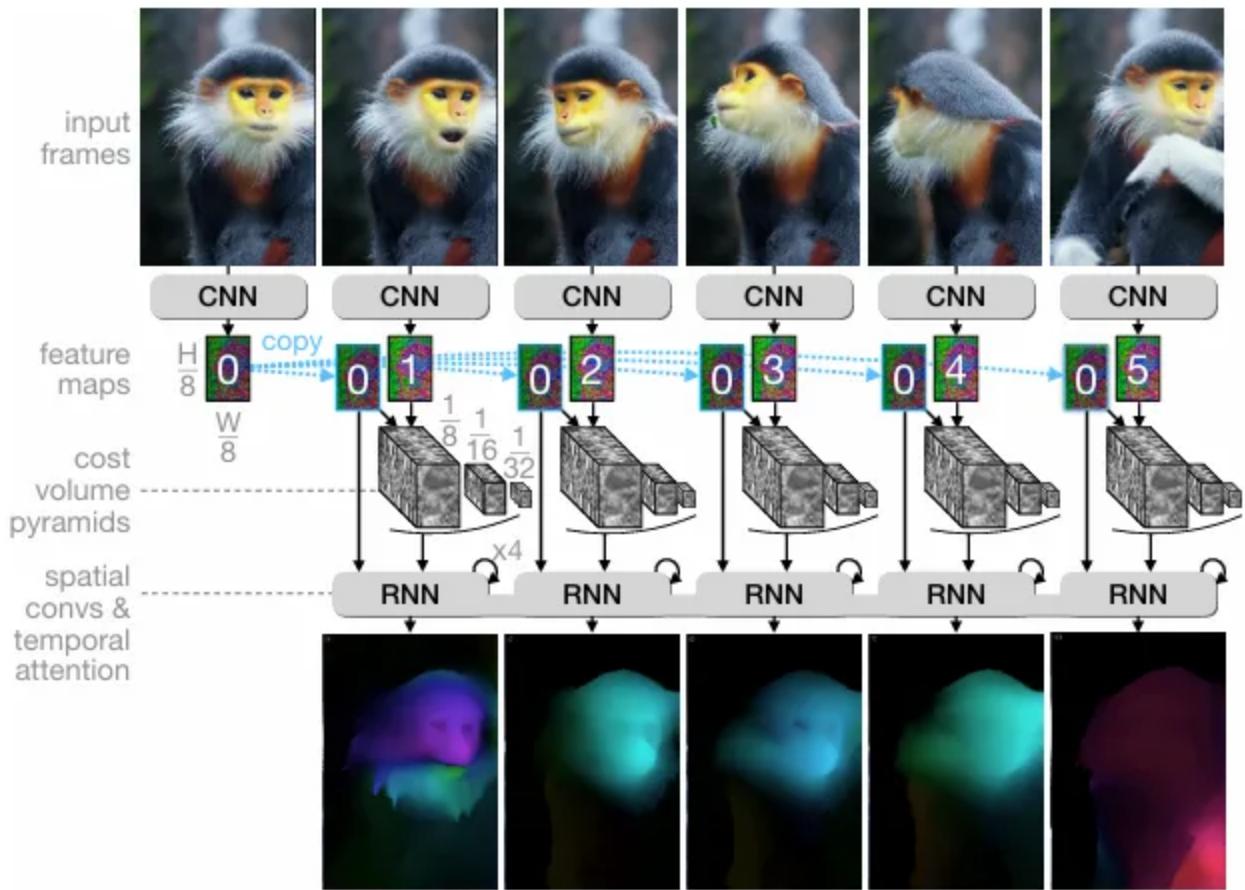


Figure 2. AllTracker architecture. First, we compute feature maps for all frames, and copy the zeroth (query) feature map to every timestep, and compute multi-scale cost volumes. Then, we iterate a recurrent module, which references the query feature map and cost volume pyramid at each timestep, and estimates a low-resolution correspondence field, using interleaved 2D convolutions and pixel-aligned temporal attentions. The output of the RNN is upsampled into high-resolution optical flow maps, which relate all pixels of the zeroth frame to every other frame.

这张图 (Figure 2) 展示了 AllTracker 的整体网络架构。它的核心逻辑是：输入一段视频序列，直接计算出第 0 帧（查询帧）到后续所有帧的密集光流。整个过程大致分为特征提取、相关性计算、循环迭代三个阶段。

一开始系统将一段视频帧 (Input Frames) 输入到一个共享权重的 CNN 编码器中。这个编码器会将每一帧图像压缩成低分辨率的特征图 (Feature Maps)，也就是图中带数字 0, 1, 2... 的这些小方块。这里的空间分辨率是原图的 $1/8$ 。

这里有一个非常关键的操作，也是本文的核心设计之一。注意看这个蓝色的箭头。因为该论文的任务是计算‘查询帧’到‘所有帧’的光流 ($Query \rightarrow All$)，所以模型将第 0 帧（即查询帧）的特征图提取出来，沿着时间轴复制 (Copy) 并平铺到了每一个时间步。这样，在后续的每一个时刻，模型都能直接拿到‘始发站’（第 0 帧）的信息来进行比对。

有了特征图之后，模型接着构建了多尺度的相关性金字塔。

接下来进入模型的RNN 迭代模块。模型不会一次性输出结果，而是通过这个 RNN 模块循环迭代 4 次。在每一次迭代中，RNN 会接收当前的特征、相关性信息，通过内部交替使用的 2D 空间卷积（处理纹理）和 Pixel-aligned 时间注意力（处理时序），不断修正对光流的估计。

最后，RNN 输出的低分辨率结果经过上采样，变成了最下方的高分辨率光流图。这些彩色的图就是最终结果，它们表示了第 0 帧的每一个像素，在后续每一帧中的确切位置。颜色代表运动方向，颜色的深浅代表运动距离。

1.4.2 计算外观相似度

为了捕捉像素间的运动对应关系，AllTracker 的核心步骤是构建一个**多尺度 4D 相关性体 (Multi-scale 4D Correlation Volume)**。这个模块负责量化查询帧 (Query Frame) 中的特征与目标帧 (Target Frame) 中特征的外观相似度。

①**特征金字塔构建 (Feature Pyramid Construction)**：模型并不直接在单一分辨率上计算相关性，而是首先将目标帧的特征图 \mathbf{F}_t 转换为特征金字塔，以覆盖不同的感受野。

- **操作：** 对特征图进行不同步长 (Stride) 的平均池化 (Average Pooling)。
- **尺度设置：** 采用 5 个不同的步长 $\{1, 2, 4, 8, 16\}$ 。这对应了原图分辨率的 $1/8$ 到 $1/128$ 。这使得模型能够同时感知精细的局部纹理 (Stride 1) 和粗糙的全局结构 (Stride 16)，从而处理不同幅度的位移。

②**点积相关性计算 (Dot-product Correlation)**: 模型计算查询特征向量与目标金字塔中每一层特征向量的点积。

计算结果是一个**4D 相关性体**。它可以被理解为一个巨大的热力图集合 (Collection of Heatmaps)：对于查询帧的每一个像素，在每一个时间步 t 和每一个尺度上，都有一个对应的 2D 响应图。如果目标帧中的某个位置与查询像素高度匹配，该位置的热力图会出现强峰值 (Strong Peak)。这不仅提供了位置信息，也隐含了可见性线索。

这段代码实现了论文中的“**多尺度 4D 相关性体**” (Multi-scale 4D Correlation Volume)。它首先通过下采样构建特征金字塔，然后计算查询特征与每一层目标特征的点积，最后进行缩放归一化。

▼ CorrBlock代码

```
class CorrBlock:  
    def __init__(self, fmap1, fmap2, corr_levels, corr_radius):  
        self.num_levels = corr_levels  
        self.radius = corr_radius  
        self.corr_pyramid = []  
  
        # 论文 3.2 节：构建特征金字塔  
        for i in range(self.num_levels):  
            # 论文 3.2 节：计算点积相关性  
            corr = CorrBlock.corr(fmap1, fmap2, 1)  
  
            batch, h1, w1, dim, h2, w2 = corr.shape  
            corr = corr.reshape(batch*h1*w1, dim, h2, w2)  
  
            # 论文 3.2 节：通过平均池化(area interpolation)进行下采样，构建下一层金字塔  
            fmap2 = F.interpolate(fmap2, scale_factor=0.5, mode='area')  
            self.corr_pyramid.append(corr)  
  
    @staticmethod  
    def corr(fmap1, fmap2, num_head):  
        batch, dim, h1, w1 = fmap1.shape  
        h2, w2 = fmap2.shape[2:]  
        fmap1 = fmap1.view(batch, num_head, dim // num_head, h1*w1)  
        fmap2 = fmap2.view(batch, num_head, dim // num_head, h2*w2)  
  
        # 论文 3.2 节：矩阵乘法 (@) 对应公式中的点积 (Dot Product)  
        # 计算查询特征与目标特征的外观相似度  
        corr = fmap1.transpose(2, 3) @ fmap2  
  
        corr = corr.reshape(batch, num_head, h1, w1, h2, w2).permute(0, 2, 3, 1, 4, 5)  
  
        # 论文实现细节：缩放因子 1/sqrt(D)  
        # 防止点积数值过大导致梯度消失，稳定训练  
        return corr / torch.sqrt(torch.tensor(dim).float())
```

```

def __init__(self, seqlen, corr_levels=5, corr_radius=4, ...):
    # ...
    # 论文 3.2 节 & Table 7: 设置金字塔层级为 5
    self.corr_levels = corr_levels

    # 论文 3.4 节 & Table 7: 设置相关性查找半径 R=4
    # 这意味着每次查找会提取  $(2*4+1)^2 = 81$  个邻域点，共 5 层
    self.corr_radius = corr_radius
    self.corr_channel = self.corr_levels * (self.corr_radius * 2 + 1) ** 2

```

1.4.3 轨迹初始化

在进入循环神经网络（RNN）进行迭代细化之前，模型必须建立一个初始的输出张量。对于当前滑动窗口内的子序列（长度 S ），模型初始化一个低分辨率的张量，其形状为 $S \times H/8 \times W/8 \times 4$

这个 4 代表通道数，存的是模型对每一个像素在每一个时间步的预测结果。

通道 0 和 1：位置坐标； 通道 2：可见性（是否被遮挡）； 通道 3：置信度

坐标通道使用 **2D 网格 (Meshgrid)** 进行初始化。这意味着对于像素 (x, y) ，其初始位置被设为 $p_0 = (x, y)$ 。这隐含了一个“静止”的先验假设，即假设在迭代开始前像素位于其原始位置。可见性 (Visibility) 和置信度 (Confidence) 通道被直接初始化为零。

为了保持长视频跟踪的连续性，模型在处理非首个窗口时采用了热启动策略。

如果当前窗口不是第一个窗口（即 $window_idx > 0$ ），模型会将上一个窗口重叠部分的估计值直接填充到当前窗口的对应位置。

对于当前窗口中新增的时间步，模型会将上一个窗口最后一帧的估计值复制过来作为初始值。

这种机制避免了每次窗口移动都从零开始，确保了轨迹在跨越窗口边界时的平滑和连续。

坐标网格初始化

Plain Text |

```
1 #在Net类中，通过coords_grid方法创建坐标网格：
2 def coords_grid(self, batch, ht, wd, device, dtype):
3     # 创建二维网格坐标
4     coords = torch.meshgrid(torch.arange(ht, device=device, dtype=dtype),
5                             torch.arange(wd, device=device, dtype=dtype),
6                             indexing='ij')
7     coords = torch.stack(coords[::-1], dim=0) # 翻转坐标轴顺序
8     return coords[None].repeat(batch, 1, 1, 1) # 扩展为批次维度
9 #这个方法用于创建形状为(batch, 2, ht, wd)的坐标网格，其中坐标通道初始化为二维网格。
```

流场和可见性置信度初始化

Plain Text |

```
1 #在forward方法中，使用零值初始化流场和可见性置信度张量：
2 # 初始化全零的流场和可见性置信度张量
3 full_flows = torch.zeros((B,T,2,H,W), dtype=dtype, device=device)
4 full_visconfs = torch.zeros((B,T,2,H,W), dtype=dtype, device=device)
5
6 # 1/8分辨率的版本
7 full_flows8 = torch.zeros((B,T,2,H_pad//8,W_pad//8), dtype=dtype, device=device)
8 full_visconfs8 = torch.zeros((B,T,2,H_pad//8,W_pad//8), dtype=dtype, device=device)
9 #这些张量具有所需的形状S,H/8,W/8,4（考虑到最后的4个通道，其中2个用于坐标，2个用于可见性和置信度）。
```

▼ 窗口间的状态传递

Plain Text |

```
1 #在滑动窗口处理中，代码实现了状态从前一个窗口到下一个窗口的传递：
2 # 对于非第一个窗口，将之前计算的结果连接起来
3 if ii == 0:
4     flows8 = torch.zeros((B,S,2,H_pad//8,W_pad//8), dtype=dtype, device=device)
5     visconfs8 = torch.zeros((B,S,2,H_pad//8,W_pad//8), dtype=dtype, device=device)
6     fmaps2 = self.get_fmaps(images_[:,ara].reshape(-1,3,H_pad,W_pad), B,
7 S, sw, is_training).reshape(B,S,C,H8,W8)
7 else:
8     # 将前一个窗口的部分结果复制到新窗口中
9     flows8 = torch.cat([flows8[:,stride:stride+S//2],
10                         flows8[:,stride+S//2-1:stride+S//2].repeat(1,S//2,
11                         1,1,1)], dim=1)
11    visconfs8 = torch.cat([visconfs8[:,stride:stride+S//2],
12                         visconfs8[:,stride+S//2-1:stride+S//2].repeat(1,
13                         S//2,1,1,1)], dim=1)
14 #这种机制允许算法在处理长视频序列时，将重叠区域的估计值从前一个窗口传递到下一个窗口，并将
15 最后一个时间步的值向前复制。
```

▼ 跟踪访问状态管理

Plain Text |

```
1 #通过visits数组跟踪每个时间步是否已被处理：
2 visits = np.zeros((T)) # 初始化访问计数为0
3
4 # 当处理某个时间窗口时，增加对应位置的访问计数
5 visits[ara] += 1
6
7 # 对于未访问的时间步，使用最近的有效预测进行填充
8 invalid_idx = np.where(visits==0)[0]
9 valid_idx = np.where(visits>0)[0]
10 for idx in invalid_idx:
11     nearest = valid_idx[np.argmin(np.abs(valid_idx - idx))]
12     full_flows8[:,idx] = full_flows8[:,nearest]
13     full_visconfs8[:,idx] = full_visconfs8[:,nearest]
```

AllTracker通过以上方式实现了论文中描述的轨道初始化：

1. 使用coords_grid方法初始化坐标网格
2. 使用torch.zeros初始化流场和可见性置信度为零值
3. 通过滑动窗口机制在窗口之间传递重叠区域的估计值

4. 使用最近邻插值填补未直接处理区域的值

1.4.4 核心迭代细化 (Iterative Refinement)

迭代细化模块是通过一个循环神经网络 (RNN)，在 $1/8$ 分辨率的低维空间上循环执行 $K = 4$ 次，不断修正对轨迹、可见性和置信度的估计。

A. 局部特征构建与输入投影 (Local Feature Construction)

在每一次迭代步骤 k ，模型首先为查询帧中的每一个像素构建一个高维的特征向量，以此捕捉当前的上下文信息。该向量包含以下五部分：

- 外观特征 (Appearance, f): 静态特征，来自编码器输出。
- 当前状态 (State): 包含当前的可见性 (v) 和置信度 (c) 估计。
- 相对运动 (Relative Motion, m): 定义为 $m = p_k - p_0$ 。模型输入的是相对于初始位置的位移，赋予模型平移不变性。
- 相关性特征 (Correlation, q): 这是动态感知的核心。模型根据当前估计位置 p_k ，在 4D 相关性金字塔 中进行多尺度查表 (Look-up)。
- 在金字塔的每一层 L (共 5 层)，提取以 p_k 为中心、半径为 $R = 4$ 的局部邻域。
- 这产生了 $L \times (2R + 1)^2$ 个局部匹配线索。

所有分量拼接后，输入维度约为 665 通道。RNN 首先通过 1×1 卷积将其压缩融合，生成隐藏状态。

论文在附录中详细描述了 RNN 内部的“输入投影”过程：这些高维的输入特征（相关性、运动、可见性等）先被拼接，然后通过 1×1 卷积层进行融合和降维，生成 RNN 的隐藏状态输入。下面的图 5 (Figure 5) 中的 "Concat → Conv1x1" 流程图也直观地展示了这一步。

- f (外观特征): 256 通道
- q (相关性特征): 5 (层) \times $(2 \times 4 + 1)^2$ (邻域) $= 405$ 通道
- m (运动/位置): 2 通道
- v (可见性): 1 通道
- c (置信度): 1 通道
- 总计: $256 + 405 + 2 + 1 + 1 = 665$ 。

B. 混合时空 RNN 架构 (Hybrid Spatial–Temporal Architecture)

为了同时利用空间纹理信息和时间连续性信息，作者设计了交替时空处理 (Interleaved Spatial–Temporal) 的 RNN 结构

- 特征拆分 (Split Strategy): 为了降低显存占用，模型将特征图一分为二：一部分作为上下文特征 (Context) 保持固定，另一部分作为隐藏状态 (Hidden State) 在循环中更新。
- 空间层 (Spatial Layer): 采用 2D ConvNeXt Block (7×7 卷积核)。

作用：在空间维度上聚合信息，利用大卷积核解决光流中的孔径问题 (Aperture Problem)，即利用纹理丰富的区域来辅助推断纹理缺乏区域的运动。

- 时间层 (Temporal Layer): 采用 像素对齐的 Transformer Block (Pixel-aligned Attention)。

作用：对每一个像素位置，在时间轴 $S=16$ 上进行自注意力计算。

机制：因为所有特征图都已对齐到查询帧坐标系，这里的 Attention 实际上是在比较同一物理点在不同时刻的状态，从而有效处理遮挡和长距离关联。

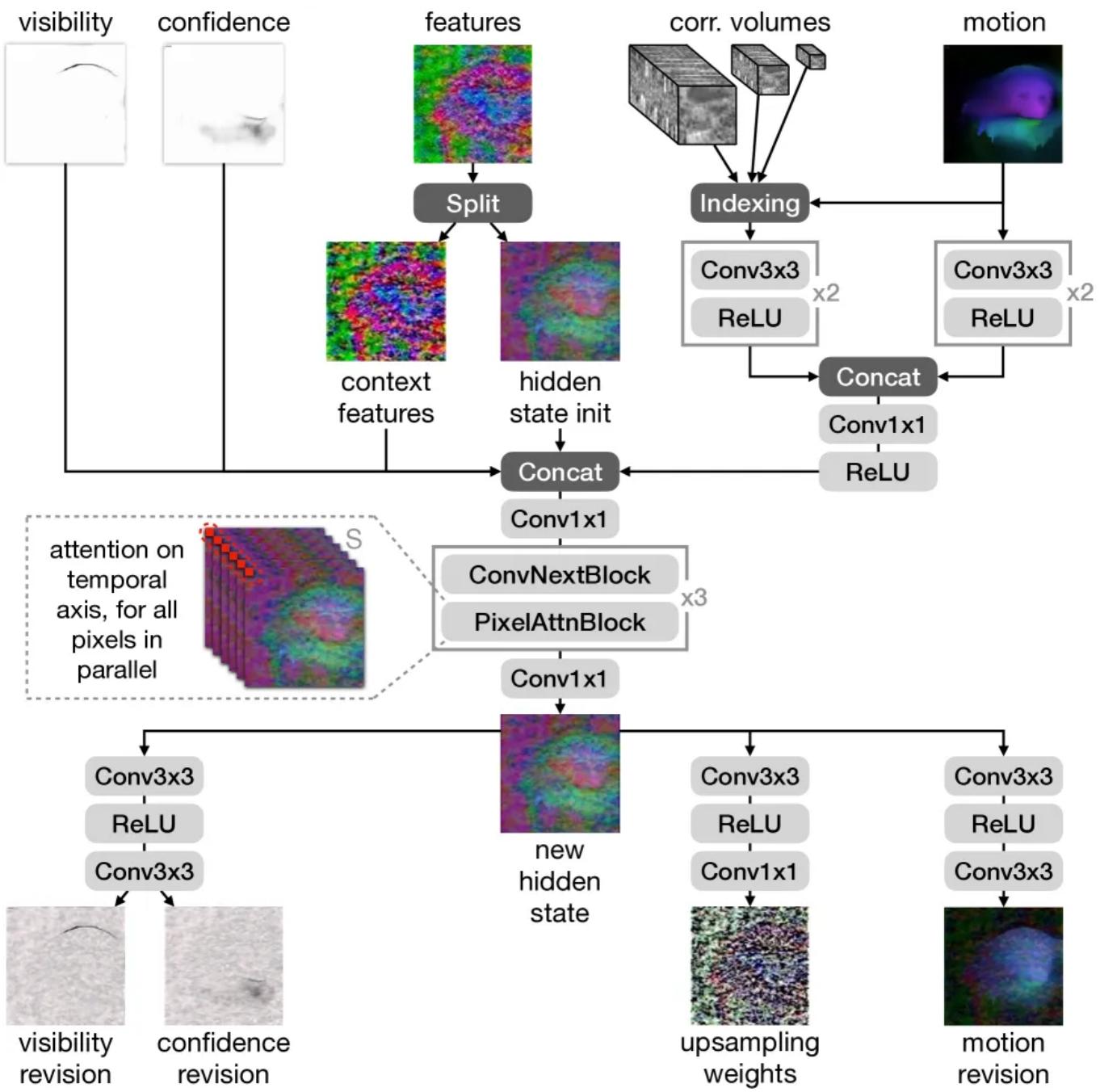
C. 残差更新与上采样 (Residual Update & Upsampling)

RNN 输出经过解码头映射为修正量 (Delta)，并应用残差更新公式：

$$\mathbf{x}_{k+1} = \mathbf{x}_k + \Delta \mathbf{x}_k, \quad \text{where } \mathbf{x} \in \{m, v, c\}$$

最后，模型预测 凸上采样掩码 (Convex Upsampling Mask)，利用 3×3 邻域的加权组合，将低分辨率的预测结果恢复为全分辨率光流图，以保持物体边缘的锐利度。

详细的步骤图，作者在附录里已经给出，如下图所示。



核心RNN更新模块

Plain Text |

```
1 在AllTracker项目中，核心的RNN更新模块是RelUpdateBlock类，定义在nets/blocks.py文件中。
2 class RelUpdateBlock(nn.Module):
3     def __init__(self, corr_channel, num_blocks, cdim=128, hdim=128, pdim=
4, use_attn=True, use_mixer=False, use_conv=False, use_convb=False, use_la
yer_scale=True, no_time=False, no_space=False, no_ctx=False):
5         super(RelUpdateBlock, self).__init__()
6         self.motion_encoder = BasicMotionEncoder(corr_channel, dim=hdim, p
dim=pdim) # B,hdim,H,W
7         self.no_ctx = no_ctx
8         if no_ctx:
9             self.compressor = conv1x1(cdim+hdim+2, hdim)
10        else:
11            self.compressor = conv1x1(2*cdim+hdim+2, hdim)
12        self.refine = []
13        for i in range(num_blocks):
14            if not no_time:
15                self.refine.append(CNBlock1d(hdim, hdim, use_attn=use_att
n, use_mixer=use_mixer, use_conv=use_conv, use_convb=use_convb, use_layer_
scale=use_layer_scale))
16            if not no_space:
17                self.refine.append(CNBlock2d(hdim, hdim, use_layer_scale=u
se_layer_scale))
18        self.refine = nn.ModuleList(self.refine)
19        self.final_conv = conv1x1(hdim, cdim)
20
21    def forward(self, flowfeat, ctxfeat, visconf, corr, flow, S, upsample=
True):
22        BS,C,H,W = flowfeat.shape
23        B = BS//S
24        motion_features = self.motion_encoder(flow, corr)
25        if self.no_ctx:
26            flowfeat = self.compressor(torch.cat([flowfeat, motion_feature
s, visconf], dim=1))
27        else:
28            flowfeat = self.compressor(torch.cat([flowfeat, ctxfeat, motio
n_features, visconf], dim=1))
29        for blk in self.refine:
30            flowfeat = blk(flowfeat, S)
31        flowfeat = self.final_conv(flowfeat)
32
33    return flowfeat
```

然后，这个模块在Net类中被实例化：

```
1 self.update_block = RelUpdateBlock(self.corr_channel, self.num_blocks, cdim
2 =dim, hdim=hdim, pdim=self.pdim,
3                                     use_attn=use_attn, use_mixer=use_mixer,
4                                     use_conv=use_conv, use_convb=use_convb,
5                                     use_layer_scale=True, no_time=no_time, n
6 o_space=no_space,
7                                     no_ctx=no_ctx)
```

▼ 特征投影与拆分

Plain Text |

1 在RelUpdateBlock的init方法中，有一个压缩器卷积层用于压缩输入特征：

```
2
3 if no_ctx:
4     self.compressor = conv1x1(cdim+hdim+2, hdim) # 例如：128+128+2 = 258通
5 else:
6     self.compressor = conv1x1(2*cdim+hdim+2, hdim) # 例如：2*128+128+2 = 3
86通道
```

7
8 虽然这里的具体数字可能不完全等于665，但这是特征压缩的关键部分。实际的输入维度取决于参数设
置。

▼ torch.split操作

Plain Text |

1 #在alltracker.py的forward_window方法中，有明确的特征拆分操作：

```
2 if self.no_split:
3     flowfeat, ctxfeat = fmap1.clone(), fmap1.clone()
4 else:
5     if flowfeat is not None:
6         _, ctxfeat = torch.split(fmap1, [self.dim, self.dim], dim=1)
7     else:
8         flowfeat, ctxfeat = torch.split(fmap1, [self.dim, self.dim], dim=1)
```

▼ 时空块堆叠

Plain Text |

```
1 #在RelUpdateBlock中，通过循环创建了一个交替的时间层和空间层堆叠结构：
2 self.refine = []
3 for i in range(num_blocks):
4     if not no_time:
5         self.refine.append(CNBlock1d(hdim, hdim, use_attn=use_attn, use_mix-
er=use_mixer, use_conv=use_conv, use_convb=use_convb, use_layer_scale=use_l-
ayer_scale))
6     if not no_space:
7         self.refine.append(CNBlock2d(hdim, hdim, use_layer_scale=use_layer_-
scale))
8 self.refine = nn.ModuleList(self.refine)
```

其中：

时间层：CNBlock1d，可以包含注意力机制(Attention)、变换器(Transformer)或运动编码器(MotionEncoder)

空间层：CNBlock2d，包含SepConv或ConvNeXt类型的操作，使用7x7卷积核

在前向传播中，这些层按顺序应用：

```
1 for blk in self.refine:
2     flowfeat = blk(flowfeat, S)
```

▼ 输出头与上采样

Plain Text |

```
1 #预测flow和mask的卷积层
2 在alltracker.py中定义了两个输出头：
3
4 self.flow_head = nn.Sequential(
5     nn.Conv2d(dim, 2*dim, kernel_size=3, padding=1),
6     nn.ReLU(inplace=True),
7     nn.Conv2d(2*dim, 2, kernel_size=3, padding=1)
8 )
9 self.visconf_head = nn.Sequential(
10    nn.Conv2d(dim, 2*dim, kernel_size=3, padding=1),
11    nn.ReLU(inplace=True),
12    nn.Conv2d(2*dim, 2, kernel_size=3, padding=1)
13 )
```

以及上采样权重预测：

```
1 self.upsample_weight = nn.Sequential(
2     # convex combination of 3x3 patches
3     nn.Conv2d(dim, dim * 2, 3, padding=1),
4     nn.ReLU(inplace=True),
5     nn.Conv2d(dim * 2, 64 * 9, 1, padding=0)
6 )
```

▼ upsample_flow函数

```
1 #其中上采样通过upsample_data方法实现:
2 def upsample_data(self, flow, mask):
3     """ Upsample [H/8, W/8, C] -> [H, W, C] using convex combination """
4     N, C, H, W = flow.shape
5     mask = mask.view(N, 1, 9, 8, 8, H, W)
6     mask = torch.softmax(mask, dim=2)
7
8     up_flow = F.unfold(8 * flow, [3,3], padding=1)
9     up_flow = up_flow.view(N, 2, 9, 1, 1, H, W)
10
11    up_flow = torch.sum(mask * up_flow, dim=2)
12    up_flow = up_flow.permute(0, 1, 4, 2, 5, 3)
13
14    return up_flow.reshape(N, 2, 8*H, 8*W).to(flow.dtype)
```

在更新循环中使用:

```
1 flow_update = self.flow_head(flowfeat)
2 visconf_update = self.visconf_head(flowfeat)
3 weight_update = .25 * self.upsample_weight(flowfeat)
4 flows8 = flows8 + flow_update
5 visconfs8 = visconfs8 + visconf_update
6 flow_up = self.upsample_data(flows8, weight_update)
7 flow_predictions.append(flow_up)
8 visconf_up = self.upsample_data(visconfs8, weight_update)
9 visconf_predictions.append(visconf_up)
```

1.4.5 损失函数与训练策略 (Model Training)

AIITracker 采用监督学习范式。由于模型是迭代输出的（共 $K=4$ 次迭代），作者采用了一个序列损失函数 (Sequence Loss)，对每一次迭代的预测结果都进行监督，但权重不同。

A. 轨迹损失 (Trajectory Loss, L_{track}) 这是核心损失函数，用于衡量预测位置与真实位置的差异。公式如下：

$$L_{track} = \alpha \sum_{k=1}^K \gamma^{K-k} \left(\frac{\mathbb{I}_{occ}}{5} + \mathbb{I}_{vis} \right) \|P_k - \hat{P}\|_1$$

L1距离 ($\|P_k - \hat{P}\|_1$): 使用曼哈顿距离衡量预测轨迹 P_k 与真值 \hat{P} 的误差。作者发现 L_1 比 Huber Loss 效果更好。

▼ L1距离计算

Plain Text |

```
1 i_loss = (flow_pred - flow_gt[j]).abs() # B, S, N, 2
2 i_loss = torch.mean(i_loss, dim=3) # B, S, N
```

序列权重 (γ^{K-k}): 随着迭代进行，权重指数级增加 ($\gamma = 0.8$)。这意味着模型越到后期，预测结果越重要，迫使模型不断精细化结果。

▼

Plain Text |

```
1 i_weight = gamma ** (n_predictions - i - 1)
2
3 gamma等于0.8 下面的代码可以看出可以改的
```

遮挡降权 ($\mathbb{I}_{occ}/5$): 这是一个关键的鲁棒性设计。对于可见点 (\mathbb{I}_{vis})，权重为 1；对于被遮挡点 (\mathbb{I}_{occ})，权重仅为 0.2 (1/5)。

原因：遮挡区域的真值往往不够精确（尤其是合成数据），且预测难度大。降低其权重可以防止模型为了强行拟合噪声标签而导致训练崩溃。

▼ 遮挡降权

Plain Text |

```

1  seq_loss_visible = utils.loss.sequence_loss(
2      coord_predictions,
3      traj_gts,
4      valids_gts,
5      vis=vis_gts,  # 可见点掩码
6      gamma=0.8,
7      use_huber_loss=args.use_huber_loss,
8      loss_only_for_visible=True,
9  )
10
11 seq_loss_invisible = utils.loss.sequence_loss(
12     coord_predictions,
13     traj_gts,
14     valids_gts,
15     vis=invis_gts,  # 不可见点掩码 (遮挡点)
16     gamma=0.8,
17     use_huber_loss=args.use_huber_loss,
18     loss_only_for_visible=True,
19 )
20
21 # 总损失计算中, 可见点权重为0.05, 不可见点权重为0.01 (即0.05/5)
22 total_loss = seq_loss_visible.mean()*0.05 + seq_loss_invisible.mean()*0.0
23 1 + vis_loss.mean() + confidence_loss.mean()

```

平衡系数 (α): 设为 0.05, 用于平衡轨迹损失与其他损失的数量级。

B. 可见性与置信度损失 (Visibility & Confidence Loss) 这两个分量使用二元交叉熵 (BCE) 进行监督:

可见性 (L_{vis}): 直接监督预测的遮挡掩码。

$$L_{vis} = \sum_k^K BCE(V_k, \hat{V})$$

▼ 可见性损失

Plain Text |

```

1 def sequence_BCE_loss(vis_preds, vis_gts, valids=None, use_logits=False):
2     total_bce_loss = 0.0
3     for j in range(len(vis_preds)):
4         n_predictions = len(vis_preds[j])
5         bce_loss = 0.0
6         for i in range(n_predictions):
7             if use_logits:
8                 loss = F.binary_cross_entropy_with_logits(vis_preds[j]
9 [i], vis_gts[j], reduction='none')
10            else:
11                loss = F.binary_cross_entropy(vis_preds[j][i], vis_gts
12 [j], reduction='none')
13                if valids is None:
14                    bce_loss += loss.mean()
15                else:
16                    bce_loss += (loss * valids[j]).mean()
17    bce_loss = bce_loss / n_predictions
18    total_bce_loss += bce_loss
19    return total_bce_loss / len(vis_preds)

```

置信度 (L_{conf}): 监督模型对自身预测的“自信程度”。

真值定义: 如果预测误差 $||P_k - \hat{P}||_2 < 12$ 像素, 则认为该预测是可信的 (Label=1), 否则不可信 (Label=0)。

公式如下

$$L_{conf} = \sum_k^K BCE(C_k, \mathbb{I}[||P_k - \hat{P}||_2 < 12])$$

置信度损失

Plain Text |

```

1 def sequence_prob_loss(
2     tracks: torch.Tensor,
3     confidence: torch.Tensor,
4     target_points: torch.Tensor,
5     visibility: torch.Tensor,
6     expected_dist_thresh: float = 12.0, #expected_dist_thresh=12.0 对
    应公式中的12像素阈值。
7     use_logits=False,
8 ):
9     """Loss for classifying if a point is within pixel threshold of its ta
    rget."""
10    # Points with an error larger than 12 pixels are likely to be useles
s; marking
11    # them as occluded will actually improve Jaccard metrics and give
12    # qualitatively better results.
13    total_logprob_loss = 0.0
14    for j in range(len(tracks)):
15        n_predictions = len(tracks[j])
16        logprob_loss = 0.0
17        for i in range(n_predictions):
18            err = torch.sum((tracks[j][i].detach() - target_points[j]) **
2, dim=-1)
19            valid = (err <= expected_dist_thresh**2).float() # 这里生成置信
度标签
20            if use_logits:
21                loss = F.binary_cross_entropy_with_logits(confidence[j]
[i], valid, reduction="none")
22            else:
23                loss = F.binary_cross_entropy(confidence[j][i], valid, red
uction="none")
24            loss *= visibility[j]
25            loss = torch.mean(loss, dim=[1, 2])
26            logprob_loss += loss
27            logprob_loss = logprob_loss / n_predictions
28            total_logprob_loss += logprob_loss
29    return total_logprob_loss / len(tracks)

```

C. 稀疏与密集监督的统一 (Unified Supervision) 为了支持联合训练，代码实现了一种通用的采样机制：

- **密集光流数据：** 全图计算 Loss。

- 稀疏点跟踪数据： 使用双线性采样 (Bilinear Sampling)，只在有 Ground Truth 的稀疏坐标位置提取预测值并计算 Loss

▼ 稀疏与密集监督的统一部分

Plain Text

```

1 # 使用双线性采样只在有 Ground Truth 的稀疏坐标位置提取预测值
2 traj_maps_e_ = traj_maps_e.reshape(B*T,2,H,W)
3 xy0_ = xy0.reshape(B,1,N,2).repeat(1,T,1,1).reshape(B*T,N,1,2)
4 trajs_e_ = utils.samp.bilinear_sampler(traj_maps_e_, xy0_) # B*T,2,N,1
5 trajs_e = trajs_e_.reshape(B,T,2,N).permute(0,1,3,2) # B,T,N,2

```

1.5. Results

该论文在多种公开基准数据集上评估了 AllTracker 的性能，涵盖了自然场景、合成场景、以及高分辨率视频。针对点跟踪任务，我们选用了 TAP–Vid [11]、BADJA [3]、RoboTAP [52] 等共 9 个数据集；针对光流任务，我们参考了 Sintel 基准。

[11] *TAP–Vid (NeurIPS 2022)*: 包含 TAP–Vid–DAVIS (真实世界视频) 和 TAP–Vid–Kinetics (YouTube 视频)，是目前最主流的长时程点跟踪基准。[3] *BADJA (CVPR 2018)*: 用于评估动物非刚性运动的跟踪性能。[52] *RoboTAP (ICRA 2024)*: 包含机器人操作场景，具有复杂的遮挡和物体交互。

论文使用 8 张 A100–40G GPU 进行训练，并在单张 A100 或 H100 上进行推理评估。与之前的 SOTA 方法相比，我们的系统在保持高效率的同时，刷新了各项精度记录。

评估指标： 主要指标为 δ_{avg} (位置准确率)，越高越好。其定义为预测点与真值点距离小于 $\{1, 2, 4, 8, 16\}$ 像素的比例的平均值。

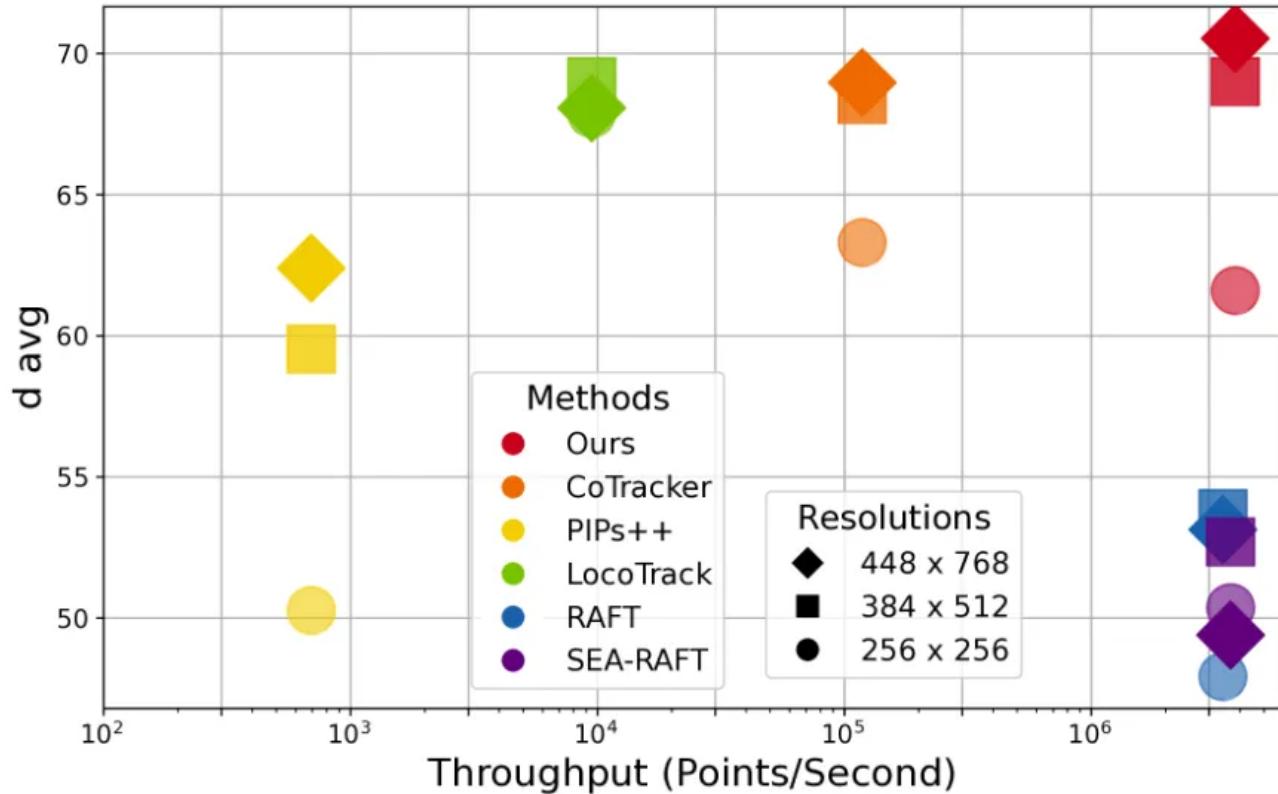


Figure 3: AllTracker (top right corner) delivers accurate multi-frame tracks at the throughput of an optical flow model.

这张散点图直观展示了 AllTracker (红色) 位于图表的“右上角”，意味着它在吞吐量 (速度) 和准确率 (Accuracy) 上都优于 CoTracker3、PIPs++ 等竞品。

1.5.1 Main Results

论文将 AllTracker 与当前最先进的点跟踪器（包括 CoTracker3 [24]、PIPs++ [59]）以及光流模型（SEA-RAFT [54]）进行了全面对比。

A. 标准分辨率下的 SOTA 性能 (Standard Resolution)

在 384×512 的标准输入分辨率下，AllTracker 展现出了卓越的跟踪精度。

总体表现：论文的完整模型 (AllTracker) 在 9 个基准数据集上的平均精度 δ_{avg} 达到了 66.1，超过了之前最强的 CoTracker3 (使用额外 15k 视频微调版) 的 65.0。

困难样本突破：在 RGB-Stacking [28] 这种包含大量无纹理区域的机器人操作数据集上，AllTracker 取得了显著的性能提升。这表明 Alltrackers 的混合架构能够比仅依赖稀疏点特征的模型更有效地利用大范围的空间上下文信息。

[24] CoTracker3 (NeurIPS 2024): 该竞品有两个版本。即使是使用了昂贵的伪标签自监督微调(+15k videos) 的版本，其平均性能仍略逊于仅使用合成数据混合训练的 AllTracker。

Table 1. Comparison against recent point trackers and optical flow models, across nine datasets. We evaluate δ_{avg} (higher is better), using an input resolution of 384×512 . The benchmarks are BADJA [3], CroHD [46], TAPVid-DAVIS [11], DriveTrack [1], EgoPoints [10] Horse10 [33], TAPVid-Kinetics [11], RGB-Stacking [28], and RoboTAP [52].

Method	Params.	Training	Bad.	Cro.	Dav.	Dri.	Ego.	Hor.	Kin.	Rgb.	Rob.	Avg.
RAFT [47]	5.26	Flow mix	23.7	29.3	48.5	44.8	41.0	27.8	64.3	82.8	72.2	48.3
SEA-RAFT [54]	19.66	Flow mix	23.9	21.9	48.7	49.4	44.0	33.1	64.3	85.7	67.6	48.7
AccFlow [55]	11.76	Flow mix	10.3	22.2	23.5	26.4	4.0	12.1	38.8	63.2	57.9	28.7
PIPs++ [59]	17.57	PointOdyssey	34.1	27.5	62.5	51.3	38.5	21.4	64.2	70.4	73.4	49.3
LocoTrack [6]	11.52	Kubric	41.4	43.1	68.0	66.5	58.4	48.9	70.0	80.3	76.9	61.5
BootsTAPIR [13]	54.70	Kubric+15M	42.7	34.9	67.9	66.9	56.8	48.8	70.6	81.0	78.2	60.9
DELTA [37]	59.17	Kubric	44.6	42.9	75.3	67.8	40.3	41.8	66.5	83.0	74.8	59.7
CoTracker2 [25]	45.43	Kubric	40.0	31.7	70.9	67.8	43.2	33.9	65.8	73.4	73	55.5
CoTracker3-Kub [24]	25.39	Kubric	47.5	48.9	77.4	69.8	58.0	47.5	70.6	83.4	77.2	64.5
CoTracker3 [24]	25.39	Kubric+15k	48.3	44.5	77.1	69.8	60.4	47.1	71.8	84.2	81.6	65.0
AllTracker-Tiny-Kub	6.29	Kubric	45.4	39.6	73.7	65.1	55.9	45.2	70.6	86.1	79.3	62.3
AllTracker-Tiny	6.29	Kubric+mix	47.5	39.8	74.3	63.9	58.3	45.5	71.5	88.1	80.7	63.3
AllTracker-Kub	16.48	Kubric	46.4	42.3	75.2	66.1	60.3	49.0	71.3	90.1	82.2	64.8
AllTracker	16.48	Kubric+mix	51.5	44.0	76.3	65.8	62.5	49.0	72.3	90.0	83.4	66.1

"Tiny" 模型 (AllTracker-Tiny) – 这是一个轻量级的模型变体，旨在测试更小的参数量下的性能。

"Kub" 模型 (AllTracker-Kub) – 这是一个训练数据受限的变体，代表“仅在 Kubric 数据集上训练”。

B. 高分辨率与显存效率 (High–Resolution & Efficiency)

这是 AllTracker 最具杀伤力的优势领域。论文将分辨率提升至 768×1024 进行测试。

分辨率扩展性： 随着分辨率提高，AllTracker 的性能稳步提升（从 66.1 升至 69.5）。相比之下，CoTracker3 的性能出现平台期甚至下降。

显存瓶颈突破： CoTracker3 在处理高分辨率时经常遭遇显存溢出 (OOM)，即使只跟踪稀疏点也需要昂贵的 H100 (96G) 显卡。而 AllTracker 凭借 1/8 的低分辨率推理网格，能够在一张标准的 A100–40G 上轻松运行，并同时输出 786,432 (全像素) 的密集轨迹。

架构优势： AllTracker 使用 stride=8 的特征编码，而 CoTracker3 使用 stride=4。这使得 AllTracker 在高分辨率下的内存占用大幅降低。

Table 2. High-resolution comparison between our model and CoTracker3 [24], which is the previous state of the art. We evaluate δ_{avg} (higher is better) on nine point tracking benchmarks, at resolutions 448×768 and 768×1024 . Parameter counts are in millions.

Model	Params.	Resolution	Bad.	Cro.	Dav.	Dri.	Ego.	Hor.	Kin.	Rgb.	Rob.	Avg.
CoTracker3-Kub [24]	25.39	448×768	49.7	59.9	78.8	70.2	57.4	50.6	70.4	81.1	76.6	66.1
CoTracker3 [24]	25.39	448×768	50.7	57.9	79.5	70.8	61.3	51.1	72.3	82.8	81.3	67.5
AllTracker-Tiny-Kub	6.29	448×768	47.7	48.0	76.3	67.8	57.3	48.3	71.2	84.8	79.0	64.5
AllTracker-Tiny	6.29	448×768	49.7	48.8	77.0	67.8	60.7	49.2	72.3	88.1	80.0	66.0
AllTracker-Kub	16.48	448×768	49.8	51.6	77.6	68.3	61.3	52.6	71.9	90.4	82.0	67.3
AllTracker	16.48	448×768	52.5	51.2	78.8	68.3	64.2	52.5	73.0	90.6	83.4	68.3
CoTracker3-Kub [24]	25.39	768×1024	47.8	62.0	77.0	72.0	51.9	46.4	67.0	76.2	74.5	63.9
CoTracker3 [24]	25.39	768×1024	49.8	64.3	79.6	72.4	58.4	48.5	71.1	77.9	80.2	66.9
AllTracker-Tiny-Kub	6.29	768×1024	48.4	56.9	78.5	71.2	55.1	49.2	71.1	80.8	78.1	65.5
AllTracker-Tiny	6.29	768×1024	51.6	57.0	79.1	71.1	59.2	50.7	72.4	87.4	79.2	67.5
AllTracker-Kub	16.48	768×1024	51.7	60.2	79.8	71.2	59.1	54.4	71.7	89.2	81.7	68.8
AllTracker	16.48	768×1024	53.6	53.4	80.6	72.3	64.3	54.6	73.1	90.6	83.2	69.5

C. 推理速度 (Inference Speed)

尽管 AllTracker 输出了全像素的密集结果，其推理速度依然极具竞争力。

- 对比结果：** AllTracker 的运行速度与光流模型（如 RAFT）相当，远快于需要繁重注意力计算的稀疏点跟踪器。
- 实时性：** 512×512 分辨率下，流式推理模式可达 **57.9 FPS**。

虽然这种模式下的精度略有下降 (δ_{avg} 从 66.1 降至 62.6)，但仍优于许多非实时竞品。

Realtime inference Our model’s sliding-window strategy can be run in “streaming” fashion real-time, at a penalty to accuracy: at **512×512 our model runs at 57.9 FPS** and achieves **62.6 δ_{avg}** (vs. 66.1 normally); BootsTAPIR [13] has also published a realtime variant, which runs at 21.4 FPS and reaches 62.2 δ_{avg} .

1.5.2. Qualitative Results & Ablation (定性结果与消融实验)

A. 定性结果：可视化对比 (Qualitative Results)

为了直观展示不同方法的差异，作者将稀疏点跟踪器的输出可视化为密集的流场图，并与光流方法进行对比。

稀疏点跟踪器 (如 CoTracker3, PIPs++): 尽管它们能跟踪长时程运动，但在将其可视化为密集流场时，往往会出现斑驳的图案错误 (Splotchy pattern errors)。这反映了它们在空间一致性上的缺失。

传统光流方法 (如 RAFT, SEA-RAFT): 它们生成的流场在局部空间上很平滑，但在长距离的大位移情况下，估计变得不可靠，且无法处理遮挡。

AllTracker: 我们的模型生成的运动场既保持了空间上的连贯性 (Coherent)，又在长时程上保持了准确性 (Accurate)。它成功结合了光流的平滑度与点跟踪的鲁棒性。

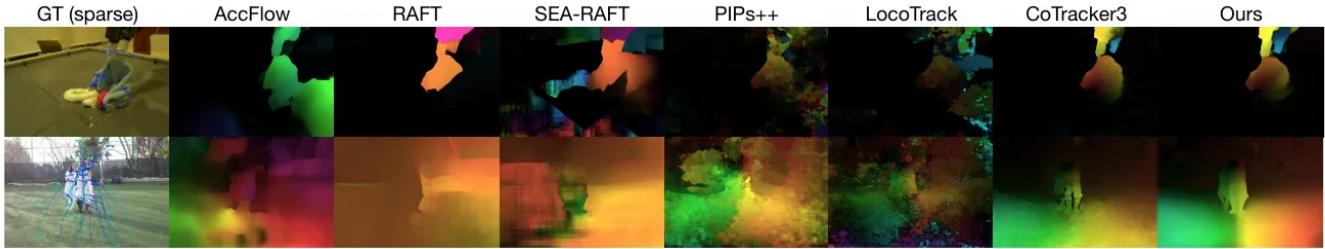


Figure 4. AllTracker produces accurate displacement fields across dozens of frames. Prior optical flow methods struggle to make correspondences across wide time gaps, while our model uses temporal priors to resolve the ambiguity; prior point trackers take multiple minutes to produce output at this density, and show splotchy pattern errors, while our method produces coherent output in less than a second.

这张图展示了不同模型在生成长距离光流时的表现。重点指出 CoTracker3 的“斑驳”感 (Splotchy) 和 AllTracker 的平滑度。

B. 消融实验：设计验证

作者在缩减的训练设置下（仅在 Kubric 上训练 100k 步）进行了一系列消融实验，以验证架构设计的有效性。

时间模块 (Temporal Module): 对比了 Transformer、MLP-Mixer 和 1D 卷积。

结论：Transformer 效果最好 ($\delta_{avg} = 56.8$)。此外，Transformer 对输入长度不敏感，这对联合训练至关重要。

运动表示 (Motion Representation): 对比了“相对于查询帧 ($0 \rightarrow t$)”和“相对于上一帧 ($t - 1 \rightarrow t$)”的运动编码。

结论：使用相对于查询帧 (Query-relative) 的位移效果最好。这与我们将问题重构为“长距离光流”的思路一致。

骨干网络与超参数 (Backbone & Hyperparams)

结论：使用预训练的 ConvNeXt 优于 BasicEncoder。最佳超参数设置为：3 个细化块，相关性半径 $R = 4$ ，5 个金字塔尺度。

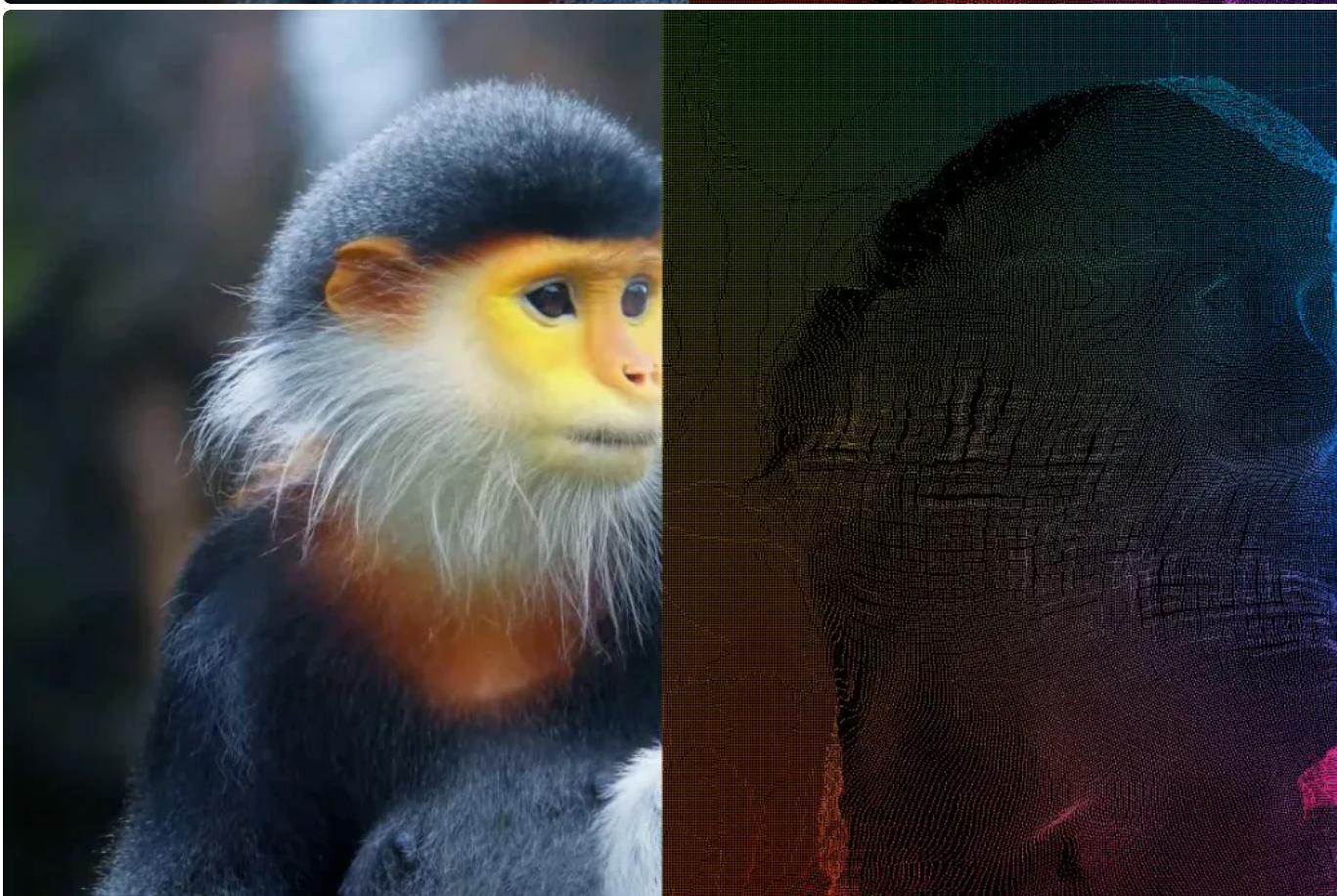
Table 5. A transformer-based temporal module performs better than mixer-based or convolution-based variants.

Time component	Acc.
Transformer [51]	56.8
MLP-Mixer [48]	56.1
ConvNext-1D [31]	54.6
Conv-1D	55.8

Table 6. Representing motion as long-range flow works better than instantaneous velocity, and positional embeddings do not help.

Motion representation	Acc.
$0 \rightarrow t$	56.8
$\text{emb}(0 \rightarrow t)$	54.8
$t-1 \rightarrow t$	56.4
$\text{emb}(t-1 \rightarrow t)$	55.3

1.6 测试截图





这张图直观展示了 AllTracker 在高分辨率下的全像素密集跟踪能力。左侧是原始视频帧，右侧是模型输出的对应光流可视化图，模型不仅完美捕捉了猴子头部的整体运动轨迹，甚至连边缘极其细微的毛发纹理都在光流图中清晰可见，且整体色彩过渡非常平滑，没有出现断裂或噪点。

1.7. Limitations and Future Work

尽管 AllTracker 在点跟踪基准上取得了 SOTA 性能，但有趣的是，它并没有使传统的光流方法变得多余。在纯粹的光流估计任务上（即两帧之间的运动估计），AllTracker 目前还无法超越最先进的光流模型（如 SEA-RAFT）。模型似乎在短距离运动估计上存在欠拟合，定性观察表明 AllTracker 生成的流场图比 SEA-RAFT 的更粗糙。

光流估计与点跟踪的权衡： AllTracker 的设计初衷是优化长时程的轨迹一致性（Long-range consistency），这可能导致它在关注微小的、瞬时的帧间运动细节时不如专门为为此优化的 SEA-RAFT 敏锐。但这暗示了通过增加计算量或模型容量，未来有可能获得更好的短距精度。

另一个限制在于**时间窗口的大小**。受限于当前的 GPU 显存，我们目前只能在训练和测试中使用有限长度的滑动窗口（如 16 帧）。这意味着对于超过这个长度的长时程遮挡，模型的处理能力受到物理算力的制约。

长时程遮挡 (Long-term Occlusion)： 指物体被遮挡的时间超过了模型一次能“看到”的时间窗口。如果窗口更宽，模型就能利用更久远的过去信息来填补当前的空白。随着更大显存 GPU 的普及，简单的扩大窗口尺寸可能直接解决此问题。

在未来的工作中，一个广阔的方向是增加对**物理和常识性运动约束**的感知。目前的模型纯粹基于数据驱动学习像素运动，未来可以尝试引入 3D 感知 或更具表达力的模型设计，使跟踪更符合物理世界的规律。

1.8. Conclusion

该论文提出了 AllTracker，这是一种将点跟踪任务彻底重构为多帧光流估计的创新方法。该系统打破了“稀疏点跟踪”与“密集光流”的界限，在实现高分辨率（ 768×1024 ）和 全像素密集输出 的同时，在多个点跟踪基准上实现了最先进的性能（SOTA）。

AllTracker 的出现使得现有的稀疏点跟踪器（在同等速度和精度下）变得多余，因为它能提供更丰富、更密集的上下文信息。我们的工作证明了通过混合时空架构和联合训练策略，可以在一个统一的框架内高效解决长时程、高分辨率的视频跟踪难题