From IsaacGymEnvs

Contents

IsaacGymEnvs was a reinforcement learning framework designed for the Isaac Gym Preview Release. As both IsaacGymEnvs and the Isaac Gym Preview Release are now deprecated, the following guide walks through the key differences between IsaacGymEnvs and Isaac Lab, as well as differences in APIs between Isaac Gym Preview Release and Isaac Sim.

> **Note**
>
> The following changes are with respect to Isaac Lab 1.0 release. Please refer to the release notes for any changes in the future releases.

Task Config Setup

In IsaacGymEnvs, task config files were defined in `.yaml` format. With Isaac Lab, configs are now specified using a specialized Python class `configclass`. The `configclass` module provides a wrapper on top of Python's `dataclasses` module. Each environment should specify its own config class annotated by `@configclass` that inherits from `DirectRLEnvCfg`, which can include simulation parameters, environment scene parameters, robot parameters, and task-specific parameters.

Below is an example skeleton of a task config class:

```
from isaaclab.envs import DirectRLEnvCfg
from isaaclab.scene import InteractiveSceneCfg
from isaaclab.sim import SimulationCfg

@configclass
class MyEnvCfg(DirectRLEnvCfg):
   # simulation
   sim: SimulationCfg = SimulationCfg()
   # robot
   robot_cfg: ArticulationCfg = ArticulationCfg()
   # scene
   scene: InteractiveSceneCfg = InteractiveSceneCfg()
   # env
   decimation = 2
   episode_length_s = 5.0
   action_space = 1
   observation_space = 4
   state_space = 0
   # task-specific parameters
   ...
```

Back to top

Simulation Config

Simulation related parameters are defined as part of the `SimulationCfg` class, which is a `configclass` module that holds simulation parameters such as `dt`, `device`, and `gravity`. Each task config must have a variable named `sim` defined that holds the type `SimulationCfg`.

In Isaac Lab, the use of `substeps` has been replaced by a combination of the simulation `dt` and the `decimation` parameters. For example, in IsaacGymEnvs, having `dt=1/60` and `substeps=2` is equivalent to taking 2 simulation steps with `dt=1/120`, but running the task step at `1/60` seconds. The `decimation` parameter is a task parameter that controls the number of simulation steps to take for each task (or RL) step, replacing the `controlFrequencyInv` parameter in IsaacGymEnvs. Thus, the same setup in Isaac Lab will become `dt=1/120` and `decimation=2`.

In Isaac Sim, physx simulation parameters such as `num_position_iterations`, `num_velocity_iterations`, `contact_offset`, `rest_offset`, `bounce_threshold_velocity`, `max_depenetration_velocity` can all be specified on a per-actor basis. These parameters have been moved from the physx simulation config to each individual articulation and rigid body config.

When running simulation on the GPU, buffers in PhysX require pre-allocation for computing and storing information such as contacts, collisions and aggregate pairs. These buffers may need to be adjusted depending on the complexity of the environment, the number of expected contacts and collisions, and the number of actors in the environment. The `PhysxCfg` class provides access for setting the GPU buffer dimensions.

```
# IsaacGymEnvs
sim:

  dt: 0.0166 # 1/60 s
  substeps: 2
  up_axis: "z"
  use_gpu_pipeline: ${eq:${...pipeline},"gpu"}
  gravity: [0.0, 0.0, -9.81]
  physx:
    num_threads: ${....num_threads}
    solver_type: ${....solver_type}
    use_gpu: ${contains:"cuda",${....sim_device}}
    num_position_iterations: 4
    num_velocity_iterations: 0
    contact_offset: 0.02
    rest_offset: 0.001
    bounce_threshold_velocity: 0.2
    max_depenetration_velocity: 100.0
    default_buffer_size_multiplier: 2.0
    max_gpu_contact_pairs: 1048576 # 1024*1024
    num_subscenes: ${....num_subscenes}
    contact_collection: 0
```

```
# IsaacLab
sim: SimulationCfg = SimulationCfg(
   device = "cuda:0" # can be "cpu", "cuda", "cuda:<device_id>
   dt=1 / 120,
   # decimation will be set in the task config
   # up axis will always be Z in isaac sim
   # use_gpu_pipeline is deduced from the device
   gravity=(0.0, 0.0, -9.81),
   physx: PhysxCfg = PhysxCfg(
       # num_threads is no longer needed
       solver_type=1,
       # use_gpu is deduced from the device
       max_position_iteration_count=4,
       max_velocity_iteration_count=0,
       # moved to actor config
       # moved to actor config
       bounce_threshold_velocity=0.2,
       # moved to actor config
       # default_buffer_size_multiplier is no longer needed
       gpu_max_rigid_contact_count=2**23
       # num_subscenes is no longer needed
       # contact_collection is no longer needed
))
```

Scene Config

The `InteractiveSceneCfg` class can be used to specify parameters related to the scene, such as the number of environments and the spacing between environments. Each task config must have a variable named `scene` defined that holds the type `InteractiveSceneCfg`.

```
# IsaacGymEnvs
env:
  numEnvs: ${resolve_default:512,${...num_envs}}
  envSpacing: 4.0
```

```
# IsaacLab
scene: InteractiveSceneCfg = InteractiveSceneCfg(
   num_envs=512,
   env_spacing=4.0)
```

Task Config

Each environment should specify its own config class that holds task specific parameters, such as the dimensions of the observation and action buffers. Reward term scaling parameters can also be specified in the config class.

The following parameters must be set for each environment config:

```
decimation = 2
episode_length_s = 5.0
action_space = 1
observation_space = 4
state_space = 0
```

Note that the maximum episode length parameter (now `episode_length_s`) is in seconds instead of steps as it was in IsaacGymEnvs. To convert between step count to seconds, use the equation: `episode_length_s = dt * decimation * num_steps`

RL Config Setup

RL config files for the rl_games library can continue to be defined in `.yaml` files in Isaac Lab. Most of the content of the config file can be copied directly from IsaacGymEnvs. Note that in Isaac Lab, we do not use hydra to resolve relative paths in config files. Please replace any relative paths such as `${....device}` with the actual values of the parameters.

Additionally, the observation and action clip ranges have been moved to the RL config file. For any `clipObservations` and `clipActions` parameters that were defined in the IsaacGymEnvs task config file, they should be moved to the RL config file in Isaac Lab.

IsaacGymEnvs Task Config	Isaac Lab RL Config

`# IsaacGymEnvs`
`env:`
`  clipObservations: 5.0`
`  clipActions: 1.0`
	`# IsaacLab`
	`params:`
	`  env:`
	`  clip_observations: 5.0`
	`  clip_actions: 1.0`

Environment Creation

In IsaacGymEnvs, environment creation generally included four components: creating the sim object with `create_sim()`, creating the ground plane, importing the assets from MJCF or URDF files, and finally creating the environments by looping through each environment and adding actors into the environments.

Isaac Lab no longer requires calling the `create_sim()` method to retrieve the sim object. Instead, the simulation context is retrieved automatically by the framework. It is also no longer required to use the `sim` as an argument for the simulation APIs.

In replacement of `create_sim()`, tasks can implement the `_setup_scene()` method in Isaac Lab. This method can be used for adding actors into the scene, adding ground plane, cloning the actors, and adding any other optional objects into the scene, such as lights.

IsaacGymEnvs

```
def create_sim(self):
  # set the up axis to be z-up
  self.up_axis = self.cfg["sim"]["up_axis"]

  self.sim = super().create_sim(self.device_id, self.graphics_device_id,
                                self.physics_engine, self.sim_params)
  self._create_ground_plane()
  self._create_envs(self.num_envs, self.cfg["env"]['envSpacing'],
                    int(np.sqrt(self.num_envs)))
```

Isaac Lab

```
def _setup_scene(self):
  self.cartpole = Articulation(self.cfg
  # add ground plane
  spawn_ground_plane(prim_path="/World/
  # clone, filter, and replicate
  self.scene.clone_environments(copy_fr
  self.scene.filter_collisions(global_p
  # add articulation to scene
  self.scene.articulations["cartpole"]
  # add lights
  light_cfg = sim_utils.DomeLightCfg(ir
  light_cfg.func("/World/Light", light_
```

Ground Plane

In Isaac Lab, most of the environment creation process has been simplified into configs with the `configclass` module.

The ground plane can be defined using the `TerrainImporterCfg` class.

```
from isaaclab.terrains import TerrainImporterCfg

terrain = TerrainImporterCfg(
     prim_path="/World/ground",
     terrain_type="plane",
     collision_group=-1,
     physics_material=sim_utils.RigidBodyMaterialCfg(
         friction_combine_mode="multiply",
         restitution_combine_mode="multiply",
         static_friction=1.0,
         dynamic_friction=1.0,
         restitution=0.0,
     ),
 )
```

The terrain can then be added to the scene in `_setup_scene(self)` by referencing the `TerrainImporterCfg` object:

Actors

Isaac Lab and Isaac Sim both use the USD (Universal Scene Description) library for describing the scene. Assets defined in MJCF and URDF formats can be imported to USD using importer tools described in the Importing a New Asset tutorial.

Each Articulation and Rigid Body actor can also have its own config class. The `ArticulationCfg` class can be used to define parameters for articulation actors, including file path, simulation parameters, actuator properties, and initial states.

Within the `ArticulationCfg`, the `spawn` attribute can be used to add the robot to the scene by specifying the path to the robot file. In addition, `RigidBodyPropertiesCfg` can be used to specify simulation properties for the rigid bodies in the articulation. Similarly, the `ArticulationRootPropertiesCfg` class can be used to specify simulation properties for the articulation. Joint properties are now specified as part of the `actuators` dictionary using `ImplicitActuatorCfg`. Joints with the same properties can be grouped into regex expressions or provided as a list of names or expressions.

Actors are added to the scene by simply calling `self.cartpole = Articulation(self.cfg.robot_cfg)`, where `self.cfg.robot_cfg` is an `ArticulationCfg` object. Once initialized, they should also be added to the `InteractiveScene` by calling `self.scene.articulations["cartpole"] = self.cartpole` so that the `InteractiveScene` can traverse through actors in the scene for writing values to the simulation and resetting.

Simulation Parameters for Actors

Some simulation parameters related to Rigid Bodies and Articulations may have different default values between Isaac Gym Preview Release and Isaac Sim. It may be helpful to double check the USD assets to ensure that the default values are applicable for the asset.

For instance, the following parameters in the `RigidBodyAPI` could be different between Isaac Gym Preview Release and Isaac Sim:

RigidBodyAPI Parameter	Default Value in Isaac Sim	Default Value in Isaac Gym Preview Release
Linear Damping	0.00	0.00
Angular Damping	0.05	0.0
Max Linear Velocity	inf	1000
Max Angular Velocity	5729.58008 (degree/s)	64.0 (rad/s)
Max Contact Impulse	inf	1e32

Articulation parameters for the `JointAPI` and `DriveAPI` could be altered as well. Note that the Isaac Sim UI assumes the unit of angle to be degrees. It is particularly worth noting that the `Damping` and `Stiffness` parameters in the `DriveAPI` have the unit of `1/deg` in the Isaac Sim UI but `1/rad` in Isaac Gym Preview Release.

Joint Parameter	Default Value in Isaac Sim	Default Value in Isaac Gym Preview Releases
Maximum Joint Velocity	1000000.0 (deg)	100.0 (rad)

Cloner

Isaac Sim introduced a concept of `Cloner`, which is a class designed for replication during the scene creation process. In IsaacGymEnvs, scenes had to be created by looping through the number of environments. Within each iteration, actors were added to each environment and their handles had to be cached. Isaac Lab eliminates the need for looping through the environments by using the `Cloner` APIs. The scene creation process is as follow:

1. Construct a single environment (what the scene would look like if number of environments = 1)
2. Call `clone_environments()` to replicate the single environment
3. Call `filter_collisions()` to filter out collision between environments (if required)

```
# construct a single environment with the Cartpole robot
self.cartpole = Articulation(self.cfg.robot_cfg)
# clone the environment
self.scene.clone_environments(copy_from_source=False)
# filter collisions
self.scene.filter_collisions(global_prim_paths=[self.cfg.terrain.prim_path])
```

Accessing States from Simulation

APIs for accessing physics states in Isaac Lab require the creation of an `Articulation` or `RigidObject` object. Multiple objects can be initialized for different articulations or rigid bodies in the scene by defining corresponding `ArticulationCfg` or `RigidObjectCfg` config as outlined in the section above. This approach eliminates the need of retrieving body handles to slice states for specific bodies in the scene.

```
self._robot = Articulation(self.cfg.robot)
self._cabinet = Articulation(self.cfg.cabinet)
self._object = RigidObject(self.cfg.object_cfg)
```

We have also removed `acquire` and `refresh` APIs in Isaac Lab. Physics states can be directly applied or retrieved using APIs defined for the articulations and rigid objects.

APIs provided in Isaac Lab no longer require explicit wrapping and un-wrapping of underlying buffers. APIs can now work with tensors directly for reading and writing data.

IsaacGymEnvs	Isaac Lab

`dof_state_tensor = self.gym.acquire_dof_state_tensor(self.sim)`
`self.dof_state = gymtorch.wrap_tensor(dof_state_tensor)`
`self.gym.refresh_dof_state_tensor(self.sim)`
	`self.joint_pos = self._robot.data.joint_pos`
	`self.joint_vel = self._robot.data.joint_vel`

Note some naming differences between APIs in Isaac Gym Preview Release and Isaac Lab. Most `dof` related APIs have been named to `joint` in Isaac Lab. APIs in Isaac Lab also no longer follow the explicit `_tensors` or `_tensor_indexed` suffixes in naming. Indexed versions of APIs now happen implicitly through the optional `indices` parameter.

Most APIs in Isaac Lab also provide the option to specify an `indices` parameter, which can be used when reading or writing data for a subset of environments. Note that when setting states with the `indices` parameter, the shape of the states buffer should match with the dimension of the `indices` list.

IsaacGymEnvs	Isaac Lab

`env_ids_int32 = env_ids.to(dtype=torch.int32)`
`self.gym.set_dof_state_tensor_indexed(self.sim,`
`gymtorch.unwrap_tensor(self.dof_state),`
`gymtorch.unwrap_tensor(env_ids_int32), len(env_ids_int32))`
	`self._robot.write_joint_state_to_sim(joint_pos, j`
	`joint_ids, env_i`

Quaternion Convention

Isaac Lab and Isaac Sim both adopt `wxyz` as the quaternion convention. However, the quaternion convention used in Isaac Gym Preview Release was `xyzw`. Remember to switch all quaternions to use the `xyzw` convention when working indexing rotation data. Similarly, please ensure all quaternions are in `wxyz` before passing them to Isaac Lab APIs.

Articulation Joint Order

Physics simulation in Isaac Sim and Isaac Lab assumes a breadth-first ordering for the joints in a given kinematic tree. However, Isaac Gym Preview Release assumed a depth-first ordering for joints in the kinematic tree. This means that indexing joints based on their ordering may be different in IsaacGymEnvs and Isaac Lab.

In Isaac Lab, the list of joint names can be retrieved with `Articulation.data.joint_names`, which will also correspond to the ordering of the joints in the Articulation.

Creating a New Environment

Each environment in Isaac Lab should be in its own directory following this structure:

```
my_environment/
    - agents/
        - __init__.py
        - rl_games_ppo_cfg.py
    - __init__.py
    my_env.py
```

- `my_environment` is the root directory of the task.
- `my_environment/agents` is the directory containing all RL config files for the task. Isaac Lab supports multiple RL libraries that can each have its own individual config file.
- `my_environment/__init__.py` is the main file that registers the environment with the Gymnasium interface. This allows the training and inferencing scripts to find the task by its name. The content of this file should be as follow:

```
import gymnasium as gym

from . import agents
from .cartpole_env import CartpoleEnv, CartpoleEnvCfg

##
# Register Gym environments.
##

gym.register(
    id="Isaac-Cartpole-Direct-v0",
    entry_point="isaaclab_tasks.direct_workflow.cartpole:CartpoleEnv",
    disable_env_checker=True,
    kwargs={
        "env_cfg_entry_point": CartpoleEnvCfg,
        "rl_games_cfg_entry_point": f"{agents.__name__}:rl_games_ppo_cfg.yaml"
    },
)
```

- `my_environment/my_env.py` is the main python script that implements the task logic and task config class for the environment.

Task Logic

In Isaac Lab, the `post_physics_step` function has been moved to the framework in the base class. Tasks are not required to implement this method, but can choose to override it if a different workflow is desired.

By default, Isaac Lab follows the following flow in logic:

IsaacGymEnvs	Isaac Lab

`pre_physics_step`
`  |-- apply_action`
 	
`post_physics_step`
`  |-- reset_idx()`
`  |-- compute_observation()`
`  |-- compute_reward()`
	`pre_physics_step`
	`  |-- _pre_physics_step(action)`
	`  |-- _apply_action()`
	
	`post_physics_step`
	`  |-- _get_dones()`
	`  |-- _get_rewards()`
	`  |-- _reset_idx()`
	`  |-- _get_observations()`

In Isaac Lab, we also separate the `pre_physics_step` API for processing actions from the policy with the `apply_action` API, which sets the actions into the simulation. This provides more flexibility in controlling when actions should be written to simulation when `decimation` is used. `pre_physics_step` will be called once per step before stepping simulation. `apply_actions` will be called `decimation` number of times for each RL step, once before each simulation step call.

With this approach, resets are performed based on actions from the current step instead of the previous step. Observations will also be computed with the correct states after resets.

We have also performed some renamings of APIs:

- `create_sim(self)` -> `_setup_scene(self)`
- `pre_physics_step(self, actions)` -> `_pre_physics_step(self, actions)` and `_apply_action(self)`
- `reset_idx(self, env_ids)` -> `_reset_idx(self, env_ids)`
- `compute_observations(self)` -> `_get_observations(self)` - `_get_observations()` should now return a dictionary `{"policy": obs}`
- `compute_reward(self)` -> `_get_rewards(self)` - `_get_rewards()` should now return the reward buffer
- `post_physics_step(self)` -> moved to the base class
- In addition, Isaac Lab requires the implementation of `_is_done(self)`, which should return two buffers: the `reset` buffer and the `time_out` buffer.

Putting It All Together

The Cartpole environment is shown here in completion to fully show the comparison between the IsaacGymEnvs implementation and the Isaac Lab implementation.

Task Config

IsaacGymEnvs

```yaml
# used to create the object
name: Cartpole

physics_engine: ${..physics_engine}

# if given, will override the device setting in gym.
env:
  numEnvs: ${resolve_default:512,${...num_envs}}
  envSpacing: 4.0
  resetDist: 3.0
  maxEffort: 400.0

  clipObservations: 5.0
  clipActions: 1.0

  asset:
    assetRoot: "../../assets"
    assetFileName: "urdf/cartpole.urdf"

  enableCameraSensors: False

sim:
  dt: 0.0166 # 1/60 s
  substeps: 2
  up_axis: "z"
  use_gpu_pipeline: ${eq:${...pipeline},"gpu"}
  gravity: [0.0, 0.0, -9.81]
  physx:
    num_threads: ${....num_threads}
    solver_type: ${....solver_type}
    use_gpu: ${contains:"cuda",${....sim_device}}
    num_position_iterations: 4
    num_velocity_iterations: 0
    contact_offset: 0.02
    rest_offset: 0.001
    bounce_threshold_velocity: 0.2
    max_depenetration_velocity: 100.0
    default_buffer_size_multiplier: 2.0
    max_gpu_contact_pairs: 1048576 # 1024*1024
    num_subscenes: ${....num_subscenes}
    contact_collection: 0
```

Isaac Lab

```python
@configclass
class CartpoleEnvCfg(DirectRLEnvCfg):

    # simulation
    sim: SimulationCfg = SimulationCfg(dt=1 / 120)
    # robot
    robot_cfg: ArticulationCfg = CARTPOLE_CFG.replace(
        prim_path="/World/envs/env_.*/Robot")
    cart_dof_name = "slider_to_cart"
    pole_dof_name = "cart_to_pole"
    # scene
    scene: InteractiveSceneCfg = InteractiveSceneCfg(
        num_envs=4096, env_spacing=4.0, replicate_physics=T
    # env
    decimation = 2
    episode_length_s = 5.0
    action_scale = 100.0  # [N]
    action_space = 1
    observation_space = 4
    state_space = 0
    # reset
    max_cart_pos = 3.0
    initial_pole_angle_range = [-0.25, 0.25]
    # reward scales
    rew_scale_alive = 1.0
    rew_scale_terminated = -2.0
    rew_scale_pole_pos = -1.0
    rew_scale_cart_vel = -0.01
    rew_scale_pole_vel = -0.005
```

Task Setup

Isaac Lab no longer requires pre-initialization of buffers through the `acquire_*` APIs that were used in IsaacGymEnvs. It is also no longer necessary to `wrap` and `unwrap` tensors.

IsaacGymEnvs

```
class Cartpole(VecTask):

  def __init__(self, cfg, rl_device, sim_device, graphics_device_id,
   headless, virtual_screen_capture, force_render):
      self.cfg = cfg

      self.reset_dist = self.cfg["env"]["resetDist"]

      self.max_push_effort = self.cfg["env"]["maxEffort"]
      self.max_episode_length = 500

      self.cfg["env"]["numObservations"] = 4
      self.cfg["env"]["numActions"] = 1

      super().__init__(config=self.cfg,
         rl_device=rl_device, sim_device=sim_device,
         graphics_device_id=graphics_device_id, headless=headless,
         virtual_screen_capture=virtual_screen_capture,
         force_render=force_render)

      dof_state_tensor = self.gym.acquire_dof_state_tensor(self.sim)
      self.dof_state = gymtorch.wrap_tensor(dof_state_tensor)
      self.dof_pos = self.dof_state.view(
          self.num_envs, self.num_dof, 2)[..., 0]
      self.dof_vel = self.dof_state.view(
          self.num_envs, self.num_dof, 2)[..., 1]
```

Isaac Lab

```
class CartpoleEnv(DirectRLEnv):
  cfg: CartpoleEnvCfg
  def __init__(self, cfg: CartpoleEnvCfg,
          render_mode: str | None = None, *

      super().__init__(cfg, render_mode, **

      self._cart_dof_idx, _ = self.cartpole
          self.cfg.cart_dof_name)
      self._pole_dof_idx, _ = self.cartpole
          self.cfg.pole_dof_name)
      self.action_scale = self.cfg.action_s

      self.joint_pos = self.cartpole.data.j
      self.joint_vel = self.cartpole.data.j
```

Scene Setup

Scene setup is now done through the `Cloner` API and by specifying actor attributes in config objects. This eliminates the need to loop through the number of environments to set up the environments and avoids the need to set simulation parameters for actors in the task implementation.

IsaacGymEnvs	Isaac Lab

```
def create_sim(self):
    # set the up axis to be z-up given that assets are y-up by default
    self.up_axis = self.cfg["sim"]["up_axis"]

    self.sim = super().create_sim(self.device_id,
        self.graphics_device_id, self.physics_engine,
        self.sim_params)
    self._create_ground_plane()
    self._create_envs(self.num_envs,
        self.cfg["env"]['envSpacing'],
        int(np.sqrt(self.num_envs)))

def _create_ground_plane(self):
    plane_params = gymapi.PlaneParams()
    # set the normal force to be z dimension
    plane_params.normal = (gymapi.Vec3(0.0, 0.0, 1.0)
        if self.up_axis == 'z'
        else gymapi.Vec3(0.0, 1.0, 0.0))
    self.gym.add_ground(self.sim, plane_params)

def _create_envs(self, num_envs, spacing, num_per_row):
    # define plane on which environments are initialized
    lower = (gymapi.Vec3(0.5 * -spacing, -spacing, 0.0)
        if self.up_axis == 'z'
        else gymapi.Vec3(0.5 * -spacing, 0.0, -spacing))
    upper = gymapi.Vec3(0.5 * spacing, spacing, spacing)

    asset_root = os.path.join(os.path.dirname(
        os.path.abspath(__file__)), "../../assets")
    asset_file = "urdf/cartpole.urdf"

    if "asset" in self.cfg["env"]:
        asset_root = os.path.join(os.path.dirname(
            os.path.abspath(__file__)),
            self.cfg["env"]["asset"].get("assetRoot", asset_root))
        asset_file = self.cfg["env"]["asset"].get(
            "assetFileName", asset_file)

    asset_path = os.path.join(asset_root, asset_file)
    asset_root = os.path.dirname(asset_path)
    asset_file = os.path.basename(asset_path)

    asset_options = gymapi.AssetOptions()
    asset_options.fix_base_link = True
    cartpole_asset = self.gym.load_asset(self.sim,
        asset_root, asset_file, asset_options)
    self.num_dof = self.gym.get_asset_dof_count(
        cartpole_asset)

    pose = gymapi.Transform()
    if self.up_axis == 'z':
        pose.p.z = 2.0
        pose.r = gymapi.Quat(0.0, 0.0, 0.0, 1.0)
    else:
        pose.p.y = 2.0
        pose.r = gymapi.Quat(
            -np.sqrt(2)/2, 0.0, 0.0, np.sqrt(2)/2)

    self.cartpole_handles = []
    self.envs = []
    for i in range(self.num_envs):
        # create env instance
        env_ptr = self.gym.create_env(
            self.sim, lower, upper, num_per_row
        )
        cartpole_handle = self.gym.create_actor(
            env_ptr, cartpole_asset, pose,
            "cartpole", i, 1, 0)

        dof_props = self.gym.get_actor_dof_properties(
            env_ptr, cartpole_handle)
        dof_props['driveMode'][0] = gymapi.DOF_MODE_EFFORT
        dof_props['driveMode'][1] = gymapi.DOF_MODE_NONE
        dof_props['stiffness'][:] = 0.0
        dof_props['damping'][:] = 0.0
        self.gym.set_actor_dof_properties(env_ptr, c
            artpole_handle, dof_props)

        self.envs.append(env_ptr)
        self.cartpole_handles.append(cartpole_handle)
```

```
def _setup_scene(self):
    self.cartpole = Articulation(self.cfg
    # add ground plane
    spawn_ground_plane(prim_path="/World/
        cfg=GroundPlaneCfg())
    # clone, filter, and replicate
    self.scene.clone_environments(
        copy_from_source=False)
    self.scene.filter_collisions(
        global_prim_paths=[])
    # add articulation to scene
    self.scene.articulations["cartpole"]
    # add lights
    light_cfg = sim_utils.DomeLightCfg(
        intensity=2000.0, color=(0.75, 0.
    light_cfg.func("/World/Light", light_

CARTPOLE_CFG = ArticulationCfg(
    spawn=sim_utils.UsdFileCfg(
        usd_path=f"{ISAACLAB_NUCLEUS_DIR}
        rigid_props=sim_utils.RigidBodyPr
            rigid_body_enabled=True,
            max_linear_velocity=1000.0,
            max_angular_velocity=1000.0,
            max_depenetration_velocity=10
            enable_gyroscopic_forces=True
        ),
        articulation_props=sim_utils.Arti
            enabled_self_collisions=False
            solver_position_iteration_cou
            solver_velocity_iteration_cou
            sleep_threshold=0.005,
            stabilization_threshold=0.001
        ),
    ),
    init_state=ArticulationCfg.InitialSta
        pos=(0.0, 0.0, 2.0),
        joint_pos={"slider_to_cart": 0.0,
    ),
    actuators={
        "cart_actuator": ImplicitActuator
            joint_names_expr=["slider_to_
            effort_limit_sim=400.0,
            velocity_limit_sim=100.0,
            stiffness=0.0,
            damping=10.0,
        ),
        "pole_actuator": ImplicitActuator
            joint_names_expr=["cart_to_po
            effort_limit_sim=400.0, veloc
            stiffness=0.0, damping=0.0
        ),
    },
)
```

Pre and Post Physics Step

In IsaacGymEnvs, due to limitations of the GPU APIs, observations had stale data when environments had to perform resets. This restriction has been eliminated in Isaac Lab, and thus, tasks follow the correct workflow of applying actions, stepping simulation, collecting states, computing dones, calculating rewards, performing resets, and finally computing observations. This workflow is done automatically by the framework such that a `post_physics_step` API is not required in the task. However, individual tasks can override the `step()` API to control the workflow.

IsaacGymEnvs

```
def pre_physics_step(self, actions):
    actions_tensor = torch.zeros(
        self.num_envs * self.num_dof,
        device=self.device, dtype=torch.float)
    actions_tensor[::self.num_dof] = actions.to(
        self.device).squeeze() * self.max_push_effort
    forces = gymtorch.unwrap_tensor(actions_tensor)
    self.gym.set_dof_actuation_force_tensor(
        self.sim, forces)

def post_physics_step(self):
    self.progress_buf += 1

    env_ids = self.reset_buf.nonzero(
        as_tuple=False).squeeze(-1)
    if len(env_ids) > 0:
        self.reset_idx(env_ids)

    self.compute_observations()
    self.compute_reward()
```

IsaacLab

```
def _pre_physics_step(self, actions: torch.Tensor) -> None
    self.actions = self.action_scale * actions

def _apply_action(self) -> None:
    self.cartpole.set_joint_effort_target(
         self.actions, joint_ids=self._cart_dof_idx)
```

Dones and Resets

In Isaac Lab, `dones` are computed in the `_get_dones()` method and should return two variables: `resets` and `time_out`. Tracking of the `progress_buf` has been moved to the base class and is now automatically incremented and reset by the framework. The `progress_buf` variable has also been renamed to `episode_length_buf`.

IsaacGymEnvs

```
def reset_idx(self, env_ids):
    positions = 0.2 * (torch.rand((len(env_ids), self.num_dof),
        device=self.device) - 0.5)
    velocities = 0.5 * (torch.rand((len(env_ids), self.num_dof),
        device=self.device) - 0.5)

    self.dof_pos[env_ids, :] = positions[:]
    self.dof_vel[env_ids, :] = velocities[:]

    env_ids_int32 = env_ids.to(dtype=torch.int32)
    self.gym.set_dof_state_tensor_indexed(self.sim,
        gymtorch.unwrap_tensor(self.dof_state),
        gymtorch.unwrap_tensor(env_ids_int32), len(env_ids_int32))
    self.reset_buf[env_ids] = 0
    self.progress_buf[env_ids] = 0
```

Isaac Lab

```
def _get_dones(self) -> tuple[torch.Tensor, t
    self.joint_pos = self.cartpole.data.joint
    self.joint_vel = self.cartpole.data.joint

    time_out = self.episode_length_buf >= sel
    out_of_bounds = torch.any(torch.abs(
        self.joint_pos[:, self._cart_dof_idx]
        dim=1)
    out_of_bounds = out_of_bounds | torch.any
        torch.abs(self.joint_pos[:, self._pol
        dim=1)
    return out_of_bounds, time_out

def _reset_idx(self, env_ids: Sequence[int] |
    if env_ids is None:
        env_ids = self.cartpole._ALL_INDICES
    super()._reset_idx(env_ids)

    joint_pos = self.cartpole.data.default_jo
    joint_pos[:, self._pole_dof_idx] += sampl
        self.cfg.initial_pole_angle_range[0]
        self.cfg.initial_pole_angle_range[1]
        joint_pos[:, self._pole_dof_idx].shap
        joint_pos.device,
    )
    joint_vel = self.cartpole.data.default_jo

    default_root_state = self.cartpole.data.d
    default_root_state[:, :3] += self.scene.e

    self.joint_pos[env_ids] = joint_pos

    self.cartpole.write_root_pose_to_sim(
        default_root_state[:, :7], env_ids)
    self.cartpole.write_root_velocity_to_sim(
        default_root_state[:, 7:], env_ids)
    self.cartpole.write_joint_state_to_sim(
        joint_pos, joint_vel, None, env_ids)
```

Observations

In Isaac Lab, the `_get_observations()` API should now return a dictionary containing the `policy` key with the observation buffer as the value. For asymmetric policies, the dictionary should also include a `critic` key that holds the state buffer.

IsaacGymEnvs

```
def compute_observations(self, env_ids=None):
    if env_ids is None:
        env_ids = np.arange(self.num_envs)

    self.gym.refresh_dof_state_tensor(self.sim)

    self.obs_buf[env_ids, 0] = self.dof_pos[env_ids, 0]
    self.obs_buf[env_ids, 1] = self.dof_vel[env_ids, 0]
    self.obs_buf[env_ids, 2] = self.dof_pos[env_ids, 1]
    self.obs_buf[env_ids, 3] = self.dof_vel[env_ids, 1]

    return self.obs_buf
```

Isaac Lab

```
def _get_observations(self) -> dict:
    obs = torch.cat(
        (
            self.joint_pos[:, self._pole_dof_idx[0]],
            self.joint_vel[:, self._pole_dof_idx[0]],
            self.joint_pos[:, self._cart_dof_idx[0]],
            self.joint_vel[:, self._cart_dof_idx[0]],
        ),
        dim=-1,
    )
    observations = {"policy": obs}
    return observations
```

Rewards

In Isaac Lab, the reward method `_get_rewards` should return the reward buffer as a return value. Similar to IsaacGymEnvs, computations in the reward function can also be performed using pytorch jit by adding the `@torch.jit.script` annotation.

IsaacGymEnvs

```
def compute_reward(self):
    # retrieve environment observations from buffer
    pole_angle = self.obs_buf[:, 2]
    pole_vel = self.obs_buf[:, 3]
    cart_vel = self.obs_buf[:, 1]
    cart_pos = self.obs_buf[:, 0]

    self.rew_buf[:], self.reset_buf[:] = compute_cartpole_reward(
        pole_angle, pole_vel, cart_vel, cart_pos,
        self.reset_dist, self.reset_buf,
        self.progress_buf, self.max_episode_length
    )

@torch.jit.script
def compute_cartpole_reward(pole_angle, pole_vel,
                            cart_vel, cart_pos,
                            reset_dist, reset_buf,
                            progress_buf, max_episode_length):

    reward = (1.0 - pole_angle * pole_angle -
        0.01 * torch.abs(cart_vel) -
        0.005 * torch.abs(pole_vel))

    # adjust reward for reset agents
    reward = torch.where(torch.abs(cart_pos) > reset_dist,
        torch.ones_like(reward) * -2.0, reward)
    reward = torch.where(torch.abs(pole_angle) > np.pi / 2,
        torch.ones_like(reward) * -2.0, reward)

    reset = torch.where(torch.abs(cart_pos) > reset_dist,
        torch.ones_like(reset_buf), reset_buf)
    reset = torch.where(torch.abs(pole_angle) > np.pi / 2,
        torch.ones_like(reset_buf), reset_buf)
    reset = torch.where(progress_buf >= max_episode_length - 1,
        torch.ones_like(reset_buf), reset)
```

Isaac Lab

```
def _get_rewards(self) -> torch.Tensor:
    total_reward = compute_rewards(
        self.cfg.rew_scale_alive,
        self.cfg.rew_scale_terminated,
        self.cfg.rew_scale_pole_pos,
        self.cfg.rew_scale_cart_vel,
        self.cfg.rew_scale_pole_vel,
        self.joint_pos[:, self._pole_dof_idx[0
        self.joint_vel[:, self._pole_dof_idx[0
        self.joint_pos[:, self._cart_dof_idx[0
        self.joint_vel[:, self._cart_dof_idx[0
        self.reset_terminated,
    )
    return total_reward

@torch.jit.script
def compute_rewards(
    rew_scale_alive: float,
    rew_scale_terminated: float,
    rew_scale_pole_pos: float,
    rew_scale_cart_vel: float,
    rew_scale_pole_vel: float,
    pole_pos: torch.Tensor,
    pole_vel: torch.Tensor,
    cart_pos: torch.Tensor,
    cart_vel: torch.Tensor,
    reset_terminated: torch.Tensor,
):
    rew_alive = rew_scale_alive * (1.0 - reset
    rew_termination = rew_scale_terminated * r
    rew_pole_pos = rew_scale_pole_pos * torch.
        torch.square(pole_pos), dim=-1)
    rew_cart_vel = rew_scale_cart_vel * torch.
        torch.abs(cart_vel), dim=-1)
    rew_pole_vel = rew_scale_pole_vel * torch.
        torch.abs(pole_vel), dim=-1)
    total_reward = (rew_alive + rew_terminatic
                    + rew_pole_pos + rew_cart
    return total_reward
```

Launching Training

To launch a training in Isaac Lab, use the command:

```
python scripts/reinforcement_learning/rl_games/train.py --task=Isaac-Cartpole-Direct-v0 --headless
```

Launching Inferencing

To launch inferencing in Isaac Lab, use the command:

```
python scripts/reinforcement_learning/rl_games/play.py --task=Isaac-Cartpole-Direct-v0 --num_envs=25 --checkpoint=<pat
```