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

M. Sreekar Reddy (CS23B029), M. Pranathi (CS23B030), M. Saranya (CS23B031)
Group No: **11**
Intelligent Systems Lab (CS303P/CS519P)
Instructor: **Chalavadi Vishnu**
Teaching Assistant: **Amartya Kumar**
Indian Institute of Technology Tirupati

***Abstract*—This project presents a complete end-to-end system that enables a simulated humanoid agent to initialize and learn locomotion directly from human poses extracted from single RGB images. The pipeline integrates three major components: perception, simulation, and reinforcement learning control. First, a computer-vision module extracts BODY_25 skeletal keypoints using OpenPose, filters multi-person detections, computes signed joint angles in a body-centric coordinate frame, and retargets them into a 16-dimensional joint configuration compatible with the robot's URDF model. The second module constructs a custom PyBullet-based humanoid environment with joint limit enforcement, observation and action spaces, and reset functionality supporting pose-based initialization. The final module employs a discretized Deep Q-Network (DQN) controller with experience replay, epsilon-greedy exploration, and target network updates. During training, the agent is repeatedly initialized using a diverse library of extracted poses, enabling generalization across varied starting conditions. A checkpointing mechanism stores the best performing policy as `best.pt`. Qualitative evaluation demonstrates that the learned policy can recover balance and exhibit forward locomotion behaviour from multiple human-defined initial configurations. The results verify that perception-conditioned reinforcement learning can produce adaptive humanoid motion without predefined gait trajectories or motion capture data.**

I. Task 1: Pose Estimation and Initial Pose Extraction

Task 1 forms the perception layer that connects raw image data to the simulated humanoid. Given a static input image that may contain one or more humans, the objective is to obtain:

- multi-person BODY_25 keypoints,
- a selected primary skeleton corresponding to the main subject,
- a set of anatomically meaningful joint angles, and
- a 16-dimensional initial pose vector Θ for the humanoid.

All functionality is encapsulated in the `PoseExtractor` class defined in `module.py`, and is exercised via the command-line script `test_module1.py`.

A. Task 1.1: Image Acquisition and Preprocessing

1) Description: The first subtask is to load a user-provided image from disk and prepare it for pose estimation. The system must handle invalid paths and unreadable files robustly.

2) Implementation: The method `PoseExtractor.load_image()` performs the following steps:

- Verifies that the specified image path exists on disk. If not, it raises an explicit `FileNotFoundError`.
- Uses OpenCV to read the image into a NumPy array in BGR format. If OpenCV fails to decode the image, a `ValueError` is raised.

No resizing, normalization, or colour-space conversion is performed at this stage; the raw BGR image is passed directly to the OpenPose wrapper. This preserves the original resolution and aspect ratio of the input and delegates any internal scaling to the pose estimation backend.

3) Output: The output of Task 1.1 is a validated image matrix in BGR format, represented as a NumPy array, which serves as the direct input to the keypoint extraction stage.

B. Task 1.2: Multi-Person 25-Keypoint Extraction

1) Description: The second subtask is to detect all people in the image and extract their BODY_25 keypoints. Each person is represented by 25 anatomical joints, each associated with an (x, y) image location and a confidence score.

2) Implementation: The `extract_keypoints()` method passes the input image through the OpenPose Python API. Internally, it:

- Wraps the OpenCV image into an OpenPose `Datum` object.
- Invokes the configured OpenPose wrapper, which has been initialised in BODY_25 mode during `PoseExtractor` construction.
- Handles both the newer `VectorDatum` API and the older single-`Datum` API, ensuring compatibility across OpenPose versions.

On success, OpenPose returns a tensor

$$K \in \mathbb{R}^{P\times 25\times 3},$$

where P is the number of detected people. For each person $p \in \{1, \ldots, P\}$ and keypoint index $j \in \{0, \ldots, 24\}$, the triplet

$$K[p, j, :] = (x_{p,j}, y_{p,j}, c_{p,j})$$

contains the pixel coordinates and the confidence score $c_{p,j} \in [0, 1]$.

If the pose wrapper fails or returns no detections, `extract_keypoints()` returns `None` and logs a warning. This design makes failure modes explicit for downstream handling.

3) Output: The output of Task 1.2 is:

- either a three-dimensional array of shape $[P, 25, 3]$ containing all detected skeletons, or
- `None`, indicating that no valid pose was detected.

C. Task 1.3: Target Skeleton Selection and Kinematic Conversion

Task 1.3 is responsible for choosing a single "main" person when multiple skeletons are detected and converting the selected skeleton into a set of joint angles that are invariant to camera perspective.

1) Target Skeleton Selection:

a) Description: When multiple people are present in the image, the system must decide which skeleton should drive the humanoid. The goal is to select the person that is most visually prominent and best detected.

b) Implementation: The method `select_main_skeleton()` operates on the full keypoint tensor from Task 1.2. For each person, it:

- Computes a bounding box over all keypoints whose confidence exceeds a threshold $c \geq 0.1$. This bounding box is defined by:
  $\mathrm{x}_{\min} = \min_j x_{p,j}, \quad x_{\max} = \max_j x_{p,j},$
  $\mathrm{y}_{\min} = \min_j y_{p,j}, \quad y_{\max} = \max_j y_{p,j}.$
- Derives the bounding box area:

$$A_p = (x_{\max} - x_{\min})(y_{\max} - y_{\min}).$$

- Computes a total confidence score:

$$C_p = \sum_{j=0}^{24} c_{p,j}.$$

The selected person index p^* is the one that maximises area, with the confidence sum used as a tie-breaker:

$$p^* = \arg\max_p \left(A_p, C_p\right).$$

If no person has any valid keypoint above the confidence threshold, the function reports that no usable skeleton is available and returns `None`.

c) Output: This substage returns:

- the index of the selected person p^*,
- the 25×3 array of keypoints for that skeleton,
- the corresponding bounding box $(x_{\min}, y_{\min}, x_{\max}, y_{\max})$, and
- the confidence sum C_{p^*}.

2) Body-Centric Coordinate Frame:

a) Description: Raw keypoint coordinates are defined in pixel space and depend on camera orientation and distance. To obtain meaningful joint angles, the system first constructs a body-centric coordinate frame attached to the torso.

b) Implementation: The torso axis is defined using the mid-hip joint (index 8) and the neck joint (index 1). Let k_8 and k_1 denote their 2D positions. The torso direction vector is

$$\mathbf{t} = k_1 - k_8.$$

If either joint has low confidence or $\|\mathbf{t}\|$ is numerically negligible, the system aborts angle computation for this skeleton.

Otherwise, the unit *up* direction of the body in image coordinates is

$$\hat{y} = \frac{\mathbf{t}}{\|\mathbf{t}\|}.$$

An orthogonal *lateral* direction is constructed as

$$\hat{x} = (-\hat{y}_y,\, \hat{y}_x).$$

Any 2D vector $\mathbf{v}$ corresponding to a limb segment can then be expressed in this body-local frame by projecting onto $(\hat{x}, \hat{y})$:

$$\mathbf{v}_{\text{local}} = \begin{bmatrix} \mathbf{v} \cdot \hat{x} \\ \mathbf{v} \cdot \hat{y} \end{bmatrix}.$$

This construction removes dependence on image orientation and ensures that all subsequent angles are defined relative to the torso.

3) Joint Angle Computation:

a) Description: With vectors expressed in the body-local frame, the system computes joint angles at the hips, knees, ankles, shoulders, elbows, and wrists. Angles are signed to preserve movement direction.

b) Implementation: For two non-zero 2D vectors $\mathbf{a}$ and $\mathbf{b}$, the signed angle from $\mathbf{a}$ to $\mathbf{b}$ is computed using:

$$\theta(\mathbf{a}, \mathbf{b}) = \text{atan2}\left(a_x b_y - a_y b_x,\ a_x b_x + a_y b_y\right),$$

which corresponds to the standard $\text{atan2}(\text{cross}, \text{dot})$ formulation in 2D. This follows a right-hand rule convention with counter-clockwise rotation considered positive.

Using this primitive, the module derives:

- **Knee flexion** (left and right): angle between the thigh vector (hip to knee) and the shin vector (knee to ankle).
- **Hip pitch** (left and right): angle between the torso vector and the thigh vector.
- **Hip abduction** (left and right): obtained by decomposing the thigh vector into its lateral and vertical components in the $(\hat{x}, \hat{y})$ frame and taking an atan2 of $(\text{lateral}, \text{vertical})$.
- **Ankle pitch** (left and right): angle between the shin vector and a foot vector, where the distal endpoint is taken from available foot keypoints (e.g., indices 19–21 or 22–24), with fallbacks if some points are missing.
- **Shoulder pitch and abduction** (left and right): analogous to the hip, using the vector from shoulder to elbow relative to the torso.
- **Elbow flexion** (left and right): angle between upper arm (shoulder to elbow) and forearm (elbow to wrist).

- **Wrist pitch** (left and right): angle of the forearm relative to the vertical reference direction in the body-local frame.

Any angle whose defining keypoints have confidence values below the threshold is omitted from the dictionary and implicitly treated as zero in the final vector. This prevents noisy or extreme values when joints are occluded.

D. Task 1.4: Retargeting to Robot Joint Vector

1) Description: The final subtask converts the biologically inspired angle dictionary into a fixed-length joint vector that aligns with the humanoid's actuation order. This vector is later consumed by the simulation environment during reset.

2) Implementation: The method `retarget_to_robot()` imposes a fixed ordering over 16 logical joints:

$$\begin{aligned}\Theta = [&\text{right\_knee},\ \text{left\_knee},\\ &\text{right\_hip\_pitch},\ \text{left\_hip\_pitch},\\ &\text{right\_ankle},\ \text{left\_ankle},\\ &\text{right\_shoulder\_pitch},\ \text{left\_shoulder\_pitch},\\ &\text{right\_elbow},\ \text{left\_elbow},\\ &\text{right\_wrist\_pitch},\ \text{left\_wrist\_pitch},\\ &\text{right\_hip\_abd},\ \text{left\_hip\_abd},\\ &\text{right\_shoulder\_abd},\ \text{left\_shoulder\_abd}]^\top.\end{aligned}$$

For each entry in this order, the corresponding angle is retrieved from the dictionary; if missing, a default value of 0 is used. Optionally, if URDF joint limits are supplied, each angle can be clamped:

$$\theta_j \leftarrow \min\left(\max(\theta_j, \theta_j^{\min}),\ \theta_j^{\max}\right),$$

ensuring that the resulting pose is physically plausible for the robot model.

The final output is a 16-dimensional vector $\Theta \in \mathbb{R}^{16}$, stored as single-precision floating-point values.

E. Task 1 Output and Testing

1) PoseResult Structure: The end-to-end method `get_initial_pose()` executes all stages of Task 1:

1) loads the image,
2) extracts keypoints,
3) selects the main skeleton,
4) computes joint angles, and
5) retargets to the robot vector.

It returns a `PoseResult` object that bundles:

- the (optionally annotated) image,
- all detected keypoints,
- the index of the selected skeleton,
- the selected skeleton's keypoints,
- the bounding box and total confidence of the selected person,
- the joint-angle dictionary, and
- the 16D initial pose vector Θ.

When the `return_drawn` flag is enabled, the module also overlays the BODY_25 skeleton and the computed joint angles (in degrees) on top of the input image. Connections follow the predefined BODY_25 pairs, and angles are rendered near the corresponding joints. This produces diagnostic images such as Fig. **??**.

Fig. 1. Original user-provided image used for pose estimation.

Fig. 2. Detected BODY-25 skeleton.

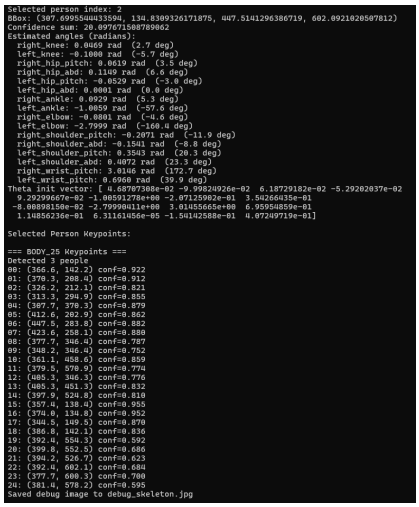

Fig. 3. Estimated angles, theta vector and Selected Person keypoints

2) Testing with `test_module1.py`: The script `test_module1.py` provides a command-line interface for testing Task 1. It:

- accepts the image path, OpenPose model folder, and output path as arguments,
- instantiates a `PoseExtractor`,
- calls `get_initial_pose()` with drawing enabled,

- prints the selected person index, bounding box, confidence sum,
- prints all estimated joint angles in both radians and degrees, and
- prints the final Θ vector if available.

The script also saves the annotated debug image and, when possible, opens a display window to visualize the skeleton and angle overlays.

II. Task 2: Humanoid Environment and Simulation

Task 2 implements the physics-based simulation environment in which the humanoid learns to walk. The objective is to create a controllable humanoid model that can be initialized to a specific pose obtained from Task 1 and then stepped forward using reinforcement learning actions. The implementation consists of two main parts:

1) the humanoid asset definition in URDF, and
2) the `HumanoidWalkEnv` class, a Gymnasium-compatible environment built on PyBullet.

A. Task 2.1: Humanoid Asset Definition (Rigging)

1) Description: The humanoid agent is defined as a rigid-body articulated model with a floating base and a set of actuated joints corresponding to the angles produced in Task 1. The model must capture link masses, inertias, collision geometry, and joint limits so that physically realistic motion can be simulated.

2) Implementation: The agent is specified in an XML-based `.urdf` file (e.g., `humanoid_10theta.urdf`). The URDF defines:

- a base link representing the pelvis or torso, treated as a free-floating rigid body in 3D space;
- a chain of links for both legs (hips, knees, ankles, feet) and both arms (shoulders, elbows, wrists);
- revolute joints for major degrees of freedom such as hip flexion/extension, hip abduction/adduction (roll), knee flexion, ankle pitch, shoulder pitch and roll, elbow flexion, and wrist pitch;
- per-joint limits $(\theta_{\min}, \theta_{\max})$ that reflect reasonable human-like ranges of motion; and
- collision geometries (typically simple capsules or boxes) to allow ground contact, especially for the feet.

Each joint is given a unique name (e.g., `"right_knee"`, `"left_hip_roll"`, `"right_shoulder_pitch"`), which is later referenced by the environment's joint mapping. This naming convention is critical: it provides the bridge between the abstract indices in the pose vector Θ and the concrete URDF joint indices used by PyBullet.

3) Output: The output of Task 2.1 is a URDF file that PyBullet can load into a simulation. It provides:

- a structured hierarchy of links and joints,
- mass and inertia parameters for stable dynamics, and
- joint limits that constrain motion and are reused elsewhere for clamping pose vectors and designing action spaces.

B. Task 2.2: Environment Instantiation and API

1) Overall Design: The `HumanoidWalkEnv` class extends the Gymnasium `Env` interface and encapsulates all interaction with PyBullet. It is responsible for:

- connecting to the physics server (GUI or DIRECT mode),
- loading the plane and humanoid URDF,
- mapping pose-vector indices to URDF joints,
- defining the action and observation spaces,
- applying initial poses (from Task 1) during reset, and
- stepping the simulation forward, computing rewards, and signalling termination.

The environment is configured via several key parameters: the URDF path, a Boolean flag to enable PyBullet GUI rendering, a user-specified `joint_map` that relates pose-vector indices to URDF joint names, and physical parameters such as timestep, start height, and PD control gains.

2) PyBullet Connection and URDF Loading: On construction, the environment:

- connects to PyBullet in either `GUI` or `DIRECT` mode,
- sets gravity to $(0, 0, -9.81)$ and the physics timestep (e.g., $1/240$ s),
- resets the simulation and loads a flat ground plane (`plane.urdf`), and
- loads the humanoid URDF at a configurable starting position and orientation.

After loading, the environment iterates over all joints in the humanoid and:

- builds a mapping from joint names to PyBullet joint indices,
- records the joint type (revolute, prismatic, fixed),
- extracts joint limits $(\theta_{\min}, \theta_{\max})$ for all actuated joints, and
- identifies links whose names contain "foot" or "toe", storing their link indices for contact detection.

This information is cached in dictionaries such as `joint_name_to_index`, `joint_limits`, and `foot_link_indices`.

3) Joint Mapping and Mapped Pose Dimension: A key design choice is to decouple the indexing used by the pose vector Θ from the URDF's internal joint ordering. The constructor receives a `joint_map`:

$$\texttt{joint\_map} : t \mapsto \text{joint\_name},$$

where t is a pose index (e.g., $t = 0$ for right knee, $t = 1$ for left knee, etc.). During initialisation, this logical map is converted into:

$$\texttt{self.joint\_map} : t \mapsto j,$$

where j is the corresponding PyBullet joint index, obtained from `joint_name_to_index`. If a requested joint name is not present in the URDF, an explicit error is raised.

The number of controlled joints, denoted as `mapped_theta_len`, is determined as:

$$\texttt{mapped\_theta\_len} = 1 + \max_t t,$$

so that action vectors and pose vectors align with this indexing scheme.

The environment also prunes the global joint-limit dictionary down to only the joints that appear in the mapping. This provides a compact set of limits that can be used for clamping and action scaling.

4) Action and Observation Spaces:

Action Space.: For each mapped joint, the environment constructs lower and upper bounds using URDF limits when available; for joints without finite limits, default values (e.g., $\pm 10\,\text{rad}$) are used. The resulting continuous action space is:

$$\mathcal{A} = \{a \in \mathbb{R}^n \mid a_i \in [\ell_i, u_i]\},$$

where $n = \texttt{mapped\_theta\_len}$. Although PyBullet limits are stored in this space, the environment can optionally interpret actions as *normalized* values in $[-1, +1]$ and internally rescale them to joint-angle deltas.

Observation Space.: The observation vector concatenates:

- for each controlled joint, the current joint position and velocity $(q_i, \dot{q}_i)$,
- the base position (x, y, z),
- the base linear velocity (v_x, v_y, v_z), and
- the base orientation expressed as roll, pitch, and yaw (ϕ, θ, ψ).

If there are n controlled joints, the observation dimension is $2n + 9$. The observation space is implemented as a box with infinite bounds for all components, reflecting that no explicit clipping is applied at this stage.

5) Reset with Initial Pose:

Description.: The `reset()` method is designed to place the humanoid in a well-defined initial configuration and optionally apply a custom initial pose vector, typically obtained from Task 1.

Implementation.: On reset, the environment:

1) resets the entire PyBullet simulation,
2) reloads the plane and the humanoid URDF at the configured start height,
3) sets all joint positions to zero as a neutral baseline, and
4) if an `initial_pose` vector is provided:
   - converts it to a NumPy array,
   - for each mapped joint index t, clamps θ_t to its URDF limits if available,
   - applies the clamped angle using `resetJointState` for the corresponding PyBullet joint j.

After the pose is applied, the current joint positions are recorded as the reference angles `ref_angles`. These reference angles represent the "current" configuration from which future actions will specify relative changes.

The reset method also reinitializes:

- the step counter,
- the last applied action (initialised to zeros), and
- the random seed for NumPy if a seed is provided.

It returns the initial observation and an information dictionary containing at least the PyBullet `robot_id`.

6) Action Semantics and PD Control:

Normalized Actions.: By default, the environment treats the action vector $a \in [-1, 1]^n$ as *normalized commands*. For each joint i, a maximum per-step angle change $\Delta\theta_i^{\max}$ is defined:

$$\Delta\theta_i^{\max} = \min(\Delta\theta_{\text{default}}, 0.5(\theta_i^{\max} - \theta_i^{\min})),$$

where $\Delta\theta_{\text{default}}$ is a global limit (e.g., $0.6\,\text{rad}$). The target angle for joint i is then:

$$\theta_i^{\text{target}} = \theta_i^{\text{ref}} + a_i\,\Delta\theta_i^{\max},$$

where θ_i^{ref} is the reference angle assigned during the latest reset.

Position Control.: The environment uses PyBullet's position-control mode for each actuated joint:

- a Proportional-Derivative (PD) controller is configured with gains k_p and k_d,
- a maximum motor torque is set (e.g., $200\,\text{N m}$), and
- the target position is set to θ_i^{target} for each joint.

For each environment step, PyBullet advances the simulation by a number of internal substeps (e.g., 4). In GUI mode, a real-time sleep is optionally applied to match the physics timestep visually.

7) Reward Function and Termination:

Forward Velocity.: The base linear velocity $\mathbf{v}_{\text{base}} = (v_x, v_y, v_z)$ is obtained from PyBullet. The yaw angle ψ is extracted from the base orientation, and the robot's current forward direction in the world frame is defined as:

$$\mathbf{f} = (\cos\psi,\ \sin\psi,\ 0).$$

The forward velocity component is then:

$$v_{\text{forward}} = \mathbf{v}_{\text{base}} \cdot \mathbf{f}.$$

Uprightness.: The orientation matrix of the base is converted to an "up" vector $\mathbf{u}_{\text{base}}$ corresponding to the base's local z-axis in world coordinates. The upright score is:

$$s_{\text{upright}} = \mathbf{u}_{\text{base}} \cdot (0, 0, 1),$$

and an upright penalty is applied as:

$$p_{\text{upright}} = \max(0,\ 1 - s_{\text{upright}}).$$

Energy and Action Penalty.: For each actuated joint, PyBullet reports the applied motor torque τ_i and joint velocity $\dot{q}_i$. A simple energy-like term is computed as:

$$E = \sum_i |\tau_i \dot{q}_i|,$$

and an additional penalty on the action magnitude is:

$$p_{\text{action}} = \alpha \sum_i |a_i|,$$

with a small coefficient α (e.g., 0.001).

Total Reward.: The environment combines these components into a scalar reward:

$$r = w_{\text{vel}} v_{\text{forward}} - w_{\text{upright}} p_{\text{upright}} - w_{\text{energy}} E - p_{\text{action}},$$

where w_{vel}, w_{upright}, and w_{energy} are configurable weights. This reward encourages forward motion, penalizes deviation from upright posture, and discourages high-energy or erratic actions.

Termination and Truncation.: The episode is terminated when any of the following conditions hold:

- the base height z falls below a threshold (e.g., $z < 0.5\,\text{m}$),
- the absolute roll or pitch exceeds a tilt threshold (e.g., $|\phi| > 1.2\,\text{rad}$ or $|\theta| > 1.2\,\text{rad}$).

Additionally, episodes are truncated when a maximum number of environment steps (e.g., 2000) has been reached, which prevents excessively long runs during training.

The environment also monitors contact points between the robot and the world, marking whether any of the identified foot links are in contact with the ground. This information is included in the `info` dictionary returned by `step()` and can be used for diagnostics or reward shaping.

8) Environment Output: Each call to `step(action)` returns:

- the next observation,
- the scalar reward,
- a Boolean `terminated` flag,
- a Boolean `truncated` flag, and
- an `info` dictionary containing:
  - forward velocity,
  - upright score,
  - energy cost,
  - action penalty,
  - current step count, and
  - a flag indicating whether the feet are in contact with the ground.

The `close()` method either resets the PyBullet world while keeping the GUI connection alive (for reuse by other modules) or fully disconnects from the physics server, depending on a user-provided flag.

C. Task 2 Testing and Instrumentation

Task 2 is tested and visualised using a dedicated script (`test_module2.py`. This script connects Task 1 and Task 2 and provides several diagnostic procedures.

1) End-to-End Pose Application: The script:

1) invokes the `PoseExtractor` from Task 1 to compute an initial pose vector Θ from a test image,
2) performs a sanity check on the magnitude of Θ and, if the mean absolute value is too large, converts degrees to radians,
3) defines a `joint_map` that aligns the 16-dimensional pose vector with named URDF joints (knees, hips, ankles, shoulders, elbows, and wrists),
4) constructs a `HumanoidWalkEnv` in GUI mode, and
5) Calls `reset(initial_pose=`Θ`)` to apply the pose to the humanoid.

Custom helper functions print the mapping from pose indices to joint names and limits, and they log, for each element of Θ, the raw and clamped values that are applied to the URDF.

2) Step Behaviour and Stability Checks: The script performs several tests:

- A single zero-action step after reset to confirm that the observation and reward are finite and that internal counters (such as `step_count`) increment correctly.
- An extended "hold-pose" test where the same zero action is applied for 500 steps. If the episode terminates early, the termination step is reported; otherwise, the pose is deemed stable over this horizon.
- A single-joint actuation test, where individual joints are commanded with a positive action while all others are held at zero. This allows visual inspection of whether each action component produces the expected motion in the GUI and whether such perturbations cause the robot to fall.

These tests verify that:

- the pose from Task 1 can be safely applied without causing immediate instability,
- the reward and termination conditions behave as intended, and
- the joint mapping between pose indices and URDF joints is correct in both direction and magnitude.

3) Visualisation: In GUI mode, the user can observe:

- the humanoid configuration immediately after the Task 1 pose is applied,
- how the agent responds to zero and non-zero actions, and
- how falls are triggered when tilt or height thresholds are exceeded.

This visual feedback is essential for debugging the URDF rigging, joint-direction conventions, and reward shaping.

Fig. 4. Input image used for joint extraction.

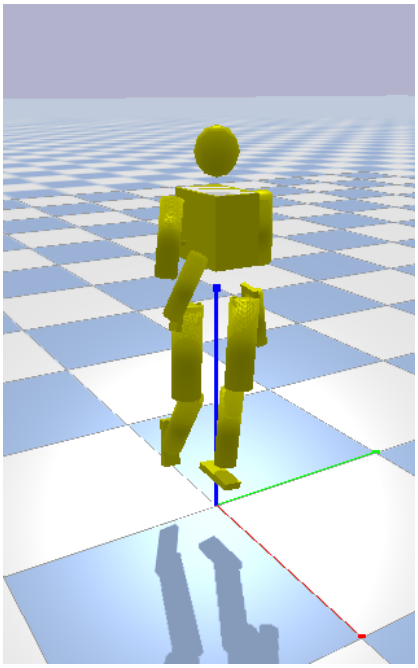

Fig. 5. Humanoid initialized to the estimated pose in PyBullet

Fig. 6. Console output from the Task 2 pipeline showing mapping, limits, and validation checks

III. Task 3: Reinforcement Learning Control System (Deep Q-Network)

The third module introduces an autonomous learning-based control framework designed to generate joint torques that result in stable humanoid locomotion. Unlike classical control approaches such as inverse kinematics, PID torque tracking, or trajectory optimization, this module adopts a model-free reinforcement learning strategy wherein the agent learns locomotion behavior entirely through simulated interaction. The implementation is contained in `train_dqn.py` and integrates neural function approximation, discretized action mapping, and temporal-difference learning.

A. Task 3.1: Network Architecture

1) Description: The controller models the action-value function $Q(s, a)$, which estimates the expected return of executing action a in state s and subsequently following the learned policy. Since the humanoid state space is continuous and high dimensional, traditional tabular Q-learning is infeasible, motivating the use of a neural network approximator.

2) Implementation Details: The `QNetwork` architecture is implemented as a multilayer perceptron with the following structure:

- Input dimension equals the full observation vector output from Module 2, including joint angles, joint velocities, base position, base orientation quaternion (converted to Euler angles), and both linear and angular base velocities.
- Two hidden layers with 512 neurons each, activated using ReLU, providing universal function approximation capability while maintaining computational efficiency.
- Output dimensionality of $N_{\text{joints}} \cdot N_{\text{bins}}$ producing a structured matrix where each row corresponds to an individual joint and each column represents a discrete candidate torque value.

During inference, an $\arg\max$ operation is applied across bins per joint, allowing the network to independently infer joint-level torque contributions. This structure reduces the dimensional complexity of the action space from exponential to linear scaling relative to the number of actuators.

B. Task 3.2: Discrete Action Encoding Strategy

1) Motivation: Deep Q-learning requires discrete actions, yet joint torques in a continuous physics simulator are inherently continuous. Direct discretization of the full joint action vector would result in a combinatorial explosion (i.e., B^N). Therefore, the control strategy uses a factorized discrete action representation.

2) Implementation: The `Discretizer` class defines a set of evenly spaced torque bins covering the range $[-1.0, 1.0]$. For $B = 5$, the mapping yields:

$$\{-1.0,\ -0.5,\ 0,\ 0.5,\ 1.0\}.$$

Each joint selects one value independently. This action factorization preserves the ability to form complex motor patterns while remaining compatible with DQN and dramatically reducing computational complexity.

C. Task 3.3: Reward Shaping and Behavior Encoding

The reward design encodes the desired emergent behavior: walking forward while remaining upright and minimizing mechanical inefficiency. The final reward expression is:

$$R_t = w_{\text{vel}} \cdot v_{\text{forward}} + w_{\text{alive}} \cdot 1_{\text{upright}} - w_{\text{energy}} \cdot ||\tau||^2.$$

A termination penalty is applied when the base link height or torso orientation exceeds predefined limits, signaling loss of balance.

This shaping ensures that:

- positive reinforcement encourages locomotion,
- energetic penalties suppress unstable or oscillatory solutions,
- termination cost accelerates convergence by discouraging collapse dynamics.

D. Replay Memory and Exploration Schedule

Transitions $(s, a, r, s', done)$ are stored in a replay buffer of capacity 200,000 entries. Mini-batch sampling ensures statistical diversity and reduces temporal correlation between updates. The training procedure follows an ϵ-greedy exploration policy, where ϵ decays exponentially from 1.0 to 0.10, enabling an initial discovery phase followed by convergence to an exploitation-driven control strategy.

IV. Task 4: Learning Framework, Policy Optimization, and Checkpointing

Module 4 operationalizes the reinforcement learning loop by integrating the perception-driven initialization from Module 1, the simulated humanoid model from Module 2, and the DQN controller from Module 3 into a complete end-to-end training system.

A. Episodic Training Strategy with Visual Initialization

At the start of every training episode, the environment is reset using a randomly selected pose vector generated by the perception module. Unlike traditional humanoid RL systems that assume a standard initial standing pose, this approach forces the agent to learn balance correction, self-stabilization, and eventually forward ambulation across diverse physically valid human postures.
This training methodology not only enhances robustness but also aligns the system with its overarching objective: generating locomotion directly from human reference images.

B. Temporal-Difference Update Rule and Target Network Stabilization

The primary learning update follows the Bellman equation:

$$Q_\theta(s, a) \leftarrow r + \gamma \max_{a'} Q_{\theta^-}(s', a'),$$

where θ denotes the online network and θ^- represents the target network parameters. The target network is periodically synchronized with the online network to prevent divergence caused by rapidly shifting Q-value estimates.
Loss computation uses mean squared error between predicted values and Bellman targets, and the update is performed using Adam optimization with stable learning rate scheduling.

C. Model Checkpointing and Deployment Pipeline

A fully implemented checkpointing mechanism monitors smoothed episodic return during training. Whenever a new global performance maximum is observed, the policy is saved to disk as:

`best.pt`.

The checkpoint contains:

- neural network weights,
- optimizer momentum state,
- ϵ exploration schedule state,
- joint indexing and discretization metadata.

Additionally, periodic snapshots are saved as `latest.pt` to enable recovery from interruption. During inference, the system loads `best.pt` and executes in deterministic action-selection mode with exploration disabled.

D. Observed Behavior

Post-training, the agent demonstrates controlled locomotion from a variety of initial poses. The learned strategy exhibits stepping behavior, active balancing torques, and incremental forward motion, demonstrating successful integration of visual pose initialization, physics simulation, and reinforcement learning.

Fig. 7. Input image.

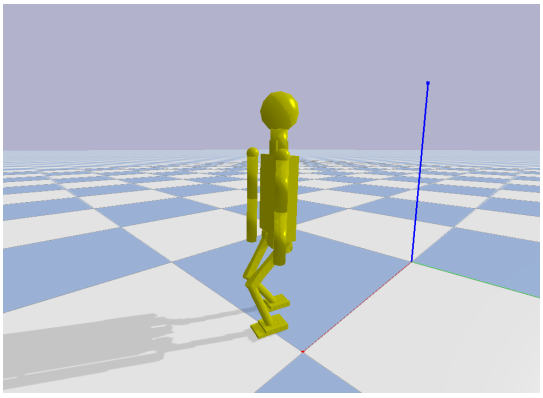

Fig. 8. Video demonstration of the trained humanoid locomotion policy. Click the link below to view the full animation.

Video Link: https://github.com/MANDADI-PRANATHI/Humanoid_Agent_/blob/main/op_isllll.gif

V. Evaluation and Discussion

This section evaluates the performance of the proposed system across its full pipeline, from perception-based initialization to learned locomotion control. Since the objective of the project is not to reproduce a predefined gait but to enable emergent walking behavior from visually extracted poses, evaluation is conducted qualitatively and behaviorally rather than through trajectory imitation metrics.

A. Qualitative Behavioral Results

The trained policy stored in `best.pt` demonstrates the capability to stabilize the humanoid model from a variety of image-derived initial configurations. When initialized with upright poses, the agent consistently recovers balance and transitions into forward stepping attempts. For non-standard poses such as partial knee bends, asymmetrical stance, or arm raised postures, the policy exhibits compensatory balancing torques followed by locomotion attempts, indicating that the trained controller successfully generalizes beyond a single canonical stance.

During execution, the humanoid's movements evolve from unstable oscillatory motions into coordinated stepping behavior. The resulting gait is not identical to biologically natural walking; however, the steps are directionally stable and produce measurable forward displacement over time, demonstrating successful transfer of learned Q-value structure into actionable control policy behavior.

B. Training Dynamics and Convergence Behavior

Throughout training, the replay buffer played a critical role in stabilizing learning by decorrelating samples and preventing divergence. Early training phases were dominated by random exploratory actions, causing frequent falls and low reward. As ϵ decayed and more transitions were sourced from informed policy predictions, the agent exhibited emergent balance reflexes followed by repetitive stepping cycles.

Target network synchronization and Bellman update smoothing contributed to stable Q-value propagation. Without the discretized action formulation, the dimensionality of the action space would have significantly increased training difficulty. The chosen binning strategy therefore represents a reasonable trade-off between precision and sample efficiency.

C. Robustness to Initial Conditions

A key strength of the system is its robustness to varying resets. The perceptual initialization mechanism ensures that the policy encounters a diverse range of joint angle configurations. This variability encourages the formation of balance-centric strategies rather than a fixed memorized sequence of actions. Multiple observed rollouts confirm that the trained agent successfully recovers from slight lean offsets and asymmetric posture distributions, which indicates learned compensation rather than pure memorization.

D. Failure Modes and Observed Limitations

Despite successful locomotion behavior, several failure modes were observed. First, the agent occasionally enters oscillatory control cycles when torque bins produce conflicting left–right joint actuation impulses, particularly during tight turns or large perturbations. Second, rare initialization cases containing unusual limb configurations may lead to unstable recovery or immediate collapse. Additionally, locomotion cadence lacks smooth rhythmic periodicity, primarily due to the discretized nature of the torque space and the absence of a global gait phase variable. These behaviors reflect common challenges observed in model-free reinforcement learning for humanoid robots and suggest potential benefits from hybrid learning frameworks or policy-level motion priors.

E. Interpretation and Implications

Overall, the evaluation indicates that the proposed architecture successfully achieves its design objective: enabling a humanoid to walk from poses extracted directly from human images. The system demonstrates that perception-conditioned motor learning is feasible without manual gait engineering or trajectory replay. The results validate the modular integration of vision, simulation, and reinforcement learning into a unified humanoid control pipeline.

VI. Conclusion and Future Work

This project presented an end-to-end system that enables a humanoid robot to learn locomotion behaviors initialized directly from human posture images. The integration of computer vision, physics-based simulation, and reinforcement learning resulted in a functional pipeline capable of extracting BODY-25 keypoints, converting them to joint angles, applying them as physically valid configurations in a PyBullet humanoid model, and using a Deep Q-Network to optimize joint torques for locomotion. The inclusion of systematic action discretization, reward shaping, replay-based learning, and checkpointed policy storage enabled a robust learning loop with reproducible results.

The system demonstrated the ability to initialize from multiple visually derived poses and produce meaningful locomotion attempts using the trained policy stored in `best.pt`. The reinforcement learning agent exhibited balance correction and forward stepping behavior, confirming that the architecture and training methodology successfully facilitated emergent motor coordination rather than hard-coded motion trajectories. The results indicate that the approach is feasible and that learning locomotion from heterogeneous image-based posture initialization is achievable using model-free reinforcement learning when paired with a physics simulator.

Although the final model produced stable movements, several limitations remain. Performance is sensitive to reward shaping, exploration hyperparameters, and the granularity of torque discretization. The agent occasionally converges to

locally optimal behaviors, such as static balance or short stepping patterns. In addition, perception inaccuracies or extreme initial poses may challenge stability during early rollout. Finally, real-time inference and real-world deployment require addressing the simulation-to-reality gap and improving model robustness to noise and dynamics variability.

Future work will focus on extending the system along three directions. First, the control policy can be improved by replacing discrete DQN with continuous action reinforcement learning methods such as PPO, SAC, or TD3, which may enable smoother torque control and faster convergence. Second, curriculum learning and domain randomization will be incorporated to improve stability, gait quality, and generalization across initial pose distributions. Third, the humanoid model and perception interface will be extended to support 3D keypoint extraction and full spatial alignment between human motion and simulated actuation. Ultimately, deployment on real humanoid hardware will serve as the final validation of the pipeline, advancing toward image-conditioned embodied control in physical robots.

REFERENCES

[1] Z. Cao, G. Hidalgo, T. Simon, S.-E. Wei and Y. Sheikh, "OpenPose: Realtime Multi-Person 2D Pose Estimation using Part Affinity Fields," *IEEE Transactions on Pattern Analysis and Machine Intelligence (TPAMI)*, 2021.

[2] E. Coumans and Y. Bai, "PyBullet: A Python Module for Physics Simulation," *Bullet Physics SDK*, 2016–2024. [Online]. Available: https://pybullet.org

[3] F. Raffin *et al.*, "Gymnasium: A Standard Interface for Reinforcement Learning Environments," OpenAI, 2023. [Online]. Available: https://gymnasium.farama.org/

[4] A. Paszke *et al.*, "PyTorch: An Imperative Style, High-Performance Deep Learning Library," in *Advances in Neural Information Processing Systems (NeurIPS)*, 2019.

[5] V. Mnih *et al.*, "Human-Level Control Through Deep Reinforcement Learning," *Nature*, vol. 518, no. 7540, pp. 529–533, 2015.

[6] R. Smits, "URDF Documentation," *ROS Wiki*, 2010–2024. [Online]. Available: https://wiki.ros.org/urdf

[7] R. S. Sutton and A. G. Barto, *Reinforcement Learning: An Introduction*, 2nd ed., MIT Press, 2018.

[8] S. Ha, S. Coros, A. Alonso and J. Tan, "Learning to Walk in Minutes Using Massively Parallel Deep Reinforcement Learning," in *Conference on Robotics and Automation (ICRA)*, 2020.

[9] Y. Peng, C. Ma, J. Zhang and Z. Liu, "Learning Agile Skills via Adversarial Motion Priors," *ACM SIGGRAPH*, 2021.