First diff is for MBRL-Lib

Second diff is for Gym-Duckietown

Comparing changes

This is a direct comparison between two commits made in this repository or its related repositories. View the default comparison for this range here.

base: c213d0b ↔ compare: 1fc1e15

Showing **20 changed files** with **2,546 additions** and **14 deletions**. Split | Unified

1 .gitignore

```
28  28  MANIFEST
29  29  wandb/
30  30  scratch.py
    31 + scratch.txt
```

42 README.md

```
28  28   Then to activate it use:
29  29
30  30      conda activate RLDucky
    31 +
    32 + You will use this python virtual environment whenever you are running code from this repo. You should use a
       seperate virtual environment when running code from the duckietown repo, such as in the commands below.
31  33  ### Gym-Duckietown
32  34  <!-- You will need to do the [duckietown laptop]
       (https://docs.duckietown.org/daffy/opmanual_duckiebot/out/laptop_setup.html) setup to use the gym-duckietown --
       >
    35 +
    36 + Make a new virtual environment for the Gym-Duckietown repo:
    37 +
    38 +     conda create --name GymDucky python=3.8
    39 +
    40 +  Then to activate it use:
    41 +
    42 +     conda activate GymDucky
    43 +
    44 +
33  45  The first repo to clone is [Gym-Duckietown](https://github.com/duckietown/gym-duckietown.git), and make sure
       you checkout and use the master branch. Additionally you can install the required python packages for that repo
       via the command `pip install -e .` where `.` specifies the current directory.
34  46
    47 +     conda activate GymDucky
35  48      git clone https://github.com/duckietown/gym-duckietown.git
36  49      cd gym-duckietown
37  50      git checkout master
42  55  While trying to use Gym-Duckietown we ran into an issue involving a malfunctioning / deprecated `geometry`
       module. If you run into the same problem, you can just comment out that import. So just navigate to the `gym-
       duckietown/gym_duckietown/simulator.py` file and comment out the `import geometry` line.
43  56
44  57  ### Importing Gym-Duckietown into MBRL-Lib
45      -
46  58
    59 + For the following commands, switch back to your RLDucky environment:
    60 +
    61 +     conda activate RLDucky
    62 +
47  63  To use the Duckietown environment seamlessly with MBRL-Lib, we will have to add the `gym-duckietown` repo as a
       python module to our python installation. There are two ways of doing this.
48  64
49  65  #### Option 1: Using Path (.pth) Files
```

```
Clone this repository and install the required python packages:

    git clone https://github.com/alik-git/duckietown-mbrl-lib
-   cd mbrl-lib
+   cd duckietown-mbrl-lib
    conda activate RLDucky
    pip install -e ".[ducky]"

    cd /your/path/to/mbrl-lib
    python -m pytest tests/mujoco

- #### Side Note:
- While trying to run MuJoCo we twice ran into an error relating to something involving `undefined symbol: __glewBindBuffer`, and the only fix we found (from a Reddit [thread](https://www.reddit.com/r/reinforcementlearning/comments/qay11a/how_to_use_mujoco_from_python3/)) was to install the following packages:
+ #### Mujoco Issues:
+
+ A few issues with mujoco - and I've encountered this during other projects as well - are about mujoco not neatly attaching to a rendering engine in your software environment. On Ubuntu it seems to try and use the [OpenGL](https://en.wikipedia.org/wiki/OpenGL) rendering engine, and to find it is uses [GLEW](https://github.com/nigels-com/glew). But this process doesn't always end up being seamless. Below are a few issues that we ran into:
+ #### Rendering Issue 1:
+
+ While trying to run MuJoCo we ran into an error twice relating to something involving `undefined symbol: __glewBindBuffer`, and the only fix we found (from a Reddit [thread](https://www.reddit.com/r/reinforcementlearning/comments/qay11a/how_to_use_mujoco_from_python3/)) was to install the following packages:

    sudo apt install curl git libgl1-mesa-dev libgl1-mesa-glx libglew-dev \
    libosmesa6-dev software-properties-common net-tools unzip vim \
    virtualenv wget xpra xserver-xorg-dev libglfw3-dev patchelf

+ #### Rendering Issue 2:
+
+ This issue is again about the [dm_control](https://github.com/deepmind/dm_control) installation of mujoco, you basically get an [error](https://github.com/deepmind/dm_control/issues/283) saying `mujoco.FatalError: gladLoadGL error`, and as described in the [aforementioned link](https://github.com/deepmind/dm_control/issues/283#issuecomment-1095490151), the solution is to find your dm_control package in your python installation, something like:
+
+ ```
+ /home/username/anaconda3/envs/RLDucky/lib/python3.8/site-packages/dm_control/_render/glfw_renderer.py
+ ```
+
+ and modify it to add some extra lines as described [here](https://github.com/deepmind/dm_control/issues/283#issuecomment-1095490151).
+

### Logging and Visualization (W&B)
We use [Weights & Biases](https://wandb.ai/site) for logging and visualizing our run metrics. If you're unfamiliar with Weights & Biases, it is a powerful and convenient library to organize and track ML experiments. You can take look at their [quick-start guide](https://docs.wandb.ai/quickstart) and [documentation](https://docs.wandb.ai/), and you'll have to create an account to be able to view and use the dashboard, you can do so [here](https://wandb.ai/site).

To run an experiment you can use commands in the following format:

    python -m mbrl.examples.main algorithm=planet dynamics_model=planet overrides=planet_duckietown
+
+ Here is another example of a command with shorter episodes:
+
+     python -m mbrl.examples.main algorithm=planet dynamics_model=planet overrides=planet_cheetah_run algorithm.test_frequency=2 overrides.sequence_length=10 overrides.batch_size=10

You will see the output of your run in the terminal as well as in a results file created by Hydra located by default at `.exp/planet/default/duckietown_gym_env/yyyy.mm.dd/hhmmss`; you can change the root directory
```

172	206	(`./exp`) by passing `root_dir=path-to-your-dir`, and the experiment sub-folder (`default`) by

451 base_sweep_config.yaml

Load diff

Large diffs are not rendered by default.

300 mbrl/algorithms/dreamer.py

```
@@ -0,0 +1,300 @@
+ ####################
+ ##### FOR GLEN ####
+ ####################
+ # This file is basically a copy of MBRL-Lib's PlaNet, modified to accommodate
+ # the dreamer method.
+ ####################
+
+ # Copyright (c) Facebook, Inc. and its affiliates. All Rights Reserved.
+ #
+ # This source code is licensed under the MIT license found in the
+ # LICENSE file in the root directory of this source tree.
+ import os
+ import pathlib
+ from typing import List, Optional, Union, cast
+
+ import gym
+ import hydra
+ import numpy as np
+ import omegaconf
+ import torch
+
+ import mbrl.constants
+ #import mbrl.third_party.pytorch_sac as pytorch_sac
+
+ from mbrl.env.termination_fns import no_termination
+ from mbrl.models import ModelEnv, ModelTrainer
+ from mbrl.planning import RandomAgent, create_trajectory_optim_agent_for_model
+ from mbrl.util import Logger
+ from mbrl.util.common import (
+     create_replay_buffer,
+     get_sequence_buffer_iterator,
+     rollout_agent_trajectories,
+ )
+ #from mbrl.planning.sac_wrapper import SACAgent
+ from mbrl.planning.dreamer_wrapper import DreamerAgent
+
+ import wandb
+ from gym.wrappers import Monitor
+
+
+ # Original modified from PlaNet
+
+ METRICS_LOG_FORMAT = [
+     ("observations_loss", "OL", "float"),
+     ("reward_loss", "RL", "float"),
+     ("gradient_norm", "GN", "float"),
+     ("kl_loss", "KL", "float"),
+ ]
+
+
+ def train(
+         env: gym.Env,
+         cfg: omegaconf.DictConfig,
```

```
+         silent: bool = False,
+         work_dir: Union[Optional[str], pathlib.Path] = None,
+ ) -> np.float32:
+     # Experiment initialization
+     debug_mode = cfg.get("debug_mode", False)
+
+     if work_dir is None:
+         work_dir = os.getcwd()
+     work_dir = pathlib.Path(work_dir)
+     print(f"Results will be saved at {work_dir}.")
+     wandb.config.update({"work_dir": str(work_dir)})
+
+     if silent:
+         logger = None
+     else:
+         logger = Logger(work_dir)
+         logger.register_group("metrics", METRICS_LOG_FORMAT, color="yellow")
+         logger.register_group(
+             mbrl.constants.RESULTS_LOG_NAME,
+             [
+                 ("env_step", "S", "int"),
+                 ("train_episode_reward", "RT", "float"),
+                 ("episode_reward", "ET", "float"),
+             ],
+             color="green",
+         )
+
+     rng = torch.Generator(device=cfg.device)
+     rng.manual_seed(cfg.seed)
+     np_rng = np.random.default_rng(seed=cfg.seed)
+
+     # Create replay buffer and collect initial data
+     replay_buffer = create_replay_buffer(
+         cfg,
+         env.observation_space.shape,
+         env.action_space.shape,
+         collect_trajectories=True,
+         rng=np_rng,
+     )
+     rollout_agent_trajectories(
+         env,
+         cfg.algorithm.num_initial_trajectories,
+         RandomAgent(env),
+         agent_kwargs={},
+         replay_buffer=replay_buffer,
+         collect_full_trajectories=True,
+         trial_length=cfg.overrides.trial_length,
+         agent_uses_low_dim_obs=False,
+     )
+
+     # Create PlaNet model
+     cfg.dynamics_model.action_size = env.action_space.shape[0]
+
+     # Use hydra to create a dreamer model (really uses PlaNet model)
+     dreamer = hydra.utils.instantiate(cfg.dynamics_model)
+     # Give it the real gym env to model
+     dreamer.setGymEnv(env, work_dir)
+
+     # adam optim that takes into account all 3 network losses
+     # actor, critic, model
+     dreamer_optim = dreamer.configure_optimizers()
+     assert isinstance(dreamer, mbrl.models.DreamerModel)
+     model_env = ModelEnv(env, dreamer, no_termination, generator=rng)
+     trainer = ModelTrainer(dreamer, logger=logger, optim_lr=1e-3, optim_eps=1e-4)
+
+     ####################
+     ##### FOR GLEN ####
+
```

```
+    ###################
122  +    #
123  +    # Some thoughts on how we were approaching this problem
124  +
125  +    # This library has a function called
126  +    # create_trajectory_optim_agent_for_model() that is used to create an agent
127  +    # (an agent that essentially uses CEM planning) given a world model.
128  +    #
129  +    # Example: agent =
130  +    # create_trajectory_optim_agent_for_model(model_env,cfg.algorithm.agent)
131  +    #
132  +    # But in our case, dreamer is the world model i.e. model_env, but Dreamer
133  +    # also includes an "agent" in the form of an actor and critic network, and
134  +    # Dreamer therefore doesn't need CEM planning.
135  +    #
136  +    # So instead of doing it the "MBRL-Lib" way and creating an agent, we just
137  +    # use the world model i.e. Dreamer to sample actions from. This doesn't sit
138  +    # well with the structure of this library so in future work we wil try to
139  +    # separate the model part of Dreamer (which is just PlaNet) and the
140  +    # agent/planning part of Dreamer, so that it fits better with this library.
141  +
142  +
143  +    # Create Dreamer Agent (Action and Value model), are these needed for this to operate properly?
144  +    # This agent rolls outs trajectories using ModelEnv, which uses planet.sample()
145  +    # to simulate the trajectories from the prior transition model
146  +    # The starting point for trajectories is conditioned on the latest observation,
147  +    # for which we use planet.update_posterior() after each environment step
148  +    #the CEM way
149  +    # agent = create_trajectory_optim_agent_for_model(model_env, cfg.algorithm.agent)
150  +    #the SAC/Dreamer way
151  +    # agent = different_or_same_function(model_env, cfg.algorithm.agent)
152  +
153  +
154  +    # Callback and containers to accumulate training statistics and average over batch
155  +    rec_losses: List[float] = []
156  +    reward_losses: List[float] = []
157  +    kl_losses: List[float] = []
158  +    grad_norms: List[float] = []
159  +
160  +    def get_metrics_and_clear_metric_containers():
161  +        metrics_ = {
162  +            "observations_loss": np.mean(rec_losses).item(),
163  +            "reward_loss": np.mean(reward_losses).item(),
164  +            "gradient_norm": np.mean(grad_norms).item(),
165  +            "kl_loss": np.mean(kl_losses).item(),
166  +        }
167  +
168  +        for c in [rec_losses, reward_losses, kl_losses, grad_norms]:
169  +            c.clear()
170  +
171  +        return metrics_
172  +
173  +    def batch_callback(_epoch, _loss, meta, _mode):
174  +        if meta:
175  +            rec_losses.append(meta["observations_loss"])
176  +            reward_losses.append(meta["reward_loss"])
177  +            kl_losses.append(meta["kl_loss"])
178  +            if "grad_norm" in meta:
179  +                grad_norms.append(meta["grad_norm"])
180  +
181  +    def is_test_episode(episode_):
182  +        return episode_ % cfg.algorithm.test_frequency == 0
183  +
184  +    # PlaNet loop
185  +    step = replay_buffer.num_stored
186  +    total_rewards = 0.0
187  +    for episode in range(cfg.algorithm.num_episodes):
188
```

```
+        dreamer.set_curr_episode(episode)
189 +
190 +        # Train the model for one epoch of `num_grad_updates`
191 +        dataset, _ = get_sequence_buffer_iterator(
192 +            replay_buffer,
193 +            cfg.overrides.batch_size,
194 +            0,  # no validation data
195 +            cfg.overrides.sequence_length,
196 +            max_batches_per_loop_train=cfg.overrides.num_grad_updates,
197 +            use_simple_sampler=True,
198 +        )
199 +        trainer.train(
200 +            dataset, num_epochs=1, batch_callback=batch_callback, evaluate=False
201 +        )
202 +        dreamer.save(work_dir / "dreamer.pth")
203 +        replay_buffer.save(work_dir)
204 +        metrics = get_metrics_and_clear_metric_containers()
205 +        logger.log_data("metrics", metrics)
206 +        wandb_metrics = metrics
207 +        wandb_metrics["global_episode"] = episode
208 +        wandb.log(wandb_metrics)
209 +
210 +        if is_test_episode(episode):
211 +            print("AHH ITS A TEST EPISODE!!!")
212 +            curr_env = Monitor(env, work_dir, force=True)
213 +            dreamer.set_currently_testing(True)
214 +        else:
215 +            curr_env = env
216 +            dreamer.set_currently_testing(False)
217 +
218 +        # Collect one episode of data
219 +        episode_reward = 0.0
220 +        obs = curr_env.reset()
221 +        # want to do
222 +        #agent.reset()
223 +        dreamer.reset_world_model(device=cfg.device)
224 +        state = None
225 +        action = None
226 +        done = False
227 +        while not done:
228 +
229 +            # hacky check to see if we are using the duckietown environment
230 +            # I just see if the obs shape ends in 3, for the other envs it does not
231 +            if obs.shape[-1] == 3:
232 +                dreamer.in_duckietown = True
233 +
234 +            if dreamer.in_duckietown:
235 +                obs = np.transpose(obs, (2,0,1))
236 +                pass
237 +
238 +            # want to do
239 +            # func dreamer.update(...)
240 +            #   planet.update(..)
241 +            #   actor_net.update(..)
242 +            #   value_net.updare(..)
243 +            # dreamer.update_alg(obs, action=action, rng=rng)
244 +            '''
245 +            Don't need yet, noise in implementation
246 +            action_noise = (
247 +                0
248 +                if is_test_episode(episode)
249 +                else cfg.overrides.action_noise_std
250 +                     * np_rng.standard_normal(curr_env.action_space.shape[0])
251 +            )
252 +            '''
253 +            # want to do (kinda)
254 +            # action, _ = dreamer.policy(obs)
255
```

```
+            #action sampler for Dreamer wants tensors with a batch dimension
256 +            action, _, state, = dreamer.action_sampler_fn(torch.FloatTensor(obs).unsqueeze(0), state, dreamer.ex
257 +            #action = action.squeeze(0).numpy()
258 +            '''
259 +            Already have noise in the implementation
260 +            action = action + action_noise
261 +            # action = agent.act(obs) + action_noise
262 +            action = np.clip(action, -1.0, 1.0)  # to account for the noise
263 +            '''
264 +
265 +            if dreamer.in_duckietown:
266 +                obs = np.transpose(obs, (1,2,0))
267 +            action = action.squeeze(0).cpu().numpy() #env likes numpy arrays for actions unbatched
268 +            next_obs, reward, done, info = curr_env.step(action)
269 +            replay_buffer.add(obs, action, next_obs, reward, done)
270 +            episode_reward += reward
271 +            obs = next_obs
272 +            if debug_mode:
273 +                print(f"step: {step}, reward: {reward}.")
274 +            step += 1
275 +        total_rewards += episode_reward
276 +        logger.log_data(
277 +            mbrl.constants.RESULTS_LOG_NAME,
278 +            {
279 +                "episode_reward": episode_reward * is_test_episode(episode),
280 +                "train_episode_reward": episode_reward * (1 - is_test_episode(episode)),
281 +                "env_step": step,
282 +            },
283 +        )
284 +        wandb.log(
285 +            {
286 +                "episode_reward": episode_reward * is_test_episode(episode),
287 +                "train_episode_reward": episode_reward * (1 - is_test_episode(episode)),
288 +                "env_step": step,
289 +                "global_episode": episode
290 +            }
291 +        )
292 +        avg_ep_reward = total_rewards / (episode+1)
293 +        wandb.log({'average_episode_reward': avg_ep_reward, "global_episode": episode})
294 +
295 +    # returns average episode reward (e.g., to use for tuning learning curves)
296 +    avg_ep_reward = total_rewards / cfg.algorithm.num_episodes
297 +    wandb.log({'average_episode_reward': avg_ep_reward, "global_episode": episode})
298 +    return avg_ep_reward
299 +
300 +
```

40 mbrl/algorithms/planet.py

```
...  ...  @@ -1,3 +1,8 @@
     1  + ####################
     2  + ##### FOR GLEN ####
     3  + ####################
     4  + # We added video recording for the Duckietown environments and better logging.
     5  + ####################
1    6    # Copyright (c) Facebook, Inc. and its affiliates. All Rights Reserved.
2    7    #
3    8    # This source code is licensed under the MIT license found in the
24   29   )
25   30
26   31   import wandb
     32 + from gym.wrappers import Monitor
     33 + from gym.wrappers.monitoring.video_recorder import VideoRecorder
     34 + from PIL import Image
     35 + import cv2
27   36
28   37   METRICS_LOG_FORMAT = [
```

```
29  38      ("observations_loss", "OL", "float"),
46  55          work_dir = os.getcwd()
47  56      work_dir = pathlib.Path(work_dir)
48  57      print(f"Results will be saved at {work_dir}.")
    58  +   wandb.config.update({"work_dir": str(work_dir)})
49  59
50  60      if silent:
51  61          logger = None
90  100     planet = hydra.utils.instantiate(cfg.dynamics_model)
91  101     assert isinstance(planet, mbrl.models.PlaNetModel)
92  102     model_env = ModelEnv(env, planet, no_termination, generator=rng)
93      -   trainer = ModelTrainer(planet, logger=logger, optim_lr=1e-3, optim_eps=1e-4)
    103 +   print(f"\n###\nUsing learning rate: {cfg.overrides.model_learning_rate}\n###")
    104 +
    105 +   trainer = ModelTrainer(planet, logger=logger, optim_lr=cfg.overrides.model_learning_rate, optim_eps=1e-4)
94  106
95  107     # Create CEM agent
96  108     # This agent rolls outs trajectories using ModelEnv, which uses planet.sample()
133 145     step = replay_buffer.num_stored
134 146     total_rewards = 0.0
135 147     for episode in range(cfg.algorithm.num_episodes):
    148 +       print(f"###\nNow on episode {episode} of {cfg.algorithm.num_episodes - 1}")
136 149         # Train the model for one epoch of `num_grad_updates`
137 150         dataset, _ = get_sequence_buffer_iterator(
138 151             replay_buffer,
148 161         planet.save(work_dir / "planet.pth")
149 162         replay_buffer.save(work_dir)
150 163         metrics = get_metrics_and_clear_metric_containers()
    164 +       wandb_metrics = metrics
    165 +       wandb_metrics['global_episode'] = episode
151 166         logger.log_data("metrics", metrics)
152     -       wandb.log(metrics)
    167 +
    168 +       wandb.log(wandb_metrics)
    169 +
    170 +       if is_test_episode(episode):
    171 +           print(f"Reached Test Episode! Episode: {episode}")
    172 +           curr_env = Monitor(env, work_dir / "videos" / f"{episode:05d}", force=True)
    173 +           planet.save(work_dir / "models" / f"{episode:05d}" / "planet.pth")
    174 +       else:
    175 +           curr_env = env
153 176
154 177         # Collect one episode of data
155 178         episode_reward = 0.0
156     -       obs = env.reset()
    179 +       obs = curr_env.reset()
157 180         agent.reset()
158 181         planet.reset_posterior()
159 182         action = None
164 187                 0
165 188                 if is_test_episode(episode)
166 189                 else cfg.overrides.action_noise_std
167     -               * np_rng.standard_normal(env.action_space.shape[0])
    190 +               * np_rng.standard_normal(curr_env.action_space.shape[0])
168 191             )
169 192             action = agent.act(obs) + action_noise
170 193             action = np.clip(action, -1.0, 1.0)  # to account for the noise
171     -           next_obs, reward, done, info = env.step(action)
    194 +           next_obs, reward, done, info = curr_env.step(action)
172 195             replay_buffer.add(obs, action, next_obs, reward, done)
173 196             episode_reward += reward
174 197             obs = next_obs
189 212                 "episode_reward": episode_reward * is_test_episode(episode),
190 213                 "train_episode_reward": episode_reward * (1 - is_test_episode(episode)),
191 214                 "env_step": step,
    215 +               "global_episode": episode
192 216             }
193 217
```

```
                )
+               avg_ep_reward = total_rewards / (episode+1)
+               wandb.log({'average_episode_reward': avg_ep_reward, "global_episode": episode})
194 220
195 221         # returns average episode reward (e.g., to use for tuning learning curves)
196     -       return total_rewards / cfg.algorithm.num_episodes
    222 +       avg_ep_reward = total_rewards / cfg.algorithm.num_episodes
    223 +       wandb.log({'average_episode_reward': avg_ep_reward, "global_episode": episode})
    224 +       return avg_ep_reward
```

59 mbrl/examples/conf/algorithm/dreamer.yaml

```
... ... @@ -0,0 +1,59 @@
  1 + # @package _group_
  2 +
  3 + ####################
  4 + ##### FOR GLEN ####
  5 + ####################
  6 + # This is the config file for the "planning parts" of Dreamer
  7 + ####################
  8 +
  9 + name: "dreamer"
 10 +
 11 + agent:
 12 +   # this calls a Dreamer agent that we made
 13 +   _target_: mbrl.planning.dreamer_wrapper.DreamerAgent #mbrl.third_party.pytorch_sac.agent.sac.SACAgent
 14 +   env: ${overrides.environment}
 15 +
 16 + num_initial_trajectories: 5
 17 + action_noise_std: 0.3
 18 + test_frequency: 25
 19 + num_episodes: 1000
 20 + dataset_size: 1000000
 21 +
 22 +   # these are from an SAC implmentation that we may use
 23 +   # from mbrl.planning.sac_wrapper import SACAgent
 24 +   # obs_dim: ??? # to be specified later
 25 +   # action_dim: ??? # to be specified later
 26 +   # action_range: ??? # to be specified later
 27 +   # device: ${device}
 28 +
 29 + #   critic_cfg: ${algorithm.double_q_critic}
 30 + #   actor_cfg: ${algorithm.diag_gaussian_actor}
 31 + #   discount: 0.99
 32 + #   init_temperature: 0.1
 33 + #   alpha_lr: ${overrides.sac_alpha_lr}
 34 + #   alpha_betas: [0.9, 0.999]
 35 + #   actor_lr: ${overrides.sac_actor_lr}
 36 + #   actor_betas: [0.9, 0.999]
 37 + #   actor_update_frequency: ${overrides.sac_actor_update_frequency}
 38 + #   critic_lr: ${overrides.sac_critic_lr}
 39 + #   critic_betas: [0.9, 0.999]
 40 + #   critic_tau: 0.005
 41 + #   critic_target_update_frequency: ${overrides.sac_critic_target_update_frequency}
 42 + #   batch_size: 256
 43 + #   learnable_temperature: true
 44 + #   target_entropy: ${overrides.sac_target_entropy}
 45 +
 46 + # double_q_critic:
 47 + #   _target_: mbrl.third_party.pytorch_sac.agent.critic.DoubleQCritic
 48 + #   obs_dim: ${algorithm.agent.obs_dim}
 49 + #   action_dim: ${algorithm.agent.action_dim}
 50 + #   hidden_dim: 1024
 51 + #   hidden_depth: ${overrides.sac_hidden_depth}
 52 +
 53 + # diag_gaussian_actor:
 54 + #   _target_: mbrl.third_party.pytorch_sac.agent.actor.DiagGaussianActor
 55 +
```

```
#   obs_dim: ${algorithm.agent.obs_dim}
+ #   action_dim: ${algorithm.agent.action_dim}
+ #   hidden_depth: ${overrides.sac_hidden_depth}
+ #   hidden_dim: 1024
+ #   log_std_bounds: [-5, 2]
```

37 mbrl/examples/conf/dynamics_model/dreamer.yaml

```
@@ -0,0 +1,37 @@
+ # @package _group_
+ ####################
+ ##### FOR GLEN ####
+ ####################
+ # This is the config file for the "model parts" of Dreamer. Again this mostly
+ # borrows settings from the MBRL-Lib PlaNet and the recommended config from
+ # Chandramouli Rajagopalan's pytorch implementation of Dreamer.
+ ####################
+ _target_: mbrl.models.DreamerModel
+ obs_shape: [3, 64, 64]
+ # obs_encoding_size: 1024
+ # encoder_config:
+ #   - [3, 32, 4, 2]
+ #   - [32, 64, 4, 2]
+ #   - [64, 128, 4, 2]
+ #   - [128, 256, 4, 2]
+ # decoder_config:
+ #   - [1024, 1, 1]
+ #   - - [1024, 128, 5, 2]
+ #     - [128, 64, 5, 2]
+ #     - [64, 32, 6, 2]
+ #     - [32, 3, 6, 2]
+ action_size: ???
+ hidden_size_fcs: 400
+ depth_size: 32
+ stoch_size: 30
+ deter_size: 200
+ # belief_size: 200
+ # latent_state_size: 30
+ device: ${device}
+ min_std: 0.1
+ # free_nats: 3.0
+ # kl_scale: 1.0
+ # # grad_clip_norm: 1000.0
+ # # To use for duckietown_gym_env
+ # grad_clip_norm: 10.0
+ # env: ${overrides.env}
```

42 mbrl/examples/conf/overrides/dreamer_cheetah_run.yaml

```
@@ -0,0 +1,42 @@
+ # @package _group_
+ ####################
+ ##### FOR GLEN ####
+ ####################
+ # This is the config file to run Dreamer on Cheetah Run. For now we mostly just
+ # use the same values as PlaNet.
+ ####################
+ env: "dmcontrol_cheetah_run"  # used to set the hydra dir, ignored otherwise
+
+ env_cfg:
+   _target_: "mbrl.third_party.dmc2gym.wrappers.DMCWrapper"
+   domain_name: "cheetah"
+   task_name: "run"
+   task_kwargs:
+     random: ${seed}
+   visualize_reward: false
+   from_pixels: true
+   height: 64
```

```
+   width: 64
+   frame_skip: 4
+   bit_depth: 5
+
+ term_fn: "no_termination"
+
+ # General configuration overrides
+ trial_length: 250
+ action_noise_std: 0.3
+
+ # Model overrides
+ num_grad_updates: 100
+ sequence_length: 50
+ batch_size: 50
+ free_nats: 3
+ kl_scale: 1.0
+
+ # Planner configuration overrides
+ planning_horizon: 12
+ cem_num_iters: 10
+ cem_elite_ratio: 0.1
+ cem_population_size: 1000
+ cem_alpha: 0.0
+ cem_clipped_normal: true
```

32 mbrl/examples/conf/overrides/dreamer_duckietown.yaml

```
@@ -0,0 +1,32 @@
+ # @package _group_
+ ####################
+ ##### FOR GLEN ####
+ ####################
+ # This is the config file to run Dreamer on Duckietown. For now we mostly just
+ # use the same values as PlaNet.
+ ####################
+ env: "duckietown_gym_env"  # used to set the hydra dir, ignored otherwise
+
+ # These settings are based on the cheetah run yaml file
+
+ # General configuration overrides
+ trial_length: 250
+ action_noise_std: 0.3
+
+ # Model overrides
+ num_grad_updates: 100
+ sequence_length: 50
+ batch_size: 50
+ free_nats: 3
+ kl_scale: 1.0
+
+ # Planner configuration overrides
+ planning_horizon: 12
+ cem_num_iters: 10
+ cem_elite_ratio: 0.1
+ cem_population_size: 1000
+ cem_alpha: 0.0
+ cem_clipped_normal: true
+
+ #Ali's overrides
+ model_learning_rate: 1e-4
```

3 mbrl/examples/conf/overrides/planet_duckietown.yaml

```
  cem_population_size: 1000
  cem_alpha: 0.0
  cem_clipped_normal: true
+
+ #Ali's overrides
```

```
+ model_learning_rate: 1e-4
```

3 mbrl/examples/main.py

```
 import mbrl.algorithms.mbpo as mbpo
 import mbrl.algorithms.pets as pets
 import mbrl.algorithms.planet as planet
+import mbrl.algorithms.dreamer as dreamer #added April 2022 for project
 import mbrl.util.env

 import pandas as pd
         return mbpo.train(env, test_env, term_fn, cfg)
     if cfg.algorithm.name == "planet":
         return planet.train(env, cfg)
+    if cfg.algorithm.name == "dreamer": #added for project
+        return dreamer.train(env, cfg)


 if __name__ == "__main__":
```

1 mbrl/models/__init__.py

```
 from .model_trainer import ModelTrainer
 from .one_dim_tr_model import OneDTransitionRewardModel
 from .planet import PlaNetModel
+from .dreamer import DreamerModel
 from .util import (
     Conv2dDecoder,
     Conv2dEncoder,
```

723 mbrl/models/dreamer.py

```
@@ -0,0 +1,723 @@
+ ####################
+ ##### FOR GLEN ####
+ ####################
+ # This file holds the bulk of the Dreamer implementation. Most of the code was
+ # inspired from two public implementations. The original Dreamer author's tf
+ # implementation here https://github.com/danijar/dreamer and Chandramouli
+ # Rajagopalan's pytorch implementation of Dreamer here
+ # https://github.com/chamorajg/pl-dreamer.
+ ####################
+ from dataclasses import dataclass
+ from typing import Any, Dict, List, Optional, Tuple, Union
+
+ import numpy as np
+ import torch
+ import torch.distributions
+ import torch.nn as nn
+ import torch.nn.functional as F
+
+ from mbrl.types import TensorType, TransitionBatch
+
+ from .model import Model
+ from .util import Conv2dDecoder, Conv2dEncoder, to_tensor #From mbrl-lib.PlaNet
+
+ from PIL import Image
+
+ import wandb
+ from collections import Iterable
+ import omegaconf
+
+ from pathlib import Path
+ import cv2
+
+ from mbrl.models.planet import PlaNetModel #will need ActionDecoder and DenseModel
+ #from mbrl.models.action import ActionDecoder
+ #from mbrl.models.dense import DenseModel
```

```
+ #https://github.com/chamorajg/pl-dreamer/blob/main/dreamer.py
+ #https://github.com/juliusfrost/dreamer-pytorch/blob/master/dreamer/models/dense.py
+
+ #For Dreamer implementation, Dreamer trainer uses Pytorch Lightning
+ from tqdm import tqdm
+ from typing import Callable, Iterator, Tuple
+ from torch.optim import Adam
+ from torch.utils.data import DataLoader, IterableDataset
+ from torch.distributions import Categorical, Normal
+ from mbrl.models.planet_imp import PLANet, FreezeParameters
+ from mbrl.models.planet_legacy import Episode, DMControlSuiteEnv
+
+ def flatten_config(cfg, curr_nested_key):
+     """The nested config file provided by Hydra cannot be parsed by wandb. This recursive function flattens the
+
+     Args:
+         cfg (Hydra config): The nested config file used by Hydra.
+         curr_nested_key (str): The current parent key (used for recursive calls).
+
+     Returns:
+         (dict): A flatt configuration dictionary.
+     """
+
+     flat_cfg = {}
+
+     for curr_key in cfg.keys():
+
+         # deal with missing values
+         try:
+             curr_item = cfg[curr_key]
+         except Exception as e:
+             curr_item = 'NA'
+
+         # deal with lists
+         if type(curr_item) == list or type(curr_item) == omegaconf.listconfig.ListConfig:
+             for nested_idx, nested_item in enumerate(curr_item):
+                 list_nested_key = f"{curr_nested_key}>{curr_key}>{nested_idx}"
+                 flat_cfg[list_nested_key] = nested_item
+
+         # check if item is also a config
+         # recurse
+         elif isinstance(curr_item, Iterable) and type(curr_item) != str:
+             flat_cfg.update(flatten_config(curr_item, f"{curr_nested_key}>{curr_key}"))
+
+         # otherwise just add to return dict
+         else:
+             flat_cfg[f"{curr_nested_key}>{curr_key}"] = curr_item
+
+     return flat_cfg
+
+ class ExperienceSourceDataset(IterableDataset):
+     """
+     Implementation from PyTorch Lightning Bolts:
+     https://github.com/PyTorchLightning/pytorch-lightning-bolts/blob/master/pl_bolts/datamodules/experience_sour
+     Basic experience source dataset. Takes a generate_batch function that returns an iterator.
+     The logic for the experience source and how the batch is generated is defined the Lightning model itself
+     """
+
+     def __init__(self, generate_batch: Callable):
+         self.generate_batch = generate_batch
+
+     def __iter__(self) -> Iterator:
+         iterator = self.generate_batch
+         return iterator
+
+ # class DreamerModel(nn.Module):
+ class DreamerModel(Model):
+
```

```
+    def __init__(
+        self,
+        obs_shape,
+        action_size,
+        hidden_size_fcs,
+        depth_size,
+        stoch_size,
+        deter_size,
+        device: Union[str, torch.device],
+        min_std,
+    ):
+        # This config is needed for now as we figure out
+        # what parameters we need to run dreamer
+        # we get values from it from time to time
+
+        outside_config = {
+            'name': 'Dreamer',
+            'env': 'quadruped_run',
+            'seed': 42,
+            'ckpt_callback': {
+                'save_top_k': 2,
+                'mode': 'min',
+                'monitor': 'loss',
+                'save_on_train_epoch_end': True,
+                'save_last': True,
+                'trainer_params': None,
+                'default_root_dir': 'None',
+                'gpus': 1,
+                'gradient_clip_val': 100.0,
+                'val_check_interval': 5,
+                'max_epochs': 1000,
+                },
+            'dreamer': {
+                'td_model_lr': 0.0005,
+                'actor_lr': 8e-05,
+                'critic_lr': 8e-05,
+                'default_lr': 0.0005,
+                'weight_decay': 1e-06,
+                'batch_size': 50,
+                'batch_length': 50,
+                'length': 50,
+                'prefill_timesteps': 5000,
+                'explore_noise': 0.3,
+                'max_episode_length': 1000,
+                'collect_interval': 100,
+                'max_experience_size': 1000,
+                'save_episodes': False,
+                'discount': 0.99,
+                'lambda': 0.95,
+                'clip_actions': False,
+                'horizon': 1000,
+                'imagine_horizon': 15,
+                'free_nats': 3.0,
+                'kl_coeff': 1.0,
+                'dreamer_model': {
+                    'obs_space': [3, 64, 64],
+                    'num_outputs': 1,
+                    'custom_model': 'DreamerModel',
+                    'deter_size': 200,
+                    'stoch_size': 30,
+                    'depth_size': 32,
+                    'hidden_size': 400,
+                    'action_init_std': 5.0,
+                    },
+                'env_config': {'action_repeat': 2},
+                },
+            }
+
```

```
+
+        super().__init__(device)
+        self.outside_config = outside_config
+
+        final_dreamer_config = {"DC" : outside_config}
+        flat_cfg = flatten_config(final_dreamer_config, ">")
+        for config_item in flat_cfg:
+            wandb.config[config_item] = flat_cfg[config_item]
+
+        self.obs_shape = obs_shape
+        self.hidden_size_fcs = hidden_size_fcs
+        self.device = device
+        self.num_outputs = 1
+
+        self.model_config = {
+            "hidden_size": self.hidden_size_fcs,
+            'deter_size': deter_size,
+            'stoch_size': stoch_size,
+            'depth_size': depth_size,
+            'action_init_std': min_std,
+            }
+        self.name = 'Dreamer'
+
+
+    def setGymEnv(self, env, workdir):
+        self.env = env
+        sample_action_space = np.zeros(self.env.action_space.shape)
+        # In the future we may want to use the MBRL-Lib
+        # implmentation of Planet here
+
+        # self.model = PlaNetModel #try this when we get the functions mapped properly
+
+        self.model = PLANet(obs_space= self.obs_shape,
+                            action_space= sample_action_space,
+                            num_outputs= self.num_outputs,
+                            model_config= self.model_config,
+                            name = self.name,
+                            device = self.device
+                            )
+        self.episodes = []
+        self.length = self.outside_config["dreamer"]['length']
+        self.timesteps = 0
+        self._max_experience_size = self.outside_config["dreamer"]['max_experience_size']
+        self._action_repeat = self.outside_config["dreamer"]["env_config"]["action_repeat"]
+        self._prefill_timesteps = self.outside_config["dreamer"]["prefill_timesteps"]
+        self._max_episode_length = self.outside_config["dreamer"]["max_episode_length"]
+
+        self.explore = self.outside_config["dreamer"]['explore_noise']
+        self.batch_size = self.outside_config["dreamer"]["batch_size"]
+        self.action_space = sample_action_space.shape[0]
+        self.imagine_horizon = self.outside_config['dreamer']["imagine_horizon"]
+        prefill_episodes = self._prefill_train_batch()
+        self._add(prefill_episodes)
+        self.workdir = workdir
+        self.curr_episode = 0
+        self.currently_testing = False
+        self.video_counter = 0
+        self.in_duckietown = False
+
+
+
+    # functions we added to make this work with MBRL-Lib
+    def set_curr_episode(self, episode):
+        self.curr_episode = episode
+
+    def set_currently_testing(self, currently_testing):
+        self.currently_testing = currently_testing
+
```

```
+
+    def reset_world_model(self, device=None):
+        self.model.get_initial_state(device=device)
+
+    def _process_pixel_obs(self, obs: torch.Tensor) -> torch.Tensor:
+        return to_tensor(obs).float().to(self.device) / 256.0 - 0.5
+
+    def _process_batch(
+        self, batch: TransitionBatch, as_float: bool = True, pixel_obs: bool = False
+    ) -> Tuple[torch.Tensor, ...]:
+        # `obs` is a sequence, so `next_obs` is not necessary
+        # sequence iterator samples full sequences, so `dones` not necessary either
+        obs, action, _, rewards, _ = super()._process_batch(batch, as_float=as_float)
+        if pixel_obs:
+            obs = self._process_pixel_obs(obs)
+        return obs, action, rewards
+
+    def compute_dreamer_loss(self,
+                             obs,
+                             action,
+                             reward,
+                             imagine_horizon,
+                             discount=0.99,
+                             lambda_=0.95,
+                             kl_coeff=1.0,
+                             free_nats=3.0,
+                             log=True):
+        """Constructs loss for the Dreamer objective
+            Args:
+                obs (TensorType): Observations (o_t)
+                action (TensorType): Actions (a_(t-1))
+                reward (TensorType): Rewards (r_(t-1))
+                model (TorchModelV2): DreamerModel, encompassing all other models
+                imagine_horizon (int): Imagine horizon for actor and critic loss
+                discount (float): Discount
+                lambda_ (float): Lambda, like in GAE
+                kl_coeff (float): KL Coefficient for Divergence loss in model loss
+                free_nats (float): Threshold for minimum divergence in model loss
+                log (bool): If log, generate gifs
+            """
+        encoder_weights = list(self.model.encoder.parameters())
+        decoder_weights = list(self.model.decoder.parameters())
+        reward_weights = list(self.model.reward.parameters())
+        dynamics_weights = list(self.model.dynamics.parameters())
+        critic_weights = list(self.model.value.parameters())
+        model_weights = list(encoder_weights + decoder_weights + reward_weights +
+                             dynamics_weights)
+
+        device = self.device
+        # PlaNET Model Loss
+        latent = self.model.encoder(obs)
+        istate = self.model.dynamics.get_initial_state(obs.shape[0], self.device)
+        post, prior = self.model.dynamics.observe(latent, action, istate)
+        features = self.model.dynamics.get_feature(post)
+        image_pred = self.model.decoder(features)
+        reward_pred = self.model.reward(features)
+        image_loss = -torch.mean(image_pred.log_prob(obs))
+        reward_loss = -torch.mean(reward_pred.log_prob(reward))
+        prior_dist = self.model.dynamics.get_dist(prior[0], prior[1])
+        post_dist = self.model.dynamics.get_dist(post[0], post[1])
+        div = torch.mean(
+            torch.distributions.kl_divergence(post_dist, prior_dist).sum(dim=2))
+        div = torch.clamp(div, min=free_nats)
+        model_loss = kl_coeff * div + reward_loss + image_loss
+
+        # Actor Loss
+        with torch.no_grad():
+
```

```
+            actor_states = [v.detach() for v in post]
306 +        with FreezeParameters(model_weights):
307 +            imag_feat = self.model.imagine_ahead(actor_states, imagine_horizon)
308 +        with FreezeParameters(model_weights + critic_weights):
309 +            reward = self.model.reward(imag_feat).mean
310 +            value = self.model.value(imag_feat).mean
311 +        pcont = discount * torch.ones_like(reward)
312 +        returns = self._lambda_return(reward[:-1], value[:-1], pcont[:-1], value[-1],
313 +                                      lambda_)
314 +        discount_shape = pcont[:1].size()
315 +        discount = torch.cumprod(
316 +            torch.cat([torch.ones(*discount_shape).to(device), pcont[:-2]], dim=0),
317 +            dim=0)
318 +        actor_loss = -torch.mean(discount * returns)
319 +
320 +        # Critic Loss
321 +        with torch.no_grad():
322 +            val_feat = imag_feat.detach()[:-1]
323 +            target = returns.detach()
324 +            val_discount = discount.detach()
325 +        val_pred = self.model.value(val_feat)
326 +        critic_loss = -torch.mean(val_discount * val_pred.log_prob(target))
327 +
328 +        # Logging purposes
329 +        prior_ent = torch.mean(prior_dist.entropy())
330 +        post_ent = torch.mean(post_dist.entropy())
331 +
332 +        log_gif = None
333 +
334 +        if log:
335 +            log_gif = self._log_summary(obs, action, latent, image_pred)
336 +
337 +        return_dict = {
338 +            "model_loss": model_loss,
339 +            "reward_loss": reward_loss,
340 +            "image_loss": image_loss,
341 +            "divergence": div,
342 +            "actor_loss": actor_loss,
343 +            "critic_loss": critic_loss,
344 +            "prior_ent": prior_ent,
345 +            "post_ent": post_ent,
346 +        }
347 +
348 +        if log_gif is not None:
349 +            return_dict["log_gif"] = self._postprocess_gif(log_gif)
350 +        return return_dict
351 +
352 +    # Loss function for dreamer
353 +    # Different from Planet cause more networks
354 +    def loss(
355 +            self,
356 +            batch: TransitionBatch,
357 +            target: Optional[torch.Tensor] = None,
358 +            reduce: bool = True,
359 +        ) -> Tuple[torch.Tensor, Dict[str, Any]]:
360 +            """Computes the Dreamer loss given a batch of transitions.
361 +
362 +            """
363 +            obs, action, rewards = self._process_batch(batch, pixel_obs=True)
364 +
365 +            # hacky check to see if we are using the duckietown environment
366 +            # I just see if the obs shape ends in 3, for the other envs it does not
367 +            if obs.shape[-1] == 3:
368 +                self.in_duckietown = True
369 +
370 +            if self.in_duckietown:
371 +                obs = obs.permute((0,1,4,2,3))
372 +
```

```
+
+            return_dict = self.compute_dreamer_loss(obs, action, rewards, self.imagine_horizon)
+
+            dreamer_obs_loss = return_dict["image_loss"]
+            dreamer_reward_loss = return_dict["reward_loss"]
+            dreamer_kl_loss = return_dict["divergence"]
+
+            meta = {
+                "reconstruction": None,
+                "observations_loss": dreamer_obs_loss.detach().mean().item(),
+                "reward_loss": dreamer_reward_loss.detach().mean().item(),
+                "kl_loss": dreamer_kl_loss.detach().mean().item(),
+            }
+
+            return return_dict["model_loss"], meta
+
+    def eval_score(
+        self, batch: TransitionBatch, target: Optional[torch.Tensor] = None
+    ) -> Tuple[torch.Tensor, Dict[str, Any]]:
+        """Computes an evaluation score for the model over the given input/target.
+
+        This is equivalent to calling loss(batch, reduce=False)`.
+        """
+        with torch.no_grad():
+            return self.loss(batch, reduce=False)
+
+    # again, Dreamer update is fundamentally different from
+    # from planet update, which causes problems when trying
+    # to integrate it into this library
+    def dreamer_update(self, dreamer_loss):
+
+        self.dreamer_optim.zero_grad()
+        dreamer_loss.backward()
+        self.dreamer_optim.step()
+
+        return dreamer_loss
+
+    def dreamer_loss(self, train_batch):
+        """ calculates dreamer loss."""
+
+        log_gif = False
+        if "log_gif" in train_batch:
+            log_gif = True
+
+        self.stats_dict = self.compute_dreamer_loss(
+            train_batch["obs"],
+            train_batch["actions"],
+            train_batch["rewards"],
+            self.outside_config["dreamer"]["imagine_horizon"],
+            self.outside_config["dreamer"]["discount"],
+            self.outside_config["dreamer"]["lambda"],
+            self.outside_config["dreamer"]["kl_coeff"],
+            self.outside_config["dreamer"]["free_nats"],
+            log_gif,
+        )
+
+        loss_dict = self.stats_dict
+        return loss_dict
+
+    def _prefill_train_batch(self, ):
+        """ Prefill episodes before the training begins."""
+
+        self.timesteps = 2
+        obs = self.env.reset()
+        episode = Episode(obs, self.action_space)
+        episodes = []
+
+        while self.timesteps < self._prefill_timesteps:
```

```
441 +            action, logp, state = self.prefill_action_sampler_fn(None,
442 +                                                                 self.timesteps)
443 +            action = action.squeeze()
444 +            obs, reward, done, _ = self.env.step(action.numpy())
445 +            episode.append((obs, action, reward, done))
446 +            self.timesteps += self._action_repeat
447 +            if done or self.timesteps == self._prefill_timesteps - 1:
448 +                episodes.append(episode.todict())
449 +                obs = self.env.reset()
450 +                if done:
451 +                    episode.reset(obs)
452 +        del episode
453 +        return episodes
454 +
455 +    def _data_collect(self):
456 +        """ Collect data from the policy after every epoch. """
457 +
458 +        obs = self.env.reset()
459 +        state = self.model.get_initial_state(self.device)
460 +        episode = Episode(obs, self.action_space)
461 +        episodes = []
462 +
463 +        max_len = self._max_episode_length // self._action_repeat
464 +        for i in range(max_len):
465 +            action, logp, state = self.action_sampler_fn(
466 +                    ((episode.obs[-1] / 255.0) - 0.5).unsqueeze(0).to(
467 +                    self.device), state, self.explore, False)
468 +            obs, reward, done, _ = self.env.step(action.detach().cpu().numpy())
469 +            episode.append((obs, action.detach().cpu(), reward, done))
470 +            if done or i == max_len - 1:
471 +                episodes.append(episode.todict())
472 +                break
473 +        del episode
474 +        return episodes
475 +
476 +    def _test(self):
477 +        """ Test the model after every few intervals."""
478 +
479 +        obs = self.env.reset()
480 +        state = self.model.get_initial_state(self.device)
481 +        obs = torch.FloatTensor(np.ascontiguousarray(obs.transpose((2, 0, 1))))
482 +
483 +        tot_reward = 0
484 +        done = False
485 +        while not done:
486 +            action, logp, state = self.action_sampler_fn(
487 +                        ((obs / 255.0) - 0.5).unsqueeze(0).to(self.device), state, self.explore, True)
488 +            obs, reward, done, _ = self.env.step(action.detach().cpu().numpy())
489 +            obs = obs.transpose((2, 0, 1))
490 +            obs = torch.FloatTensor(np.ascontiguousarray(obs))
491 +            tot_reward += reward
492 +        return tot_reward
493 +
494 +    def _add(self, batch):
495 +        """ Adds the collected episode samples as well as the prefilled
496 +            episode samples into the episode memory."""
497 +
498 +        self.episodes.extend(batch)
499 +
500 +        if len(self.episodes) > self._max_experience_size:
501 +            remove_episode_index = len(self.episodes) -\
502 +                                       self._max_experience_size
503 +            self.episodes = self.episodes[remove_episode_index:]
504 +
505 +        if self.outside_config["dreamer"]["save_episodes"] and\
506 +            self.trainer is not None and self.trainer.log_dir is not None:
507 +            save_episodes = np.array(self.episodes)
508
```

```
+            if not os.path.exists(f'{self.trainer.log_dir}/episodes'):
509 +                os.makedirs(f'{self.trainer.log_dir}/episodes', exist_ok=True)
510 +            np.savez(f'{self.trainer.log_dir}/episodes/episodes.npz', save_episodes)
511 +
512 +    def _sample(self, batch_size):
513 +        """ Samples a batch of episode of length T from the config."""
514 +
515 +        episodes_buffer = []
516 +        while len(episodes_buffer) < batch_size:
517 +            rand_index = random.randint(0, len(self.episodes) - 1)
518 +            episode = self.episodes[rand_index]
519 +            if episode["count"] < self.length:
520 +                continue
521 +            available = episode["count"] - self.length
522 +            index = int(random.randint(0, available))
523 +            episodes_buffer.append({"count": self.length,
524 +                                    "obs": episode["obs"][index : index + self.length],
525 +                                    "action": episode["action"][index: index + self.length],
526 +                                    "reward": episode["reward"][index: index + self.length],
527 +                                    "done": episode["done"][index: index + self.length],
528 +                                    })
529 +        total_batch = {}
530 +        for k in episodes_buffer[0].keys():
531 +            if k == "count" or k == "state":
532 +                continue
533 +            else:
534 +                total_batch[k] = torch.stack([e[k] for e in episodes_buffer], axis=0)
535 +        return total_batch
536 +
537 +    def _train_batch(self, batch_size):
538 +        for _ in range(self.outside_config["dreamer"]["collect_interval"]):
539 +            total_batch = self._sample(batch_size)
540 +            def return_batch(i):
541 +                return (total_batch["obs"][i] / 255.0 - 0.5),\
542 +                    total_batch["action"][i], total_batch["reward"][i], total_batch["done"][i]
543 +            for i in range(batch_size):
544 +                yield return_batch(i)
545 +
546 +    def prefill_action_sampler_fn(self, state, timestep):
547 +        """Action sampler function during prefill phase where
548 +        actions are sampled uniformly [-1, 1].
549 +        """
550 +        # Custom Exploration
551 +        logp = [0.0]
552 +        # Random action in space [-1.0, 1.0]
553 +        action = torch.FloatTensor(1, self.model.action_size).uniform_(-1.0,
554 +                                                1.0)
555 +        state = self.model.get_initial_state(self.device)
556 +        return action, logp, state
557 +
558 +    def action_sampler_fn(self, obs, state, explore, test=False):
559 +        """Action sampler during training phase, actions
560 +        are evaluated through DreamerPolicy and
561 +        an additive gaussian is added
562 +        to incentivize exploration."""
563 +
564 +        action, logp, state_new = self.model.policy(obs, state,
565 +                                    explore=not(test))
566 +        if not test:
567 +            action = Normal(action, explore).sample()
568 +        action = torch.clamp(action, min=-1.0, max=1.0)
569 +        return action, logp, state_new
570 +
571 +    def training_step(self, batch, batch_idx):
572 +        """ Trains the model on the samples collected."""
573 +
574 +        obs, action, reward, __ = batch
575 +
```

```
+         loss = self.dreamer_loss({"obs":obs,
576 +                         "actions":action, "rewards":reward,
577 +                         "log_gif": True})
578 +         outputs = []
579 +         for k, v in loss.items():
580 +             if "loss" in k:
581 +                 self.log(k, v)
582 +             if k in ["model_loss", "critic_loss", "actor_loss"]:
583 +                 outputs.append(v)
584 +         return sum(outputs)
585 + 
586 +     def training_epoch_end(self, outputs):
587 +         """ Collects data samples after every epoch end and tests the
588 +             model on the environment of maximum length from the config every
589 +             few intervals."""
590 + 
591 +         total_loss = 0
592 +         for out in outputs:
593 +             total_loss += out['loss'].item()
594 +         if len(outputs) != 0:
595 +             total_loss /= len(outputs)
596 +         self.log('loss', total_loss)
597 + 
598 +         with torch.no_grad():
599 +             data_collection_episodes = self._data_collect()
600 +             self._add(data_collection_episodes)
601 +             data_dict = data_collection_episodes[0]
602 +             self.log('avg_reward_collection', torch.mean(data_dict['reward']))
603 + 
604 +         if self.current_epoch > 0 and \
605 +                 self.current_epoch % self.outside_config["trainer_params"]["val_check_interval"] == 0:
606 +             self.model.eval()
607 +             episode_reward = self._test()
608 +             self.log('avg_reward_test', episode_reward)
609 +             self.model.train()
610 + 
611 +     def _collate_fn(self, batch):
612 +         return_batch = {}
613 +         for k in batch[0].keys():
614 +             if k == 'count':
615 +                 return_batch[k] = torch.LongTensor([data[k] for data in batch])
616 +             return_batch[k] = torch.stack([data[k] for data in batch])
617 +         return return_batch
618 + 
619 +     def train_dataloader(self) -> DataLoader:
620 +         """Get train loader"""
621 +         dataset = ExperienceSourceDataset(self._train_batch(self.batch_size))
622 +         dataloader = DataLoader(dataset=dataset,
623 +                                 batch_size=self.batch_size,
624 +                                 pin_memory=True,
625 +                                 num_workers=1)
626 +         return dataloader
627 + 
628 +     def configure_optimizers(self,):
629 +         """ Configure optmizers."""
630 + 
631 +         encoder_weights = list(self.model.encoder.parameters())
632 +         decoder_weights = list(self.model.decoder.parameters())
633 +         reward_weights = list(self.model.reward.parameters())
634 +         dynamics_weights = list(self.model.dynamics.parameters())
635 +         actor_weights = list(self.model.actor.parameters())
636 +         critic_weights = list(self.model.value.parameters())
637 + 
638 +         model_opt = Adam(
639 +             [
640 +             {'params': encoder_weights + decoder_weights + reward_weights + dynamics_weights,
641 +             'lr':self.outside_config["dreamer"]["td_model_lr"]},
642 
```

```
+            {'params':actor_weights, 'lr':self.outside_config["dreamer"]["actor_lr"]},
643 +            {'params':critic_weights, 'lr':self.outside_config["dreamer"]["critic_lr"]}],
644 +            lr=self.outside_config["dreamer"]["default_lr"],
645 +            weight_decay=self.outside_config["dreamer"]["weight_decay"])
646 +        self.dreamer_optim = model_opt
647 +        return model_opt
648 +
649 +    def _postprocess_gif(self, gif: np.ndarray):
650 +        gif = gif.detach().cpu().numpy()
651 +        gif = np.clip(255*gif, 0, 255).astype(np.uint8)
652 +        B, T, C, H, W = gif.shape
653 +        frames = gif.transpose((1, 2, 3, 0, 4)).reshape((1, T, C, H, B * W))
654 +        frames = frames.squeeze(0)
655 +
656 +        def display_image(frame):
657 +            frame = frame.transpose((1, 2, 0))
658 +            return Image.fromarray(frame)
659 +
660 +        # create destination path for movies
661 +        movie_save_folder = f'{self.workdir}/movies'
662 +        Path(movie_save_folder).mkdir(parents=True, exist_ok=True)
663 +        video_name = f"movie_e{self.curr_episode}_t{int(self.currently_testing)}_c{self.video_counter}"
664 +
665 +        self.video_counter += 1
666 +        # # create a videowriter
667 +        # videodims = (frames.shape[-2], frames.shape[-1])
668 +        # fourcc = cv2.VideoWriter_fourcc(*'MPEG')
669 +        # video = cv2.VideoWriter(f"{movie_save_folder}/{video_name}.mp4",fourcc, 10,videodims)
670 +
671 +        # # write to video
672 +        # for frame in list(frames):
673 +        #     curr_img_frame = display_image(frame)
674 +        #     video.write(cv2.cvtColor(np.array(curr_img_frame), cv2.COLOR_RGB2BGR))
675 +        # video.release()
676 +
677 +        if self.video_counter<200 or self.video_counter%50:
678 +            # also save gif
679 +            img, *imgs = [display_image(frame) for frame in list(frames)]
680 +            # img.save(f'{self.trainer.log_dir}/movies/movie_{self.current_epoch}.gif', format='GIF', append_ima
681 +            #  save_all=True, loop=0)
682 +
683 +            img_save_loc = f'{movie_save_folder}/{video_name}.gif'
684 +            img.save(img_save_loc, format='GIF', append_images=imgs,
685 +            save_all=True, loop=0)
686 +
687 +            image_array = [display_image(frame) for frame in list(frames)]
688 +            # images = wandb.Image(image_array, caption=f"{video_name}")
689 +            wandb_saved_gif = wandb.Image(img_save_loc)
690 +            wandb.log({"video": wandb_saved_gif, "global_episode": self.curr_episode})
691 +
692 +        return frames
693 +
694 +    def _log_summary(self, obs, action, embed, image_pred):
695 +        truth = obs[:6] + 0.5
696 +        recon = image_pred.mean[:6]
697 +        istate = self.model.dynamics.get_initial_state(6, self.device)
698 +        init, _ = self.model.dynamics.observe(embed[:6, :5],
699 +                                              action[:6, :5], istate)
700 +        init = [itm[:6, -1] for itm in init]
701 +        prior = self.model.dynamics.imagine(action[:6, 5:], init)
702 +        openl = self.model.decoder(self.model.dynamics.get_feature(prior)).mean
703 +
704 +        mod = torch.cat([recon[:, :5] + 0.5, openl + 0.5], 1)
705 +        error = (mod - truth + 1.0) / 2.0
706 +        return torch.cat([truth, mod, error], 3)
707 +
708 +    def _lambda_return(self, reward, value, pcont, bootstrap, lambda_):
709
```

```
+         def agg_fn(x, y):
710 +             return y[0] + y[1] * lambda_ * x
711 +
712 +         next_values = torch.cat([value[1:], bootstrap[None]], dim=0)
713 +         inputs = reward + pcont * next_values * (1 - lambda_)
714 +
715 +         last = bootstrap
716 +         returns = []
717 +         for i in reversed(range(len(inputs))):
718 +             last = agg_fn(last, [inputs[i], pcont[i]])
719 +             returns.append(last)
720 +
721 +         returns = list(reversed(returns))
722 +         returns = torch.stack(returns, dim=0)
723 +         return returns
```

12 mbrl/models/model.py

```
124 124                  from strings to objects with metadata computed by the model
125 125                  (e.g., reconstructions, entropy, etc.) that will be used for logging.
126 126              """
127 +     # Changes to model class because Dreamer is different
128 +     def dreamer_update(self):
129 +         return None
127 130
128 131      def update(
129 132          self,
153 156               (dict): any additional metadata dictionary computed by :meth:`loss`.
154 157          """
155 158          self.train()
156     -         optimizer.zero_grad()
159 +         # Changes to model class because Dreamer is different
160 +         # optimizer.zero_grad()
157 161          loss, meta = self.loss(model_in, target)
158     -         loss.backward()
162 +         # loss.backward()
159 163          if meta is not None:
160 164              with torch.no_grad():
161 165                  grad_norm = 0.0
162 166                  for p in list(filter(lambda p: p.grad is not None, self.parameters())):
163 167                      grad_norm += p.grad.data.norm(2).item() ** 2
164 168                  meta["grad_norm"] = grad_norm
165     -         optimizer.step()
169 +         # optimizer.step()
170 +         # Changes to model class because Dreamer is different
171 +         loss = self.dreamer_update(loss)
166 172          return loss.item(), meta
167 173
168 174      def reset(
```

559 mbrl/models/planet_imp.py

```
... ...   @@ -0,0 +1,559 @@
  1 + ####################
  2 + ##### FOR GLEN ####
  3 + ####################
  4 + # This implementation of PlaNet servers as the world model for our Dreamer
  5 + # implementation.
  6 + ####################
  7 + import torch
  8 + import torch.nn as nn
  9 + import torch.nn.functional as F
 10 +
 11 + ####################
 12 + ##### FOR GLEN ####
 13 + ####################
 14 + # This is the PlaNet implementation we ported from Chandramouli Rajagopalan's
```

```
+ # pytorch implementation of Dreamer here https://github.com/chamorajg/pl-dreamer
+ # which has helper functions for the non-MBRL-Lib PlaNet, and all of the networks
+ # (Action and Value Nets as well) and are siloed from the dependencies for QA
+ # of this version of PlaNet outside of MBRL-Lib PlaNet.
+ ####################
+ import torch.distributions as td
+
+ import numpy as np
+
+ from typing import Any, List, Tuple, Optional
+ TensorType = Any
+
+ def atanh(x):
+     return 0.5 * torch.log((1 + x) / (1 - x))
+
+ class TanhBijector(torch.distributions.Transform):
+     def __init__(self):
+         super().__init__()
+         self.bijective = True
+         self.domain = torch.distributions.constraints.real
+         self.codomain = torch.distributions.constraints.interval(-1.0, 1.0)
+
+     @property
+     def sign(self): return 1.
+
+     def _call(self, x): return torch.tanh(x)
+
+     def _inverse(self, y: torch.Tensor):
+         y = torch.where(
+             (torch.abs(y) <= 1.),
+             torch.clamp(y, -0.99999997, 0.99999997),
+             y)
+         y = atanh(y)
+         return y
+
+     def log_abs_det_jacobian(self, x, y):
+         return 2. * (np.log(2) - x - F.softplus(-2. * x))
+
+ class Reshape(nn.Module):
+
+     def __init__(self, shape: List):
+         super().__init__()
+         self.shape = shape
+
+     def forward(self, x):
+         return x.view(*self.shape)
+
+ class Encoder(nn.Module):
+     """ As mentioned in the paper, VAE is used
+         to calculate the state posterior needed
+         for parameter learning in RSSM. The training objective
+         here is to create bound on the data. Here losses
+         are written only for observations as rewards losses
+         follow them. """
+
+     def __init__(self,
+                     device,
+                     depth : int = 32,
+                     input_channels : Optional[int] = 3
+                     ):
+         super(Encoder, self).__init__()
+         """
+         Initialize the parameters of the Encoder.
+         Args
+             depth (int) : Number of channels in the first convolution layer.
+             input_channels (int) : Number of channels in the input observation.
+         """
+
```

```
+         self.depth = depth
83  +         self.input_channels = input_channels
84  +         self.encoder = nn.Sequential(
85  +                 nn.Conv2d(self.input_channels, self.depth, 4, stride=2),
86  +                 nn.ReLU(),
87  +                 nn.Conv2d(self.depth, self.depth * 2, 4, stride=2),
88  +                 nn.ReLU(),
89  +                 nn.Conv2d(self.depth * 2, self.depth * 4, 4, stride=2),
90  +                 nn.ReLU(),
91  +                 nn.Conv2d(self.depth * 4, self.depth * 8, 4, stride=2),
92  +                 nn.ReLU(),
93  +         )
94  +         self.encoder = self.encoder.to(device)
95  +
96  +     def forward(self, x):
97  +         """ Flatten the input observation [batch, horizon, 3, 64, 64]
98  +             into shape [batch * horizon, 3, 64, 64] before feeding it
99  +             to the input. """
100 +         orig_shape = x.shape
101 +         x = x.reshape(-1, *x.shape[-3:])
102 +         x = self.encoder(x)
103 +         x = x.reshape(*orig_shape[:-3], -1)
104 +         return x
105 +
106 +
107 + class Decoder(nn.Module):
108 +     """
109 +     Takes the input from the RSSM model
110 +     and then decodes it back to images from
111 +     the latent space model. It is mainly used
112 +     in calculating losses.
113 +     """
114 +     def __init__(self, device,
115 +                     input_size : int,
116 +                     depth: int = 32,
117 +                     shape: Tuple[int] = (3, 64, 64)):
118 +         super(Decoder, self).__init__()
119 +         self.depth = depth
120 +         self.shape = shape
121 +         self.decoder = nn.Sequential(
122 +             nn.Linear(input_size, 32 * self.depth),
123 +             Reshape([-1, 32 * self.depth, 1, 1]),
124 +             nn.ConvTranspose2d(32 * self.depth, 4 * self.depth, 5, stride=2),
125 +             nn.ReLU(),
126 +             nn.ConvTranspose2d(4 * self.depth, 2 * self.depth, 5, stride=2),
127 +             nn.ReLU(),
128 +             nn.ConvTranspose2d(2 * self.depth, self.depth, 6, stride=2),
129 +             nn.ReLU(),
130 +             nn.ConvTranspose2d(self.depth, self.shape[0], 6, stride=2),
131 +         )
132 +         self.decoder = self.decoder.to(device)
133 +
134 +     def forward(self, x):
135 +         orig_shape = x.shape
136 +         x = self.decoder(x)
137 +         reshape_size = orig_shape[:-1] + self.shape
138 +         mean = x.view(*reshape_size)
139 +         return td.Independent(  td.Normal(mean, 1), len(self.shape))
140 +
141 +
142 + class ActionDecoder(nn.Module):
143 +     """
144 +     ActionDecoder is the policy module in Dreamer.
145 +
146 +     It outputs a distribution parameterized by mean and std, later to be
147 +     transformed by a custom TanhBijector.
148 +     """
149
```

```
+    def __init__(self,
150 +                 device,
151 +                 input_size: int,
152 +                 action_size: int,
153 +                 layers: int,
154 +                 units: int,
155 +                 dist: str = "tanh_normal",
156 +                 min_std: float = 1e-4,
157 +                 init_std: float = 5.0,
158 +                 mean_scale: float = 5.0):
159 +        super(ActionDecoder, self).__init__()
160 +        self.layrs = layers
161 +        self.units = units
162 +        self.dist = dist
163 +        self.act = nn.ReLU
164 +        self.min_std = min_std
165 +        self.init_std = init_std
166 +        self.mean_scale = mean_scale
167 +        self.action_size = action_size
168 +
169 +        self.layers = []
170 +        self.softplus = nn.Softplus()
171 +
172 +        # MLP Construction
173 +        cur_size = input_size
174 +        for _ in range(self.layrs):
175 +            self.layers.extend([nn.Linear(cur_size, self.units), self.act()])
176 +            cur_size = self.units
177 +        self.layers.append(nn.Linear(cur_size, 2 * action_size))
178 +        self.model = nn.Sequential(*self.layers)
179 +        self.model = self.model.to(device)
180 +
181 +    def forward(self, x):
182 +        raw_init_std = np.log(np.exp(self.init_std) - 1)
183 +        x = self.model(x)
184 +        mean, std = torch.chunk(x, 2, dim=-1)
185 +        mean = self.mean_scale * torch.tanh(mean / self.mean_scale)
186 +        std = self.softplus(std + raw_init_std) + self.min_std
187 +        dist = td.Normal(mean, std)
188 +        transforms = [TanhBijector()]
189 +        dist = td.transformed_distribution.TransformedDistribution(
190 +            dist, transforms)
191 +        dist = td.Independent(dist, 1)
192 +        return dist
193 +
194 +
195 + class DenseDecoder(nn.Module):
196 +    """
197 +    FC network that outputs a distribution for calculating log_prob.
198 +    Used later in DreamerLoss.
199 +    """
200 +
201 +    def __init__(self,
202 +                 device,
203 +                 input_size: int,
204 +                 output_size: int,
205 +                 layers: int,
206 +                 units: int,
207 +                 dist: str = "normal"):
208 +        """Initializes FC network
209 +        Args:
210 +            input_size (int): Input size to network
211 +            output_size (int): Output size to network
212 +            layers (int): Number of layers in network
213 +            units (int): Size of the hidden layers
214 +            dist (str): Output distribution, parameterized by FC output
215 +                logits.
216
```

```
+            act (Any): Activation function
217 +        """
218 +        super().__init__()
219 +        self.layrs = layers
220 +        self.units = units
221 +        self.act = nn.ELU
222 +        self.dist = dist
223 +        self.input_size = input_size
224 +        self.output_size = output_size
225 +        self.layers = []
226 +        cur_size = input_size
227 +        for _ in range(self.layrs):
228 +            self.layers.extend([nn.Linear(cur_size, self.units), self.act()])
229 +            cur_size = units
230 +        self.layers.append(nn.Linear(cur_size, output_size))
231 +        self.model = nn.Sequential(*self.layers)
232 +        self.model = self.model.to(device)
233 +
234 +    def forward(self, x):
235 +        x = self.model(x)
236 +        if self.output_size == 1:
237 +            x = torch.squeeze(x)
238 +        if self.dist == "normal":
239 +            output_dist = td.Normal(x, 1)
240 +        elif self.dist == "binary":
241 +            output_dist = td.Bernoulli(logits=x)
242 +        else:
243 +            raise NotImplementedError("Distribution type not implemented!")
244 +        return td.Independent(output_dist, 0)
245 +
246 +
247 + class RSSM(nn.Module):
248 +    """RSSM is the core recurrent part of the PlaNET module. It consists of
249 +    two networks, one (obs) to calculate posterior beliefs and states and
250 +    the second (img) to calculate prior beliefs and states. The prior network
251 +    takes in the previous state and action, while the posterior network takes
252 +    in the previous state, action, and a latent embedding of the most recent
253 +    observation.
254 +    """
255 +
256 +    def __init__(self,
257 +                 device,
258 +                 action_size: int,
259 +                 embed_size: int,
260 +                 stoch: int = 30,
261 +                 deter: int = 200,
262 +                 hidden: int = 200):
263 +        """Initializes RSSM
264 +        Args:
265 +            action_size (int): Action space size
266 +            embed_size (int): Size of ConvEncoder embedding
267 +            stoch (int): Size of the distributional hidden state
268 +            deter (int): Size of the deterministic hidden state
269 +            hidden (int): General size of hidden layers
270 +            act (Any): Activation function
271 +        """
272 +        super().__init__()
273 +        self.stoch_size = stoch
274 +        self.deter_size = deter
275 +        self.hidden_size = hidden
276 +        self.act = nn.ELU
277 +        # self.act = self.act.to(device)
278 +        self.obs1 = nn.Linear(embed_size + deter, hidden)
279 +        self.obs1 = self.obs1.to(device)
280 +        self.obs2 = nn.Linear(hidden, 2 * stoch)
281 +        self.obs2 = self.obs2.to(device)
282 +
283
```

```
+        self.cell = nn.GRUCell(self.hidden_size, hidden_size=self.deter_size)
284     +        self.cell = self.cell.to(device)
285     +        self.img1 = nn.Linear(stoch + action_size, hidden)
286     +        self.img1 = self.img1.to(device)
287     +        self.img2 = nn.Linear(deter, hidden)
288     +        self.img2 = self.img2.to(device)
289     +        self.img3 = nn.Linear(hidden, 2 * stoch)
290     +        self.img3 = self.img3.to(device)
291     +
292     +        self.softplus = nn.Softplus
293     +        # self.softplus = self.softplus.to(device)
294     +
295     +
296     +
297     +    def get_initial_state(self, batch_size: int, device) -> List[TensorType]:
298     +        """Returns the inital state for the RSSM, which consists of mean,
299     +        std for the stochastic state, the sampled stochastic hidden state
300     +        (from mean, std), and the deterministic hidden state, which is
301     +        pushed through the GRUCell.
302     +        Args:
303     +            batch_size (int): Batch size for initial state
304     +        Returns:
305     +            List of tensors
306     +        """
307     +        return [
308     +            torch.zeros(batch_size, self.stoch_size).to(device),
309     +            torch.zeros(batch_size, self.stoch_size).to(device),
310     +            torch.zeros(batch_size, self.stoch_size).to(device),
311     +            torch.zeros(batch_size, self.deter_size).to(device),
312     +        ]
313     +
314     +    def observe(self,
315     +                embed: TensorType,
316     +                action: TensorType,
317     +                state: List[TensorType] = None
318     +                ) -> Tuple[List[TensorType], List[TensorType]]:
319     +        """Returns the corresponding states from the embedding from ConvEncoder
320     +        and actions. This is accomplished by rolling out the RNN from the
321     +        starting state through eacn index of embed and action, saving all
322     +        intermediate states between.
323     +        Args:
324     +            embed (TensorType): ConvEncoder embedding
325     +            action (TensorType): Actions
326     +            state (List[TensorType]): Initial state before rollout
327     +        Returns:
328     +            Posterior states and prior states (both List[TensorType])
329     +        """
330     +        if state is None:
331     +            state = self.get_initial_state(action.size()[0])
332     +
333     +        if embed.dim() <= 2:
334     +            embed = torch.unsqueeze(embed, 1)
335     +
336     +        if action.dim() <= 2:
337     +            action = torch.unsqueeze(action, 1)
338     +
339     +        embed = embed.permute(1, 0, 2)
340     +        action = action.permute(1, 0, 2)
341     +
342     +        priors = [[] for i in range(len(state))]
343     +        posts = [[] for i in range(len(state))]
344     +        last = (state, state)
345     +        for index in range(len(action)):
346     +            # Tuple of post and prior
347     +            last = self.obs_step(last[0], action[index], embed[index])
348     +            [o.append(s) for s, o in zip(last[0], posts)]
349     +            [o.append(s) for s, o in zip(last[1], priors)]
350
```

```
+
+        prior = [torch.stack(x, dim=0) for x in priors]
+        post = [torch.stack(x, dim=0) for x in posts]
+
+        prior = [e.permute(1, 0, 2) for e in prior]
+        post = [e.permute(1, 0, 2) for e in post]
+
+        return post, prior
+
+    def imagine(self, action: TensorType,
+                state: List[TensorType] = None) -> List[TensorType]:
+        """Imagines the trajectory starting from state through a list of actions.
+        Similar to observe(), requires rolling out the RNN for each timestep.
+        Args:
+            action (TensorType): Actions
+            state (List[TensorType]): Starting state before rollout
+        Returns:
+            Prior states
+        """
+        if state is None:
+            state = self.get_initial_state(action.size()[0])
+
+        action = action.permute(1, 0, 2)
+
+        indices = range(len(action))
+        priors = [[] for _ in range(len(state))]
+        last = state
+        for index in indices:
+            last = self.img_step(last, action[index])
+            [o.append(s) for s, o in zip(last, priors)]
+
+        prior = [torch.stack(x, dim=0) for x in priors]
+        prior = [e.permute(1, 0, 2) for e in prior]
+        return prior
+
+    def obs_step(
+            self, prev_state: TensorType, prev_action: TensorType,
+            embed: TensorType) -> Tuple[List[TensorType], List[TensorType]]:
+        """Runs through the posterior model and returns the posterior state
+        Args:
+            prev_state (TensorType): The previous state
+            prev_action (TensorType): The previous action
+            embed (TensorType): Embedding from ConvEncoder
+        Returns:
+            Post and Prior state
+        """
+        prior = self.img_step(prev_state, prev_action)
+        x = torch.cat([prior[3], embed], dim=-1)
+        x = self.obs1(x)
+        x = self.act()(x)
+        x = self.obs2(x)
+        mean, std = torch.chunk(x, 2, dim=-1)
+        std = self.softplus()(std) + 0.1
+        stoch = self.get_dist(mean, std).rsample()
+        post = [mean, std, stoch, prior[3]]
+        return post, prior
+
+    def img_step(self, prev_state: TensorType,
+                 prev_action: TensorType) -> List[TensorType]:
+        """Runs through the prior model and returns the prior state
+        Args:
+            prev_state (TensorType): The previous state
+            prev_action (TensorType): The previous action
+        Returns:
+            Prior state
+        """
+        x = torch.cat([prev_state[2], prev_action], dim=-1)
+
```

```
+         x = self.img1(x)
418     +         x = self.act()(x)
419     +         deter = self.cell(x, prev_state[3])
420     +         x = deter
421     +         x = self.img2(x)
422     +         x = self.act()(x)
423     +         x = self.img3(x)
424     +         mean, std = torch.chunk(x, 2, dim=-1)
425     +         std = self.softplus()(std) + 0.1
426     +         stoch = self.get_dist(mean, std).rsample()
427     +         return [mean, std, stoch, deter]
428     +
429     +     def get_feature(self, state: List[TensorType]) -> TensorType:
430     +         # Constructs feature for input to reward, decoder, actor, critic
431     +         return torch.cat([state[2], state[3]], dim=-1)
432     +
433     +     def get_dist(self, mean: TensorType, std: TensorType) -> TensorType:
434     +         return td.Normal(mean, std)
435     +
436     +
437     + # Dreamer Model
438     + class PLANet(nn.Module):
439     +     def __init__(self, obs_space, action_space, num_outputs, model_config,
440     +                  name, device):
441     +         super().__init__()
442     +
443     +         nn.Module.__init__(self)
444     +         #         'dreamer_model': {
445     +         #             'obs_space': [3, 64, 64],
446     +         #             'num_outputs': 1,
447     +         #             'custom_model': 'DreamerModel',
448     +         #             'deter_size': 200,
449     +         #             'stoch_size': 30,
450     +         #             'depth_size': 32,
451     +         #             'hidden_size': 400,
452     +         #             'action_init_std': 5.0,
453     +         #             },
454     +         self.depth = model_config["depth_size"]
455     +         self.deter_size = model_config["deter_size"]
456     +         self.stoch_size = model_config["stoch_size"]
457     +         self.hidden_size = model_config["hidden_size"]
458     +
459     +         self.action_size = action_space.shape[0]
460     +         self.device = device
461     +
462     +         self.encoder = Encoder(device, self.depth)
463     +         self.decoder = Decoder(device,
464     +             self.stoch_size + self.deter_size, depth=self.depth)
465     +         self.reward = DenseDecoder(device, self.stoch_size + self.deter_size, 1, 2,
466     +                                    self.hidden_size)
467     +         self.dynamics = RSSM(device,
468     +             self.action_size,
469     +             32 * self.depth,
470     +             stoch=self.stoch_size,
471     +             deter=self.deter_size)
472     +         self.actor = ActionDecoder(device, self.stoch_size + self.deter_size,
473     +                                    self.action_size, 4, self.hidden_size)
474     +         self.value = DenseDecoder(device, self.stoch_size + self.deter_size, 1, 3,
475     +                                   self.hidden_size)
476     +         self.state = None
477     +
478     +     def policy(self, obs: TensorType, state: List[TensorType], explore=True
479     +                ) -> Tuple[TensorType, List[float], List[TensorType]]:
480     +         """Returns the action. Runs through the encoder, recurrent model,
481     +         and policy to obtain action.
482     +         """
483     +         if state is None:
484     +
```

```
+            # self.state = self.get_initial_state(batch_size=obs.shape[0])
485 +            self.state = self.get_initial_state(self.device)
486 +        else:
487 +            self.state = state
488 +        post = self.state[:4]
489 +        action = self.state[4]
490 +
491 +        obs = obs.to(self.device)
492 +
493 +        embed = self.encoder(obs)
494 +        post, _ = self.dynamics.obs_step(post, action, embed)
495 +        feat = self.dynamics.get_feature(post)
496 +
497 +        action_dist = self.actor(feat)
498 +        if explore:
499 +            action = action_dist.sample()
500 +        else:
501 +            samples = []
502 +            for _ in range(1000):
503 +                samples.append(action_dist.sample())
504 +            action = torch.mean(torch.cat(samples), dim=0)
505 +            if action.ndim == 1:
506 +                action = action.unsqueeze(0)
507 +        logp = action_dist.log_prob(action)
508 +
509 +        self.state = post + [action]
510 +        return action, logp, self.state
511 +
512 +    def imagine_ahead(self, state: List[TensorType],
513 +                      horizon: int) -> TensorType:
514 +        """Given a batch of states, rolls out more state of length horizon.
515 +        """
516 +        start = []
517 +        for s in state:
518 +            s = s.contiguous().detach()
519 +            shpe = [-1] + list(s.size())[2:]
520 +            start.append(s.view(*shpe))
521 +
522 +        def next_state(state):
523 +            feature = self.dynamics.get_feature(state).detach()
524 +            action = self.actor(feature).rsample()
525 +            next_state = self.dynamics.img_step(state, action)
526 +            return next_state
527 +
528 +        last = start
529 +        outputs = [[] for i in range(len(start))]
530 +        for _ in range(horizon):
531 +            last = next_state(last)
532 +            [o.append(s) for s, o in zip(last, outputs)]
533 +        outputs = [torch.stack(x, dim=0) for x in outputs]
534 +
535 +        imag_feat = self.dynamics.get_feature(outputs)
536 +        return imag_feat
537 +
538 +    def get_initial_state(self, device) -> List[TensorType]:
539 +        self.state = self.dynamics.get_initial_state(1, device) + [
540 +            torch.zeros(1, self.action_size).to(device)
541 +        ]
542 +        return self.state
543 +
544 +    def value_function(self) -> TensorType:
545 +        return None
546 +
547 +
548 + class FreezeParameters:
549 +    def __init__(self, parameters):
550 +        self.parameters = parameters
551
```

```
+         self.param_states = [p.requires_grad for p in self.parameters]
+
+     def __enter__(self):
+         for param in self.parameters:
+             param.requires_grad = False
+
+     def __exit__(self, exc_type, exc_val, exc_tb):
+         for i, param in enumerate(self.parameters):
+             param.requires_grad = self.param_states[i]
```

138 mbrl/models/planet_legacy.py

```
@@ -0,0 +1,138 @@
+ ####################
+ ##### FOR GLEN ####
+ ####################
+ # This implementation of PlaNet servers as the world model for our Dreamer
+ # implementation.
+
+ ####################
+ ##### FOR GLEN ####
+ ####################
+ # This was used to do QA on the PlaNet implementation we ported from
+ # Chandramouli Rajagopalan's pytorch implementation of Dreamer here
+ # https://github.com/chamorajg/pl-dreamer , which has helper functions for the
+ # non-MBRL-Lib PlaNet, but do not get used with MBRL-Lib or Duckietown.
+ ########################################
+ import torch
+ import numpy as np
+ from typing import Tuple, Any, Union, Optional, Dict
+ import gym
+ TensorType = Any
+
+ class DMControlSuiteEnv:
+
+     def __init__(self,
+                  name: str,
+                  max_episode_length: int = 1000,
+                  action_repeat:int = 2,
+                  size: Tuple[int] = (64, 64),
+                  camera: Optional[Any] = None,
+                  ):
+         domain, task = name.split('_', 1)
+         if domain == 'cup':
+             domain = 'ball_in_cup'
+         if isinstance(domain, str):
+             from dm_control import suite
+             self._env = suite.load(domain, task)
+         else:
+             assert task is None
+             self._env = domain()
+         self._size = size
+         if camera is None:
+             camera = dict(quadruped=2).get(domain, 0)
+         self._camera = camera
+         self._step = 0
+         self._max_episode_length = max_episode_length
+         self._action_repeat = action_repeat
+
+     @property
+     def observation_space(self):
+         spaces = {}
+         for key, value in self._env.observation_spec().items():
+             spaces[key] = gym.spaces.Box(
+                 -np.inf, np.inf, value.shape, dtype=np.float32)
+             spaces['image'] = gym.spaces.Box(
+                 0, 255, self._size + (3,), dtype=np.uint8)
+
```

```
            return gym.spaces.Dict(spaces)
+
+    @property
+    def action_space(self):
+        spec = self._env.action_spec()
+        return gym.spaces.Box(spec.minimum, spec.maximum, dtype=np.float32)
+
+    def step(self, action):
+        reward = 0
+        obs = None
+        for k in range(self._action_repeat):
+            time_step = self._env.step(action)
+            self._step += 1
+            obs = dict(time_step.observation)
+            obs['image'] = self.render()
+            reward += time_step.reward or 0
+            done = time_step.last() or self._step == self._max_episode_length
+            if done:
+                break
+        info = {'discount': np.array(time_step.discount, np.float32)}
+        return obs["image"], reward, done, info
+
+    def reset(self):
+        time_step = self._env.reset()
+        self._step = 0
+        obs = dict(time_step.observation)
+        obs['image'] = self.render()
+        return obs["image"]
+
+    def render(self, *args, **kwargs):
+        if kwargs.get('mode', 'rgb_array') != 'rgb_array':
+            raise ValueError("Only render mode 'rgb_array' is supported.")
+        return self._env.physics.render(*self._size, camera_id=self._camera)
+
+ class Episode(object):
+    """ Episode Class which contains the related
+        attributes of an environment episode in the
+        the format similar to queue"""
+
+    def __init__(self,
+                 obs:TensorType,
+                 action_space: int = 1,
+                 action_repeat: int = 2) -> None:
+        """Initializes a list of all episode attributes"""
+        self.action_space = action_space
+        self.action_repeat = action_repeat
+        obs = torch.FloatTensor(np.ascontiguousarray(obs.transpose
+                                                     ((2, 0, 1))))
+        self.t = 1
+        self.obs = [obs]
+        self.action = [torch.FloatTensor(torch.zeros(1, self.action_space)).squeeze()]
+        self.reward = [0]
+        self.done = [False]
+
+    def append(self,
+                 episode_attrs: Tuple[TensorType]) -> None:
+        """ Appends episode attribute to the list."""
+        obs, action, reward, done = episode_attrs
+        obs = torch.FloatTensor(np.ascontiguousarray(obs.transpose
+                                                     ((2, 0, 1))))
+        self.t += 1
+        self.obs.append(obs)
+        self.action.append(action)
+        self.reward.append(reward)
+        self.done.append(done)
+
+    def reset(self,
```

```
+            obs:TensorType) -> None:
123 +        """ Resets Episode list of attributes."""
124 +        obs = torch.FloatTensor(np.ascontiguousarray(obs.transpose
125 +                                                     ((2, 0, 1))))
126 +        self.t = 1
127 +        self.obs = [obs]
128 +        self.action = [torch.FloatTensor(torch.zeros(1, self.action_space)).squeeze()]
129 +        self.reward = [0]
130 +        self.done = [False]
131 +
132 +    def todict(self,) -> Dict:
133 +        episode_dict = dict({'count': self.t,
134 +                             'obs': torch.stack(self.obs),
135 +                             'action': torch.cat(self.action),
136 +                             'reward': torch.FloatTensor(self.reward),
137 +                             'done': torch.BoolTensor(self.done)})
138 +        return
```

71 mbrl/planning/dreamer_wrapper.py

```
... ... @@ -0,0 +1,71 @@
1  + ####################
2  + ##### FOR GLEN ####
3  + ####################
4  + # This library uses SAC as a "planner" or "agent" for some dynamics models. We
5  + # tried to get Dreamer to be an agent similar to this approach. But the planner
6  + # isn't supposed to have access to the dynamics model for planning, but Dreamer
7  + # needs access to "imagine" future trajectories to determine what action to
8  + # take. We decided it would be simpler to instead put the planning part inside
9  + # the world model, and query the world model for actions as well.
10 +
11 + # We also need to differentiate through the dynamics model, using analytical
12 + # gradients and stop gradients - but were if unsure that the MBRL-Lib accounted
13 + # for this. All of these are design choices needing to be made before being neck
14 + # deep in an implementation, with a commitment made in algorithms/dreamer.py
15 + # before developing models/dreamer.py or planning/dreamer_wrapper.py. It would
16 + # seem usually you take your algorithm and use it in the library, but in this
17 + # case, you need to design your algorithm around MBRL with its training helper
18 + # functions.
19 +
20 + # This was the beginning of our properly MBRL-Lib philosophy- abiding version of
21 + # Dreamer, with a proper agent, based on the SAC implementation shown below.
22 + # Might be possible to add what we did as a third_party extension in this way as
23 + # well.
24 + ####################
25 +
26 + # Copyright (c) Facebook, Inc. and its affiliates. All Rights Reserved.
27 + #
28 + # This source code is licensed under the MIT license found in the
29 + # LICENSE file in the root directory of this source tree.
30 + import numpy as np
31 + import torch
32 +
33 + # import mbrl.third_party.pytorch_sac as pytorch_sac
34 + # import mbrl.third_party.pytorch_sac.utils as pytorch_sac_utils
35 +
36 + from .core import Agent
37 +
38 +
39 + class DreamerAgent(Agent):
40 +     """A Soft-Actor Critic agent.
41 +
42 +     This class is a wrapper for
43 +     https://github.com/luisenp/pytorch_sac/blob/master/pytorch_sac/agent/sac.py
44 +
45 +
46 +     Args:
47 +         (pytorch_sac.SACAgent): the agent to wrap.
```

```
+    """
+
+    def __init__(self, world_model = None, actor_network = None, critic_network = None):
+        self.world_model = world_model
+        self.actor_network = actor_network
+        self.critic_network = critic_network
+
+    def act(
+        self, obs: np.ndarray, sample: bool = False, batched: bool = False, **_kwargs
+    ) -> np.ndarray:
+        """Issues an action given an observation.
+
+        Args:
+            obs (np.ndarray): the observation (or batch of observations) for which the action
+                is needed.
+            sample (bool): if ``True`` the agent samples actions from its policy, otherwise it
+                returns the mean policy value. Defaults to ``False``.
+            batched (bool): if ``True`` signals to the agent that the obs should be interpreted
+                as a batch.
+
+        Returns:
+            (np.ndarray): the action.
+        """
+        return self.actor_network(obs)
```

15 mbrl/util/env.py

```
@@ -1,3 +1,9 @@
+ ####################
+ ##### FOR GLEN ####
+ ####################
+ # So here we just added a fixed start point for the Duckietown environment. That
+ # required modifying the Duckietown environment as well.
+ ####################
  # Copyright (c) Facebook, Inc. and its affiliates. All Rights Reserved.
  #
  # This source code is licensed under the MIT license found in the
@@ -92,7 +98,7 @@
         # Use the duckietown environment with settings that
         # closely match the settings used for cheetah run
         elif cfg.overrides.env == "duckietown_gym_env":
-            env = mbrl.env.DuckietownEnv(domain_rand=False, camera_width=64, camera_height=64)
+            # env = mbrl.env.DuckietownEnv(domain_rand=False, camera_width=64, camera_height=64, set_start_pos=n
+            env = mbrl.env.DuckietownEnv(
+                domain_rand=False,
+                camera_width=64,
+                camera_height=64,
+                set_start_pos=np.array([3.68680743, 0.0, 2.01533089]),
+                set_start_angle=np.array([-1.564463624086557])
+            )
             term_fn = mbrl.env.termination_fns.no_termination
             reward_fn = None
         else:
```

1 requirements/ducky.txt

```
@@ -21,3 +21,4 @@ torch==1.10.1
 torchvision==0.11.2
 tqdm==4.62.3
+typing_extensions==4.1.1
 wandb==0.12.9
```

30 sweep.py

```
@@ -0,0 +1,30 @@
+ ####################
+ ##### FOR GLEN ####
+ ####################
```

```
+ # This is just a script to run hyperparameter sweeps using weights and biases.
+ ####################
+
+ import os
+ import sys
+
+ base_command = f"python -m mbrl.examples.main algorithm=planet dynamics_model=planet overrides=planet_duckietown
+
+ final_hydra_command = base_command
+
+ for wandb_arg in sys.argv[1:]:
+     hydra_arg = wandb_arg[2:]
+     final_hydra_command += ' ' + f"{hydra_arg}"
+
+
+ print(final_hydra_command)
+ os.system(final_hydra_command)
+
+ # Alternate base commands:
+
+ # python -m mbrl.examples.main algorithm=planet dynamics_model=planet overrides=planet_duckietown algorithm.test
+
+ # python -m mbrl.examples.main algorithm=dreamer dynamics_model=dreamer overrides=dreamer_cheetah_run algorithm.
+
+ # python -m mbrl.examples.main algorithm=planet dynamics_model=planet overrides=dreamer_duckietown algorithm.tes
+
+ # python -m mbrl.examples.main algorithm=dreamer dynamics_model=dreamer overrides=dreamer_duckietown algorithm.t
```

Comparing changes

This is a direct comparison between two commits made in this repository or its related repositories. View the default comparison for this range here.

base: 3af2f04 ↔ compare: 8fa9fc6

Showing **9 changed files** with **262 additions** and **12 deletions**. Split | Unified

2 .gitignore

```
 7   7   bin
 8   8
 9   9   trained_models
    10 + saved_runs
    11 + wandb
```

21 .vscode/launch.json

```
...  ...  @@ -0,0 +1,21 @@
     1 + {
     2 +     // Use IntelliSense to learn about possible attributes.
     3 +     // Hover to view descriptions of existing attributes.
     4 +     // For more information, visit: https://go.microsoft.com/fwlink/?linkid=830387
     5 +     "version": "0.2.0",
     6 +     "configurations": [
     7 +         {
     8 +             "name": "Manual Control",
     9 +             "type": "python",
    10 +             "request": "launch",
    11 +             "python": "/home/kuwajerw/anaconda3/envs/dgym/bin/python",
    12 +             "program": "manual_control.py",
    13 +             "args": [
    14 +                 "--env-name",
    15 +                 "Duckietown-udem1-v0"
    16 +             ],
    17 +             "console": "integratedTerminal",
    18 +             "justMyCode": true
    19 +         }
    20 +     ]
    21 + }
```

34 gym_duckietown/simulator.py

```
...  ...  @@ -1,12 +1,14 @@
 1   1   # coding=utf-8
 2   2   from __future__ import division
 3   3
 4       - from collections import namedtuple
     4   + from collections import namedtuple, deque
 5   5   from ctypes import POINTER
 6   6   from dataclasses import dataclass
 7   7   from typing import Tuple
 8   8   # import geometry
 9   9
    10   + # import wandb
    11   +
10  12   @dataclass
11  13   class DoneRewardInfo:
12  14       done: bool
158 160              seed=None,
```

```
159  161              distortion=False,
160  162              randomize_maps_on_reset=False,
     163  +           set_start_pos=None,
     164  +           set_start_angle=None
161  165      ):
162  166          """
163  167
305  309              self.map_names = os.listdir('maps')
306  310              self.map_names = [mapfile.replace('.yaml', '') for mapfile in self.map_names]
307  311
     312  +
     313  +       self.set_start_pos = set_start_pos
     314  +       self.set_start_angle = set_start_angle
     315  +
308  316          # Initialize the state
309  317          self.reset()
310  318
513  521          self.cur_pos = propose_pos
514  522          self.cur_angle = propose_angle
515  523
     524  +       if self.set_start_pos is not None:
     525  +           self.cur_pos = self.set_start_pos
     526  +       if self.set_start_angle is not None:
     527  +           # self.cur_angle = self.set_start_angle
     528  +           # v Dunno why I need to do this but ugh v
     529  +           self.cur_angle = np.array([-1.56446362])
     530  +           xyz = 5
     531  +
     532  +       self.cur_poses = deque(maxlen=10)
     533  +       self.cur_angles = deque(maxlen=10)
     534  +       self.cur_poses.append(self.cur_pos)
     535  +       self.cur_angles.append(self.cur_angle)
     536  +
516  537          logger.info('Starting at %s %s' % (self.cur_pos, self.cur_angle))
517  538
518  539          # Generate the first camera image
1220 1241                                                  self.wheel_dist,
1221 1242                                                  wheelVels=self.wheelVels,
1222 1243                                                  deltaTime=delta_time)
     1244 +
     1245 +       self.cur_poses.append(self.cur_pos)
     1246 +       self.cur_angles.append(self.cur_angle)
     1247 +
1223 1248          self.step_count += 1
1224 1249          self.timestamp += delta_time
1225 1250
1324 1349
1325 1350              # Compute the reward
1326 1351              reward = (
1327      -                   +1.0 * speed * lp.dot_dir +
     1352 +                   # +1.0 * speed * lp.dot_dir +
1328 1353                      -10 * np.abs(lp.dist) +
1329 1354                      +40 * col_penalty
1330 1355              )
     1356 +           # Ali's reward:
     1357 +           delta_pos = self.cur_pos - self.cur_poses[0]
     1358 +           distance_travelled = np.linalg.norm(delta_pos)
     1359 +           # wandb.log({"dist_traveled": distance_travelled})
     1360 +           reward += distance_travelled*50
1331 1361          return reward
1332 1362
1333 1363      def step(self, action: np.ndarray):
```

4 learning/reinforcement/pytorch/ddpg.py

```
67  67          x = self.bn2(self.lr(self.conv2(x)))
68  68          x = self.bn3(self.lr(self.conv3(x)))
69  69          x = self.bn4(self.lr(self.conv4(x)))
```

```
70    -         x = x.view(x.size(0), -1)  # flatten
      70  +     x = x.reshape(x.size(0), -1)  # flatten
71    71        x = self.dropout(x)
72    72        x = self.lr(self.lin1(x))
73    73
129   129       x = self.bn2(self.lr(self.conv2(x)))
130   130       x = self.bn3(self.lr(self.conv3(x)))
131   131       x = self.bn4(self.lr(self.conv4(x)))
132   -         x = x.view(x.size(0), -1)  # flatten
      132 +     x = x.reshape(x.size(0), -1)  # flatten
133   133       x = self.lr(self.lin1(x))
134   134       x = self.lr(self.lin2(torch.cat([x, actions], 1)))  # c
135   135       x = self.lin3(x)
```

▾ 6 learning/reinforcement/pytorch/enjoy_reinforcement.py

```
6     6   import numpy as np
7     7
8     8   # Duckietown Specific
9     -   from reinforcement.pytorch.ddpg import DDPG
10    -   from utils.env import launch_env
11    -   from utils.wrappers import NormalizeWrapper, ImgWrapper, \
      9   + from learning.reinforcement.pytorch.ddpg import DDPG
      10  + from learning.utils.env import launch_env
      11  + from learning.utils.wrappers import NormalizeWrapper, ImgWrapper, \
12    12      DtRewardWrapper, ActionWrapper, ResizeWrapper
13    13
14    14
```

▾ BIN +19.6 MB learning/reinforcement/pytorch/models/planet.pth/model.pth

Binary file not shown.

▾ 193 learning/reinforcement/pytorch/test_algo.py

```
...   ...   @@ -0,0 +1,193 @@
      1   + ####################
      2   + ##### FOR GLEN ####
      3   + ####################
      4   + # This is just a script I used to help figure out how to record videos in
      5   + # Duckietown, and also how to use a trained agent to pick actions in Duckietown.
      6   + ####################
      7   + import ast
      8   + import argparse
      9   + import logging
      10  +
      11  + import os
      12  + from tkinter import W
      13  + import numpy as np
      14  +
      15  + from gym.wrappers import Monitor
      16  + from gym.wrappers.monitoring.video_recorder import VideoRecorder
      17  +
      18  + # Duckietown Specific
      19  + from learning.reinforcement.pytorch.ddpg import DDPG
      20  + from learning.utils.env import launch_env
      21  + from learning.utils.wrappers import NormalizeWrapper, ImgWrapper, \
      22  +     DtRewardWrapper, ActionWrapper, ResizeWrapper
      23  +
      24  + import wandb
      25  + import torch
      26  +
      27  + import errno
      28  + import os
      29  + from datetime import datetime
      30  +
      31  + import hydra
      32  + import omegaconf
```

```
+ from learning.reinforcement.pytorch.utils import planet_config_dict
+
+ cfg = omegaconf.OmegaConf.create(planet_config_dict)
+
+ # ADD THE MBRL LIB TO PATH SO YOU CAN IMPORT IT
+ import sys
+ sys.path.append('/home/kuwajerw/repos/duckietown-mbrl-lib')
+
+
+ import mbrl.algorithms.planet as planet
+ import mbrl.util.env
+ import mbrl.constants
+ from mbrl.env.termination_fns import no_termination
+ from mbrl.models import ModelEnv, ModelTrainer
+ from mbrl.planning import RandomAgent, create_trajectory_optim_agent_for_model
+
+ # folder in which to save logs
+ # will make timestamped folder in here
+ base_logdir = 'saved_runs'
+
+ # timestamp stuff
+ base_day_str = datetime.now().strftime('%Y-%m-%d')
+ base_time_str = datetime.now().strftime('%H-%M-%S')
+ curr_timestamp_str = base_day_str + '_' + base_time_str
+
+ # made this helper function to manually save the logs
+ # in addition to the W&B logs just in case
+ def make_save_folder(label=''):
+
+     # make timestamp logdir path
+     base_path = os.getcwd() + f'/{base_logdir}/'
+     mydir = os.path.join(
+         base_path,
+         base_day_str, base_time_str + "_" + label)
+
+     # actually create the folder
+     try:
+         os.makedirs(mydir)
+     except OSError as e:
+         if e.errno != errno.EEXIST:
+             raise  # This was not a "directory exist" error..
+
+     return mydir + "/"
+
+
+ def save_list(items, label, dest):
+     with open(os.path.join(dest, label), 'w') as f:
+         f.writelines(items)
+
+
+
+ # curr_timestamp_str = datetime.now().strftime('%Y-%m-%d_%H-%M-%S')
+
+ video_name = f"{curr_timestamp_str}.mp4"
+
+
+
+ def _enjoy():
+
+     # save run logs in folder that matches the wandb run name
+     logdir = make_save_folder(label=wandb.run.name)
+
+     # Launch the env with our helper function
+     env = launch_env()
+     print("Initialized environment")
+
+     # Wrappers
```

```
+     env = ResizeWrapper(env)
101  +     env = NormalizeWrapper(env)
102  +     env = ImgWrapper(env) # to make the images from 160x120x3 into 3x160x120
103  +     env = ActionWrapper(env)
104  +     env = DtRewardWrapper(env)
105  +     env = Monitor(env, logdir, force=True)
106  +     print("Initialized Wrappers")
107  +
108  +     # record video
109  +     video_rec = VideoRecorder(env, logdir + video_name)
110  +
111  +     state_dim = env.observation_space.shape
112  +     action_dim = env.action_space.shape[0]
113  +     max_action = float(env.action_space.high[0])
114  +
115  +     # PLANET STUFF ##############################
116  +
117  +     obs_shape = env.observation_space.shape
118  +     act_shape = env.action_space.shape
119  +     # Create PlaNet model
120  +     rng = torch.Generator(device=cfg.device)
121  +     rng.manual_seed(cfg.seed)
122  +     cfg.dynamics_model.action_size = env.action_space.shape[0]
123  +
124  +     planet = hydra.utils.instantiate(cfg.dynamics_model)
125  +     assert isinstance(planet, mbrl.models.PlaNetModel)
126  +     planet.load('learning/reinforcement/pytorch/models/planet.pth')
127  +     model_env = ModelEnv(env, planet, no_termination, generator=rng)
128  +     trainer = ModelTrainer(planet, optim_lr=1e-3, optim_eps=1e-4)
129  +
130  +     agent = create_trajectory_optim_agent_for_model(model_env, cfg.algorithm.agent)
131  +     # PLANET STUFF ##############################
132  +
133  +     # Initialize policy
134  +     # policy = DDPG(state_dim, action_dim, max_action, net_type="cnn")
135  +     # policy.load(filename='ddpg', directory='learning/reinforcement/pytorch/models/')
136  +
137  +     obs = env.reset()
138  +     done = False
139  +     reward_list = []
140  +     action_list = []
141  +
142  +     # PLANET STUFF ##############################
143  +     agent.reset()
144  +     planet.reset_posterior()
145  +     action = None
146  +     # PLANET STUFF ##############################
147  +
148  +     # while True:
149  +     # while not done:
150  +     for step in range(100):
151  +
152  +         # PLANET STUFF ##############################
153  +         planet.update_posterior(obs, action=action, rng=rng)
154  +         action_noise = 0
155  +         action = agent.act(obs) + action_noise
156  +         action = np.clip(action, -1.0, 1.0)  # to account for the noise
157  +         # PLANET STUFF ##############################
158  +
159  +         # action = policy.predict(np.array(obs))
160  +         # action = env.action_space.sample()
161  +         # Perform action
162  +         obs, reward, done, _ = env.step(action)
163  +
164  +         # log stuff with wandb
165  +         wandb.log({'step' : step})
166  +         wandb.log({'action0': action[0], 'action1' : action[1], 'global_step': step})
167
```

```
+         wandb.log({'reward' : reward, 'global_step': step})
168     +         wandb.log({'done' : int(done), 'global_step': step})
169     +
170     +         # env.render() # optional
171     +         video_rec.capture_frame() # record video
172     +
173     +         # maintain manual log just in case
174     +         reward_list.append(str(reward) + '\n')
175     +         action_list.append(str(action) + '\n')
176     +
177     +
178     +     # cleanup here
179     +     # done = False
180     +     # obs = env.reset()
181     +     video_rec.close()
182     +     env.close()
183     +
184     +     # save manual log
185     +     save_list(reward_list, f"rewards_{curr_timestamp_str}.txt", logdir)
186     +     save_list(action_list, f"actions_{curr_timestamp_str}.txt", logdir)
187     +
188     +
189     +
190     + if __name__ == '__main__':
191     +     wandb.init(project="MBRL_Duckyt", entity="mbrl_ducky", monitor_gym=True)
192     +     _enjoy()
193     +
```

10 learning/reinforcement/pytorch/train_reinforcement.py

```
6   6   import numpy as np
7   7
8   8   # Duckietown Specific
9       - from reinforcement.pytorch.ddpg import DDPG
10      - from reinforcement.pytorch.utils import seed, evaluate_policy, ReplayBuffer
11      - from utils.env import launch_env
12      - from utils.wrappers import NormalizeWrapper, ImgWrapper, \
    9   + from learning.reinforcement.pytorch.ddpg import DDPG
    10  + from learning.reinforcement.pytorch.utils import seed, evaluate_policy, ReplayBuffer
    11  + from learning.utils.env import launch_env
    12  + from learning.utils.wrappers import NormalizeWrapper, ImgWrapper, \
13  13      DtRewardWrapper, ActionWrapper, ResizeWrapper
14  14
15  15  logger = logging.getLogger(__name__)
138 138     parser.add_argument("--policy_freq", default=2, type=int)  # Frequency of delayed policy updates
139 139     parser.add_argument("--env_timesteps", default=500, type=int)  # Frequency of delayed policy updates
140 140     parser.add_argument("--replay_buffer_max_size", default=10000, type=int)  # Maximum number of steps to keep
141     -     parser.add_argument('--model-dir', type=str, default='reinforcement/pytorch/models/')
    141 +     parser.add_argument('--model-dir', type=str, default='learning/reinforcement/pytorch/models/')
142 142
143 143     _train(parser.parse_args())
```

4 manual_control.py

```
38  38          domain_rand = args.domain_rand,
39  39          frame_skip = args.frame_skip,
40  40          distortion = args.distortion,
    41  +       set_start_pos=np.array([3.68356218, 0., 1.50287902]),
    42  +       set_start_angle=np.array([-1.564463624086557])
41  43      )
42  44  else:
43  45      env = gym.make(args.env_name)
98  100
99  101     obs, reward, done, info = env.step(action)
100 102     print('step_count = %s, reward=%.3f' % (env.unwrapped.step_count, reward))
    103 +   print(f"curr_pos is {env.cur_pos} and cur_angle is {env.cur_angle}")
    104 +
```

```
101  105
102  106      if key_handler[key.RETURN]:
103  107          from PIL import Image
```