Indian Institute of Technology Tirupati
Department of Computer Science and Engineering
Intelligent Systems Lab (CS303P/CS519P)

Update: Weekly Instructors: Chalavadi Vishnu Jul-Dec 2025

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

To develop an end-to-end pipeline through a custom python library (with `setup.py`) that enables a simulated humanoid agent to learn stable walking using Deep Reinforcement Learning. The agent's initial starting pose for each training episode will be determined by a **25-keypoint skeleton** extracted from a user-provided static image. The system must be robust enough to learn to initiate walking from a variety of starting poses.

Expected Inputs:

1. A static image file (e.g., `.jpg`, `.png`) provided by the user. The image can contain a single person or multiple people in various poses.
2. A pre-defined humanoid model specified in a URDF (Unified Robot Description Format) or MJCF file.

Expected Outputs:

1. A trained Deep Q-Network (DQN) model representing a robust walking policy that can be initiated from numerous starting configurations.
2. Utilize a multi-person computer vision model to accurately extract a 25-keypoint skeleton for every person detected in the user's image.
3. Implement a selection mechanism to choose a target skeleton if multiple people are detected through template matching using ORB/SIFT/SURF/Haar Cascades.
4. Develop a custom `gymnasium`-compliant physics environment with a `reset` function capable of setting the humanoid's joint angles to match the initial pose derived from the image, and other poses to be generated through GANs.
5. Implement a DQN agent to solve the locomotion task, which involves a high-dimensional, continuous state space and a discretized action space.
6. Engineer a reward function that primarily encourages forward velocity and stability, enabling the agent to learn a dynamic walking gait regardless of the initial pose.

Module 1: Pose Estimation & Initial Pose Extraction

Task 1.1: Image Acquisition and Preprocessing

Description: Load an image provided by the user.
Tools: **OpenCV (`cv2`)** or **Pillow (PIL)**.
Implementation: A function loads the image from a file path and converts it into a format suitable for the pose estimation model (e.g., a NumPy array).

Task 1.2: Multi-Person 25-Keypoint Extraction

Description: Process the image using a pose estimation model capable of detecting multiple people and outputting a 25-keypoint skeleton for each.
Tools: **OpenPose** is the standard for the BODY_25 model. Its Python API can be used.
Output: A list of detected skeletons. Each skeleton is an array of 25 joint coordinates (e.g., `[25, 3]` for `(x, y, confidence)`).

Task 1.3: Target Selection and Kinematic Conversion

Description: If multiple people are detected, select one skeleton. Convert this skeleton's Cartesian coordinates into the initial joint angles for the humanoid model.
Implementation:

1. **Selection**: A simple heuristic, like choosing the person with the largest bounding box area, can be used to select the main subject.

2. **Conversion**: For the selected skeleton, define vectors between adjacent joints (e.g., `v_hip_knee`, `v_knee_ankle`). Use trigonometric functions like `atan2` to calculate the angle for each required joint.

Output: A single NumPy vector representing the **Initial Pose Vector** (θ_{init}), which will be passed to the simulation environment's `reset` function.

Module 2: Simulation (Physics Environment)

Objective: To create a physically accurate, controllable humanoid model and an environment that can be initialized to a specific pose.

Task 2.1: Humanoid Asset Definition (Rigging)

Description: Define the physical properties of the humanoid agent.
Tools: **URDF** or **MJCF**.
Implementation: An XML-based `.urdf` file specifies links (mass, inertia, collision geometry), joints (axes, limits), and actuators (torque limits).

Task 2.2: Environment Instantiation and API

Description: Create a custom Python environment class that loads the humanoid and manages the simulation state.
Tools: **PyBullet** and **Gymnasium** (`gym`).
Implementation (`HumanoidWalkEnv` **class**):

1. `__init__()`: Initializes PyBullet and loads the URDF. Defines observation and action spaces.
2. `reset(initial_pose=None)`: This function is key. It calls `pybullet.resetSimulation()`, and if an `initial_pose` vector is provided, it iterates through each joint and uses `pybullet.resetJointState()` to set the model to that specific pose. It returns the initial observation.
3. `step(action)`: Takes an action, steps the physics, calculates the reward for walking, checks for termination (falling), and returns the results.

Module 3: Control (DQN Agent)

Objective: To implement the DQN agent that learns a policy for walking, generalized from various starting poses.

Task 3.1: Network Architecture

Description: Define the architecture for the Q-Network.
Tools: **PyTorch (`torch.nn`)** or TensorFlow/Keras.
Implementation: An MLP where:

1. **Input Layer**: Size equals the dimension of the state vector (e.g., joint angles, joint velocities, torso orientation, etc.).
2. **Hidden Layers**: 2-3 fully connected layers (256-512 neurons) with ReLU activations.
3. **Output Layer**: Size equals the total number of discrete actions.

Task 3.2: Action Space Discretization

Description: Discretize the continuous torque values for compatibility with DQN.
Implementation Strategy: The network has one "head" for each joint. For example, for 8 joints with 5 torque bins each (`[-1.0, -0.5, 0, 0.5, 1.0]`), the network's output layer has $8 \times 5 = 40$ neurons. This output is reshaped to `[8, 5]`, and the `argmax` for each joint determines the action.

Task 3.3: Reward Function Engineering

Description: Design a reward function that incentivizes a stable, forward walking gait.
Implementation (Reward Formula):

$$R_t = w_{vel} \cdot r_{vel} + w_{live} \cdot r_{live} - w_{energy} \cdot r_{energy}$$

1. r_{vel}: The primary reward component. A large positive reward for the forward velocity of the torso's center of mass.
2. r_{live}: A small, constant positive reward for each timestep the agent remains above a certain height threshold (i.e., doesn't fall).
3. r_{energy}: A small penalty for the sum of squared torques applied, to encourage efficient, smooth movements.
4. A large negative reward is given if the agent terminates by falling.

Final Module: Training

Description: Implement the DQN training loop with Experience Replay.
Implementation: The training loop is designed to promote generalization.

1. A library of initial poses is pre-generated by running the Perception Module on many different images.
2. At the start of each training episode, a **random initial pose** is selected from this library and passed to the `env.reset(initial_pose=...)` function.
3. The agent then attempts to learn to walk from this randomized starting point.
4. By training on a wide distribution of starting poses, the agent learns a more robust and generalized walking policy.

Best `Moved` wishes!