## Detailed Explanation of Changes and Code Workflow

### Overview

Initially, a custom Graph Convolutional Network (GCN) model trained on MPI-INF-3DHP data was being used to predict 3D joint positions from 2D MediaPipe keypoints. However, multiple issues arose:

1. **Joint-Index Mismatch**: The GCN was trained on 28 joints (MPI format), while MediaPipe produces 33 landmarks in a different order.
2. **Normalization & Scaling Errors**: Projecting 3D outputs to 2D incorrectly, causing coordinates to fall off-screen or collapse.
3. **Visualization Bugs**: The draw\_3d\_pose function assumed the model’s output was already in normalized [0,1] range and used a skeleton edge list that didn’t match the model’s training edges.

To address these, we replaced the GCN-based pipeline with a straightforward usage of MediaPipe’s 2D pose landmarks. Below is a detailed rundown of what changed, why it was necessary, how the final code works, and a description of the original model and data used.

## Part 1: The Custom GCN Model and MPI-INF-3DHP Data

### 1. Model Architecture: PoseEstimator (in model.py)

* **Input**: Takes an input tensor of shape (B, 28, 3) representing 28 joints, each with 3D coordinates (x, y, z).
* **Positional Encoding**: A PositionalEncoding module adds sinusoidal positional embeddings across the 28 joint positions, enabling the Transformer to understand joint ordering.
* **Sparse GCN Blocks**: Two SparseGCNBlock modules perform graph convolution using torch\_geometric.nn.GCNConv, followed by layer normalization and ReLU, with a residual connection. These learn pairwise relationships between joints defined by the edge index.
* **AttentionRoutingTransformer**: A small Transformer encoder (2 layers, 4 heads) further refines the node features. It uses a learned positional encoding to inject the notion of sequence (joint) order.
* **Head**: After graph + transformer processing, a linear head with a ReLU in between maps each node’s embedding to a final 3D output (x, y, z).

**Edge Index Definition** (in model.py):

edges = [

(0,1), (1,8), (8,12),

(1,2), (2,3), (3,4),

(1,5), (5,6), (6,7),

(8,9), (9,10), (10,11),

(8,13), (13,14), (14,15),

(0,16), (0,17)

]

edges += [(j,i) for i, j in edges]

return torch.tensor(edges, dtype=torch.long).t().contiguous()

This edge list encodes an MPI-INF-3DHP skeleton: head-to-neck, limbs, torso. The GCN propagates information along these edges.

### 2. MPI-INF-3DHP Dataset and Custom Extraction

* **MPI-INF-3DHP** is a publicly available benchmark dataset containing multi-view 3D pose annotations captured in a motion-capture studio. It provides:
  + pose2d: 2D joint positions (in normalized camera coordinates) for 28 joints.
  + pose3d: Corresponding 3D joint positions (in world coordinates).
  + Multiple subjects (S1–S8) with two sequences each (Seq1, Seq2).
* **Data Extraction**: We wrote extract\_mpi\_inf\_3dhp.py to:
  + Load Mat-files (annot.mat) containing annot2 and annot3 structures for 2D and 3D.
  + Reshape them into (frames, joints, coords) arrays, filter by valid frames, and save as individual compressed .npz files per subject & sequence.
  + Concatenate all subjects’ .npz files into a single mpi\_inf\_combined.npz with keys:
  + pose2d = np.concatenate([pose2d\_all]) # shape: (N\_total, 28, 2)
  + pose3d = np.concatenate([pose3d\_all]) # shape: (N\_total, 28, 3)
  + In train\_pose\_model\_v2.py, we append a zero z-coordinate to pose2d when only 2D data is present, yielding (N, 28, 3).
* **Normalization**: During dataset initialization, we compute:
* self.pose2d\_mean = pose2d.mean(axis=(0,1)) # shape: (3,)
* self.pose2d\_std = pose2d.std(axis=(0,1)) # shape: (3,)
* self.pose2d = (pose2d - self.pose2d\_mean) / (self.pose2d\_std + 1e-6)

We save pose2d\_mean\_std.npy so that inference code can use the same normalization.

* **Training Loop** (in train\_pose\_model\_v2.py):
  + Batch size: 64.
  + Optimizer: Adam (lr=1e-3, weight\_decay=1e-4).
  + Loss: MPJPE (mean per-joint position error) + bone-length regularization.
  + Bone edges (same as GCN edges) penalize deviations from unit bone length.
  + Learning rate scheduler: ReduceLROnPlateau on validation MPJPE.
  + Best model weights are saved to best\_model\_weights.pth.

## Part 2: Why the GCN Pipeline Ultimately Failed for Overlay

1. **Joint-Index & Skeleton Mismatch**: The GCN expected exactly the MPI-INF joint order. MediaPipe’s 33-point output required careful remapping, but even with slicing or indexing, the skeleton edges and joint semantics never aligned perfectly. This led to inaccurate or collapsed skeletons when drawing.
2. **3D-to-2D Projection Distortion**: We tried normalizing the GCN’s 3D outputs by min-max scaling before drawing. Because the GCN’s absolute scale was arbitrary, the skeleton frequently went off-screen or appeared as a straight line.
3. **Excessive Complexity**: Combining MediaPipe → custom GCN → projection → custom visualizer introduced too many places for bugs. Time spent troubleshooting joint mismatches and projection logic outweighed the benefit of a custom 3D lift.

Hence, we opted for a **direct 2D overlay** using MediaPipe’s 2D output, which is guaranteed to match the image exactly.

## Part 3: Final 2D Overlay Code

Below is a summary of how the final, simplified pipeline works.

### visualize.py

import cv2

import numpy as np

# 33‐point 2D connections (MediaPipe’s built-in skeleton)

POSE\_CONNECTIONS = [

(0,1), (1,2), (2,3), (3,7),

(0,4), (4,5), (5,6), (6,8),

(9,10),

(11,12),

(11,13), (13,15),

(12,14), (14,16),

(11,23), (12,24),

(23,25), (25,27),

(24,26), (26,28),

(27,29), (28,30),

(29,31), (30,32)

]

BONE\_COLOR = (0, 255, 0) # Green

JOINT\_COLOR = (0, 0, 255) # Red

JOINT\_RADIUS = 4

BONE\_THICKNESS = 2

TEXT\_COLOR = (255, 255, 255) # White

def draw\_2d\_pose(frame, landmarks):

H, W = frame.shape[:2]

# Convert normalized (x,y) ∈ [0,1] → pixel coords

pts\_px = np.array([[int(lm.x \* W), int(lm.y \* H)]

for lm in landmarks]) # (33,2)

# Draw bones (green) between connected pairs

for (i, j) in POSE\_CONNECTIONS:

if i < len(pts\_px) and j < len(pts\_px):

p1 = tuple(pts\_px[i]); p2 = tuple(pts\_px[j])

if p1 != p2:

cv2.line(frame, p1, p2, BONE\_COLOR, BONE\_THICKNESS)

# Draw joints (red circles) and labels

for idx, pt in enumerate(pts\_px):

cv2.circle(frame, tuple(pt), JOINT\_RADIUS, JOINT\_COLOR, -1)

cv2.putText(frame, str(idx), tuple(pt), cv2.FONT\_HERSHEY\_SIMPLEX,

0.4, TEXT\_COLOR, 1)

return frame

### predict\_image.py

import cv2

import mediapipe as mp

from visualize import draw\_2d\_pose

# 1) Initialize MediaPipe Pose for 2D landmarks

tmp = mp.solutions.pose

pose\_obj = tmp.Pose(

static\_image\_mode=True,

model\_complexity=2,

enable\_segmentation=False,

min\_detection\_confidence=0.3,

min\_tracking\_confidence=0.3

)

# 2) Read & resize the input image

image\_path = "input.jpeg"

frame = cv2.imread(image\_path)

if frame is None:

raise FileNotFoundError(f"❌ Could not find image: {image\_path}")

frame = cv2.resize(frame, (480, 480))

frame\_rgb = cv2.cvtColor(frame, cv2.COLOR\_BGR2RGB)

# 3) Run MediaPipe

results = pose\_obj.process(frame\_rgb)

if not results.pose\_landmarks:

print("❌ No pose detected. Try a different image or higher resolution.")

exit()

# 4) Draw 2D pose overlay

output\_img = draw\_2d\_pose(frame.copy(), results.pose\_landmarks.landmark)

# 5) Show & save

cv2.imshow("2D Pose Overlay", output\_img)

cv2.imwrite("output.jpg", output\_img)

print("✅ Saved output to 'output.jpg'")

cv2.waitKey(0)

cv2.destroyAllWindows()

**Explanation:**

* We take MediaPipe’s 33 normalized (x,y) outputs (landmarks) and multiply by the frame’s pixel width/height to get exact pixel coordinates. No further normalization is needed.
* Using POSE\_CONNECTIONS, we draw the skeleton exactly as MediaPipe expects—this ensures perfect alignment.
* Red circles mark each joint; green lines connect bones. Labels show joint indices for debugging if needed.

## Summary of the Full Workflow

1. **Data Preparation & Model Training (Original)**
   * Extracted MPI-INF-3DHP sequences using extract\_mpi\_inf\_3dhp.py, combining into mpi\_inf\_combined.npz.
   * Trained a GCN-based PoseEstimator on 28-joint data (2D + zero-depth → 3D ground truth), saving normalization stats and best model weights.
2. **Attempted Inference & Visualization (Original)**
   * Used MediaPipe 2D landmarks → remapped to 28 MPI indices → normalized → GCN inference → 3D output → projected back to 2D → custom draw\_3d\_pose.
   * Encountered mismatches in joint order and incorrect projections, resulting in inaccurate skeleton overlays.
3. **Final Simplified Pipeline**
   * **Bypass the GCN entirely for 2D overlay**.
   * Use MediaPipe’s built-in 2D pose detector, read results.pose\_landmarks directly.
   * Draw 2D skeleton with draw\_2d\_pose, which maps normalized (x,y) to pixel coordinates and connects bones via POSE\_CONNECTIONS.

**The final code** provides a pixel-perfect, fully aligned 2D skeleton overlay on the input image. This satisfies the requirement of “accurate 2D graph from the picture” without any downstream 3D complications.