CartPole Control with PPO

Architecture:

ActorCritic Network:
├── Shared features (64 hidden units, Tanh activation)
├── Actor head → Policy (mean + log std)
└── Critic head → Value function

Algorithm components:

1. **Policy network**: Outputs continuous actions (forces)
2. **Value network**: Estimates state values for advantage estimation
3. **GAE (Generalized Advantage Estimation)**: Balances bias vs. variance
4. **Clipped objective**: Prevents too-large policy updates (ε=0.2)
5. **Mini-batch updates**: 4 epochs, batch size 2048

Hyperparameters:

- Learning rate: 3e-4
- Discount factor (γ): 0.99
- GAE lambda (λ): 0.95
- Clip epsilon: 0.2
- Value function coefficient: 0.5
- Entropy coefficient: 0.01

Why PPO?

- Sample efficient for simple control tasks
- Stable training (clipped updates)
- Well-suited for continuous actions
- Proven performance on CartPole

3. Training Script (train_ppo.py)

Purpose: Orchestrates the entire training process

Workflow:

1. Create 4096 parallel environments
2. Initialize PPO agent with ActorCritic network
3. For each iteration:
   a. Collect 32 steps of experience from all environments
   b. Compute advantages using GAE
   c. Update policy with PPO (4 epochs, mini-batches)
   d. Log metrics to TensorBoard
   e. Save best model if performance improves
4. Save final model

Training loop details:

- **Rollout collection**: Gather state-action-reward tuples
- **Advantage computation**: Calculate GAE
- **Policy update**: Multiple epochs over collected data
- **Progress tracking**: Loss metrics, rewards, entropy

TensorBoard logging:

- Loss curves (policy, value, entropy)
- Reward statistics (mean, std, min, max)
- Running averages (10-iteration, 100-iteration)
- Value function and advantage statistics

Expected training time: 5-10 minutes with 4096 environments

4. Testing Script (test_ppo.py)

Purpose: Evaluate trained policy and generate visualizations

Process:

1. Load trained model
2. Set policy to eval mode (deterministic)
3. For each test episode:
   a. Reset environment
   b. Run policy until termination or max steps
   c. Log all states, actions, rewards
4. Save rollout data to CSV
5. Generate summary statistics

Output data (ppo_rollouts.csv):

- Time series of all state variables
- Control actions applied
- Rewards received
- Episode metadata

Key difference from training:

- **Deterministic policy**: Uses mean action (no sampling)
- **Single environment**: Easier to analyze
- **Full logging**: Records every timestep for analysis

5. Visualization (plot_ppo_results.py)

Purpose: Create publication-quality plots

Generates two figures:

Figure 1: Trajectory plots (6 subplots)

1. Pole angle trajectories (degrees vs. time)
2. Cart position trajectories (meters vs. time)
3. Control actions (force vs. time)
4. Cart velocity (m/s vs. time)
5. Pole angular velocity (rad/s vs. time)
6. Cumulative rewards

Figure 2: Performance summary (4 subplots)

1. Bar chart of episode rewards
2. Bar chart of episode lengths
3. Final pole angles per episode
4. Text summary with statistics

Colors: Uses Viridis colormap for distinguishable episodes

6. Main Demo Script (provided code)

Purpose: Simple demonstration of the environment with random actions

What it does:

1. Creates CartPole environment with custom parameters
2. Runs simulation loop with random actions
3. Resets every 300 steps
4. Prints state of first environment
5. Displays rewards

Use case:

- Testing environment setup
- Verifying physics parameters
- Debugging before training

Complete Workflow

Training workflow:

```
# Step 1: Train PPO policy (5-10 minutes with 4096 envs)
~/IsaacLab/isaaclab.sh -p train_ppo.py --num_envs 4096 --headless

# Step 2: Monitor training (in separate terminal)
tensorboard --logdir=runs
```

```
# Step 3: Wait for convergence (~500 iterations)
# Model automatically saves to ppo_cartpole_best.pth
```

Testing workflow:

```
# Step 1: Test trained policy and save rollouts
~/IsaacLab/isaaclab.sh -p test_ppo.py \
  --model ppo_cartpole_best.pth \
  --num_envs 1 \
  --num_episodes 5 \
  --headless

# Step 2: Generate plots
python plot_ppo_results.py

# Output: ppo_evaluation_rollouts.png, ppo_performance_summary.png
```

Key Design Decisions

1. Why parallel environments?

- **Speed**: 4096 envs
- **Diversity**: Different initial conditions improve exploration
- **Efficiency**: GPU parallelization through Isaac Lab

2. Why PPO over other algorithms?

- **Stability**: Clipped updates prevent destructive changes
- **Sample efficiency**: Good for simple control tasks
- **Proven**: Standard baseline for continuous control
- **Simplicity**: Fewer hyperparameters than SAC/TD3

Technical Implementation Details

State observation:

- **Format**: Dictionary with "policy" key
- **Shape**: (num_envs, 4) tensor
- **Contents**: [x, v_x, θ, vθ]
- **Device**: CUDA for GPU environments

Action space:

- **Type**: Continuous
- **Dimension**: 1 (force)
- **Distribution**: Gaussian with learned mean/std
- **Bounds**: Typically normalized [-1, 1] then scaled

Physics integration:

- **Method**: Semi-implicit Euler (PhysX)
- **Timestep**: 0.02s (50 Hz)
- **Solver**: Articulation solver with joint constraints

Termination conditions:

- Pole angle exceeds threshold (too far from vertical)
- Cart position exceeds bounds (fell off track)
- Maximum episode length reached