Domain Randomization & Fault Injection Implementation

🎲 Where Domain Randomization is Defined

Domain Randomization (DR) is implemented through a **wrapper pattern** around the base environment. Here's the complete structure:

1. **Configuration** (configs/experiments/ppo_dr.yaml)

```yaml
domain_randomization:
  enabled: true
  curriculum:
    enabled: true
    phases:
      - name: "warmup"
        epochs: [0, 200]
        fault_prob: 0.0
        sensor_noise: 0.01
      - name: "isolated"
        epochs: [200, 600]
        fault_prob: 0.2
        sensor_noise: 0.05
        max_failed_joints: 1
      - name: "full"
        epochs: [600, -1]
        fault_prob: 0.4
        sensor_noise: 0.1
        max_failed_joints: 3

faults:
  actuator_dropout:
    enabled: true
    mode: "lock"  # "lock" or "zero_torque"
    affected_joints: [0, 1, 2, 3, 4, 5, 6, 7]  # All 8 joints
  sensor_noise:
    enabled: true
    position_std: 0.05
    velocity_std: 0.1
    orientation_std: 0.02
```

2. Implementation (src/envs/fault_injection.py)

```python
```

```python
import numpy as np
import gymnasium as gym
from gymnasium import spaces

class FaultInjectionWrapper(gym.Wrapper):
    """
    Wrapper that adds actuator faults and sensor noise to RealAnt
    """

    def __init__(self, env, fault_config, curriculum_phase=None):
        super().__init__(env)
        self.fault_config = fault_config
        self.curriculum_phase = curriculum_phase

        # Actuator fault parameters
        self.fault_prob = fault_config.get('fault_prob', 0.0)
        self.max_failed_joints = fault_config.get('max_failed_joints', 1)
        self.fault_mode = fault_config.get('mode', 'lock')

        # Sensor noise parameters
        self.sensor_noise_config = fault_config.get('sensor_noise', {})

        # State tracking
        self.failed_joints = []
        self.locked_positions = {}
        self.episode_count = 0

    def reset(self, **kwargs):
        obs, info = self.env.reset(**kwargs)

        # Decide if faults occur this episode
        if np.random.random() < self.fault_prob:
            self._inject_actuator_faults()
        else:
            self.failed_joints = []
            self.locked_positions = {}

        # Add sensor noise to initial observation
        obs = self._add_sensor_noise(obs)

        # Add fault info to observation
        info['failed_joints'] = self.failed_joints.copy()
```

```python
        return obs, info

    def _inject_actuator_faults(self):
        """Randomly select joints to fail"""
        num_joints = self.env.action_space.shape[0]  # 8 for RealAnt

        # How many joints will fail?
        num_failures = np.random.randint(1, min(self.max_failed_joints + 1, num_joints))

        # Which joints fail?
        self.failed_joints = np.random.choice(
            num_joints,
            size=num_failures,
            replace=False
        ).tolist()

        # Lock positions at current state
        if self.fault_mode == 'lock':
            # Get current joint positions from the simulator
            joint_positions = self._get_joint_positions()
            for joint_idx in self.failed_joints:
                self.locked_positions[joint_idx] = joint_positions[joint_idx]

    def step(self, action):
        # Modify actions for failed joints
        modified_action = action.copy()

        for joint_idx in self.failed_joints:
            if self.fault_mode == 'lock':
                # For locked joints, we need to apply control to maintain position
                # This requires access to current position and PD control
                current_pos = self._get_joint_positions()[joint_idx]
                target_pos = self.locked_positions[joint_idx]

                # Simple P controller to maintain position
                kp = 100.0  # Tunable
                modified_action[joint_idx] = kp * (target_pos - current_pos)

            elif self.fault_mode == 'zero_torque':
                # Simply disable the motor
                modified_action[joint_idx] = 0.0

        # Execute step with modified action
        obs, reward, terminated, truncated, info = self.env.step(modified_action)
```

```python
        # Add sensor noise
        obs = self._add_sensor_noise(obs)

        # Add fault information
        info['failed_joints'] = self.failed_joints
        info['original_action'] = action
        info['modified_action'] = modified_action

        return obs, reward, terminated, truncated, info

    def _add_sensor_noise(self, obs):
        """Add Gaussian noise to observations"""
        if not self.sensor_noise_config.get('enabled', False):
            return obs

        # RealAnt observation structure:
        # [0:8] - joint positions
        # [8:16] - joint velocities
        # [16:20] - orientation quaternion
        # [20:28] - contact sensors

        noisy_obs = obs.copy()

        # Joint position noise
        if 'position_std' in self.sensor_noise_config:
            noise = np.random.normal(0, self.sensor_noise_config['position_std'], 8)
            noisy_obs[0:8] += noise

        # Joint velocity noise
        if 'velocity_std' in self.sensor_noise_config:
            noise = np.random.normal(0, self.sensor_noise_config['velocity_std'], 8)
            noisy_obs[8:16] += noise

        # Orientation noise (careful with quaternion normalization)
        if 'orientation_std' in self.sensor_noise_config:
            noise = np.random.normal(0, self.sensor_noise_config['orientation_std'], 4)
            noisy_obs[16:20] += noise
            # Renormalize quaternion
            noisy_obs[16:20] /= np.linalg.norm(noisy_obs[16:20])

        return noisy_obs

    def _get_joint_positions(self):
```

```python
    """Get current joint positions from MuJoCo"""
    # This depends on your specific MuJoCo environment
    # For RealAnt, it's typically:
    return self.env.unwrapped.data.qpos[7:15]  # Skip base position/orientation
```

3. **Curriculum Learning** (src/envs/curriculum.py)

```python
```

```python
class CurriculumManager:
    """
    Manages the progression through training phases
    """

    def __init__(self, curriculum_config):
        self.phases = curriculum_config['phases']
        self.current_phase_idx = 0
        self.epoch = 0

    def get_current_config(self):
        """Get fault config for current training phase"""
        phase = self.phases[self.current_phase_idx]

        return {
            'fault_prob': phase['fault_prob'],
            'max_failed_joints': phase.get('max_failed_joints', 1),
            'sensor_noise': {
                'enabled': True,
                'position_std': phase['sensor_noise'],
                'velocity_std': phase['sensor_noise'] * 2,  # Velocities are noisier
                'orientation_std': phase['sensor_noise'] * 0.5
            }
        }

    def update(self, epoch):
        """Progress to next phase if needed"""
        self.epoch = epoch

        # Check if we should advance to next phase
        for i, phase in enumerate(self.phases):
            start_epoch, end_epoch = phase['epochs']
            if end_epoch == -1:  # Last phase
                end_epoch = float('inf')

            if start_epoch <= epoch < end_epoch:
                if i != self.current_phase_idx:
                    print(f"Advancing to phase: {phase['name']}")
                    self.current_phase_idx = i
                break
```

4. Integration in Training (src/train.py)

python

```python
def create_env(config):
    """Create environment with appropriate wrappers"""

    # Base environment
    env = gym.make('RealAnt-v0')

    # Add domain randomization if enabled
    if config.domain_randomization.enabled:
        if config.domain_randomization.curriculum.enabled:
            # Create curriculum manager
            curriculum = CurriculumManager(config.domain_randomization.curriculum)
            fault_config = curriculum.get_current_config()
        else:
            # Fixed fault configuration
            fault_config = config.faults

        # Wrap environment
        env = FaultInjectionWrapper(env, fault_config)

    # Add other wrappers (normalization, etc.)
    env = gym.wrappers.NormalizeObservation(env)
    env = gym.wrappers.NormalizeReward(env)

    return env
```

🔧 How Fault Injection Works

Method 1: Joint Locking (Realistic)

python

```python
```

```
# When a joint "breaks", we lock it at current position
# This simulates a seized bearing or mechanical failure

if joint_failed:
    # Get current position when failure occurs
    locked_position = current_joint_angle

    # During each step, apply PD control to maintain position
    error = locked_position - current_joint_angle
    torque = Kp * error + Kd * joint_velocity
    action[joint_idx] = torque  # Override user action
```

Method 2: Zero Torque (Simple)

```python
# Simply set torque to zero - motor provides no force
if joint_failed:
    action[joint_idx] = 0.0
```

Method 3: Reduced Torque (Partial Failure)

```python
# Reduce maximum torque - simulates weak motor
if joint_degraded:
    action[joint_idx] *= 0.3  # 30% strength
```

🎯 MuJoCo-Specific Implementation

For RealAnt in MuJoCo, here's how to directly interact with joints:

```python
```

```python
class RealAntFaultWrapper(gym.Wrapper):
    def __init__(self, env):
        super().__init__(env)

        # Access MuJoCo model and data
        self.model = env.unwrapped.model
        self.data = env.unwrapped.data

        # Joint indices for RealAnt
        # Assuming joints are named: 'hip_1', 'ankle_1', 'hip_2', etc.
        self.joint_names = [
            'hip_1', 'ankle_1', 'hip_2', 'ankle_2',
            'hip_3', 'ankle_3', 'hip_4', 'ankle_4'
        ]

        # Get joint IDs from names
        self.joint_ids = [
            self.model.joint_name2id(name)
            for name in self.joint_names
        ]

    def _lock_joint(self, joint_idx, position):
        """Lock a joint at specific position"""
        # Method 1: Modify joint limits (temporary)
        joint_id = self.joint_ids[joint_idx]
        self.model.jnt_limited[joint_id] = 1
        self.model.jnt_range[joint_id] = [position, position]

        # Method 2: Apply constraint force
        # This is done in step() by modifying action

    def _disable_actuator(self, joint_idx):
        """Completely disable an actuator"""
        # Find actuator for this joint
        actuator_id = self._joint_to_actuator(joint_idx)

        # Set actuator gain to zero (no force produced)
        self.model.actuator_gainprm[actuator_id, 0] = 0

        # Or set actuator control range to zero
        self.model.actuator_ctrlrange[actuator_id] = [0, 0]
```

📊 Testing Fault Injection

```python
# Test script to verify faults work correctly
def test_fault_injection():
    env = gym.make('RealAnt-v0')
    env = FaultInjectionWrapper(env, {
        'fault_prob': 1.0,  # Always inject faults
        'max_failed_joints': 2,
        'mode': 'lock'
    })

    obs, info = env.reset()
    print(f"Failed joints: {info['failed_joints']}")

    # Take random actions
    for _ in range(100):
        action = env.action_space.sample()
        obs, reward, done, truncated, info = env.step(action)

        # Verify failed joints don't move
        joint_positions = obs[0:8]
        print(f"Joint positions: {joint_positions}")
        print(f"Modified action: {info['modified_action']}")
```

🔍 Where Everything Connects

1. **Config files** define what faults to inject and when
2. **FaultInjectionWrapper** implements the actual fault logic
3. **CurriculumManager** controls progression of difficulty
4. **Training loop** creates wrapped environment and trains
5. **Evaluation** tests with specific fault scenarios

The beauty is that your PPO agent doesn't need to know about faults - it just sees a "broken" robot and learns to adapt!