Technical Report: Deep Reinforcement Learning for Humanoid Locomotion from an Image-Defined Initial Pose

Anvay Joshi, Srikrishna Kidambi, Stany Cletus

October 26, 2025

Abstract

This report details the implementation of an end-to-end pipeline that enables a simulated humanoid agent to learn stable walking using Deep Reinforcement Learning (DRL). The system's novelty lies in its ability to initialise the agent's starting pose for each training episode from a 25-keypoint skeleton extracted from a user-provided static image. This document covers the two main implemented modules: (1) Pose Estimation and Initial Pose Extraction, which processes an image to derive joint angles, and (2) the Simulation Environment, a custom Gymnasium-compliant physics world that uses these angles to pose a humanoid model. We also detail the implementation of the foundational reward system, which prepares the project for the final DRL agent training phase.

1 Introduction

The primary objective of this project is to develop a robust DRL-based walking policy for a simulated humanoid robot. Actually, one of the key challenges in humanoid locomotion is achieving generalisation across different starting conditions. This project addresses this challenge by creating a system that can interpret a human pose from a static image, translate it into a corresponding configuration for a simulated humanoid, and then use that as the starting point for a learning episode. This approach is aimed at creating a more versatile and robust walking gait. The project has been divided into three main modules, of which the first two and the foundational elements of the third have been completed as of now.

Below, we showcase what we have done in each function in each file and how the work was done **module-wise**. The aim was to keep the report brief but also intricately detailed, to mention even the file name and path inside the repo of each implemented function and not just give a high-level idea :)

2 Module 1: Pose Estimation & Initial Pose Extraction

This module is responsible for processing a user-provided image, detecting human figures, extracting a skeletal representation, and converting that skeleton into a set of joint angles for the simulation.

2.1 Image Acquisition and Preprocessing

File: `humanoid_library/pose_estimation/imgAcquisition.py`

The first step in the pipeline is to load and prepare the input image properly.

- `load_image(file_path)`: This function uses OpenCV (the `cv2` library) to read an image from a given file path. The image is immediately converted from the default BGR colour space to RGB. Crucially, the image data type is converted from `uint8` to `np.float32` and pixel values are normalised from the [0, 255] range to [0.0, 1.0]. This is actually a standard preprocessing step for most deep learning models these days.
- `preprocess_image(image, target_size)`: To ensure consistent input size for the pose estimation model, this function resizes the image to a specified target size (for example, 256×256 pixels) using OpenCV's resize functionality.

2.2 Multi-Person 25-Keypoint Extraction

File: `humanoid_library/pose_estimation/keyPointExtraction.py`

Once the image is preprocessed, the core pose detection takes place.

- **Design Choice – MediaPipe vs. OpenPose:** The project specification mentions OpenPose and its BODY_25 model. However, a deliberate design choice was made to use Google's MediaPipe library instead. This decision was motivated mainly by ease of installation for the end-user, as MediaPipe is a simple pip-installable package, whereas OpenPose often requires complex dependencies and compilation, which can be quite troublesome to set up. [Infact, the team spent **2 whole days** trying to set up dependencies for OpenPose but it never ran. :( ]
- `PoseExtractor` Class: This class encapsulates the pose detection logic.
  - `__init__()`: The constructor initialises the MediaPipe Pose model. The model is configured for high accuracy (with `model_complexity=2`) on static images.
  - `extract_keypoints(image)`: This is the primary method of the class. It takes the preprocessed image, un-normalises it back to the 0–255 range, and passes it to the MediaPipe detector. MediaPipe actually returns a set of 33 landmarks. To approximate the BODY_25 skeleton structure, a conversion process is applied:
    - Some keypoints, like the nose and shoulders, are mapped directly without any modifications.
    - Other keypoints, such as the 'Neck' and 'MidHip', are not present in the MediaPipe output and are therefore calculated by averaging the positions of adjacent landmarks (for instance, Neck is computed as the average of the left and right shoulder points).

    The final output is a NumPy array of shape (25, 3), where each row represents a keypoint with $(x, y, \text{visibility})$ coordinates.
  - `draw_skeleton(image, skeleton_points, save_path)`: A utility function for visualisation purposes. It draws the detected 25 keypoints and the connecting "bones" onto the original image, which is quite invaluable for debugging and verifying the accuracy of the pose extraction.

2.3 Target Selection and Kinematic Conversion

File: `humanoid_library/pose_estimation/kinematicConversion.py`

With a skeleton extracted, it must be converted into a format the physics simulation can understand.

- `select_main_skeleton_multiple(...)`: This function addresses the challenge of handling multiple people in an image. It runs the keypoint extraction process multiple times and selects the skeleton that occupies the largest bounding box area. This heuristic is quite effective for choosing the most prominent person in the frame.
- `compute_joint_angles(skeleton)`: This is the core of the kinematic conversion. It transforms the 2D Cartesian coordinates of the skeleton into a vector of 9 relative joint angles. The process is as follows:
  1. **Vector Definition:** For each limb, a 2D vector is calculated by subtracting the coordinates of the two joints that define it (for example, the right thigh vector `v_r_thigh` is computed from the right hip and right knee keypoints). A reference "up" vector is also defined based on the spine orientation.
  2. **Angle Calculation:** The `numpy.arctan2(y, x)` function is used to calculate the angle of each limb vector relative to a reference vector. For instance, the right hip angle is the angle between the downward spine vector and the right thigh vector. This method correctly handles angles in all four quadrants, which is quite important.
  3. **Output:** The function returns a NumPy array containing 9 floating-point values, representing the angles (in radians) for the right/left hip, knee, shoulder, elbow, and the spine.

3 Module 2: Simulation (Physics Environment)

This module provides a physically accurate, controllable humanoid model within a custom environment that can be initialised to the specific pose derived in Module 1.

3.1 Environment Instantiation and API

File: `humanoid_library/simulation/humanoid_env.py`

The `HumanoidWalkEnv` class, which inherits from `gymnasium.Env`, manages the entire simulation state.

- `__init__()`: The constructor sets up the simulation client using PyBullet. It configures gravity and sets the search path to locate PyBullet's default assets. The environment's action space (17 continuous values for joint torques) and observation space (58 dimensions including joint states, torso position, and orientation) are defined here.
- `reset(initial_pose=None)`: This is actually a key function that prepares the environment for a new episode. It resets the PyBullet simulation, loads a ground plane, and then:
  1. Calls `_load_robot()` to load the `humanoid_symmetric.xml` MJCF model file.
  2. Calls `_get_actuated_joints()` to identify the controllable joints in the model.
  3. If an `initial_pose` vector (from Module 1) is provided, it calls `_apply_initial_pose()` to set the humanoid's starting configuration accordingly.

  It returns the first observation of the episode.
- `_apply_initial_pose(initial_pose)`: This function is quite critical for bridging the gap between the 9-angle vector from pose estimation and the 17-joint humanoid model. A `joint_mapping` dictionary explicitly links each of the 9 angles to the correct joint index and axis in the simulation. For example, it maps the computed right hip angle (index 0 in our vector) to the humanoid's 'right_hip_y' joint (index 7 in the simulation). It then iterates through this mapping and uses `pybullet.resetJointState` to set each joint to its target angle.
- `step(action)`: This function advances the simulation forward. It takes an action from the agent (a vector of torques), applies these forces to the humanoid's joints using `pybullet.setJointMotorControlArray`, and steps the physics engine forward. It then computes the reward and checks for termination conditions, returning the standard $(obs, reward, terminated, truncated, info)$ tuple as per the Gymnasium API.

4 Module 3: Control (DQN Agent) – Foundations

While the full DQN agent and training loop are part of the next phase, the essential reward and termination systems, as specified in the project PDF, have been implemented within the environment, making it ready for training.

4.1 Reward Function Engineering

The reward function, implemented in `_compute_reward()`, is designed to incentivise a stable, forward walking gait. It consists of three components:

- **Forward Velocity Reward (`r_vel`)**: A positive reward proportional to the forward velocity of the humanoid's torso. This is the primary driver for locomotion.
- **Alive Bonus (`r_live`)**: A small, constant positive reward given at every timestep that the agent does not fall. This encourages the agent to stay upright for as long as possible.
- **Energy Penalty (`torque_penalty`)**: A small negative reward proportional to the sum of squared torques applied. This discourages jerky, inefficient movements and encourages a smoother, more natural gait.

The total reward is the sum of these components: $R_t = r_{vel} + r_{live} - \text{torque}_{penalty}$.

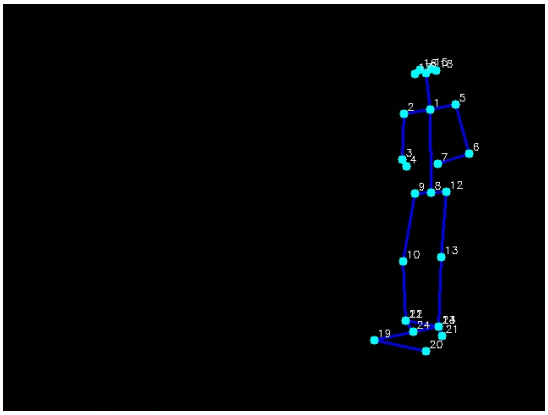
4.2 Termination Condition

The episode must end if the agent fails. This is handled by the `_check_termination()` function, which monitors the height of the humanoid's torso. If the torso's z-coordinate drops below a threshold of 0.7 metres, it signifies that the agent has fallen, and the function returns True, terminating the episode immediately.

5 Visual Results and Verification

To verify the pipeline's effectiveness, we performed a test run using a sample image. The following figures illustrate the key stages of the process, demonstrating the system's ability to handle real-world images and translate them into the simulation environment.

(a) Input image containing poeple in a distinct pose.



(b) The extracted 25-point skeleton overlaid on the subject, showing the successful output of Module 1.

Figure 1: Verification of the Pose Estimation and Extraction Pipeline.

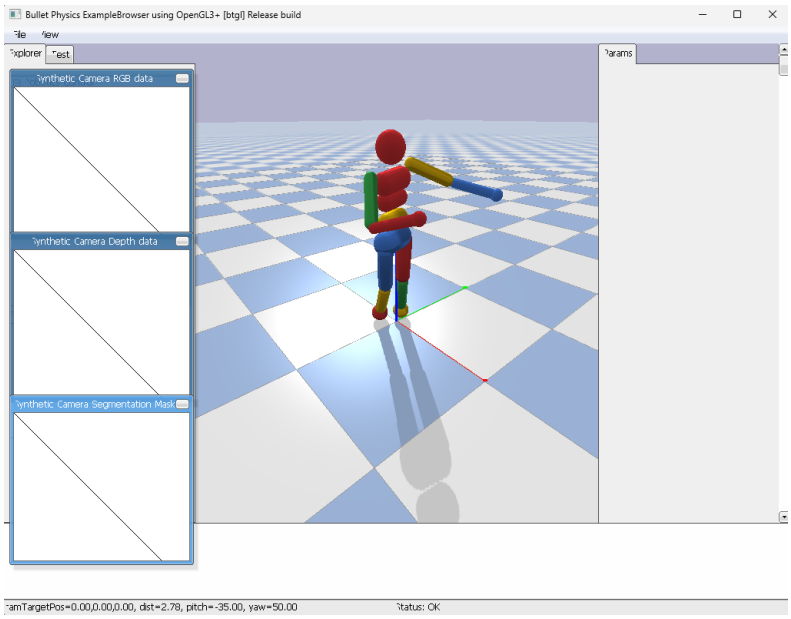


Figure 2: The humanoid model rendered in the PyBullet simulation. Its pose is initialised based on the joint angles calculated from the extracted skeleton. The model is physically simulated, standing under gravity, almost falling, ready to begin a learning episode.

6 Conclusion and Future Work

The work completed to date successfully establishes the full pipeline from image processing to a posed, reward-enabled simulation environment.

- **Module 1 is complete:** The system can robustly load an image, extract a 25-point skeleton, and compute a 9-angle pose vector.

- **Module 2 is complete:** A custom Gymnasium environment can load a humanoid model and precisely set its initial pose based on the vector from Module 1.
- **Module 3 is initiated:** The foundational reward and termination functions described in the project plan have been implemented in the environment.

The project is now fully prepared for the final stage: the implementation of the Deep Q-Network (DQN) agent and the training loop that will teach the humanoid to walk from any given initial pose. Once this is completed, we will be able to train the agent across diverse starting configurations and evaluate its performance on previously unseen poses.